UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

Accelerated Garbled Scaling of Residue Representations with Applications to Secure Machine Learning

Masterarbeit

im Rahmen des Studiengangs
Informatik
der Universität Hamburg

vorgelegt von
Felix Maurer

ausgegeben und betreut von
Prof. Dr. Thomas Eisenbarth

mit Unterstützung von
M. Sc. Jonas Sander
Prof. Dr. Mathias Fischer

Hamburg, den 13. August 2024

Abstract

As the demand for secure and privacy-preserving distributed machine learning solutions grows across numerous areas of application, the importance of enhancing the efficiency and applicability of such solutions becomes increasingly paramount. The DASH framework approaches the problem of secure machine learning by deploying optimized arithmetic garbled circuits, where large quantized floating-point inputs are scaled down to smaller integer domains for deeper neural networks. This allows DASH to securely operate on more complex network topologies, enabling a broader range of applications. Previous research has identified that rescaling during circuit evaluation poses a significant performance bottleneck. To address this, we propose an improvement to DASH's scaling operation on integer residues by generalizing beyond the current limitations of the scaling factor. This leads to enhanced network inference speed and memory efficiency: Our solution improves both metrics by a logarithmic factor. Furthermore, as a side contribution, we showcase a garbled pooling operation for DASH, which further expands the range of supported neural network layouts, thereby increasing the versatility of the DASH framework.

Acknowledgements

I want to thank Jonas for being an excellent supervisor and mentor while writing this thesis, always taking the time to discuss theoretic approaches in detail and helping me out whenever I was stuck during implementation. Furthermore, I would like to express my thanks to Thomas and Mathias for making this collaboration between UHH and UzL possible and to the entire ITS crowd in Lübeck for welcoming and supporting me.

Contents

1 Introduction

1.1 Motivation

The application of machine learning solutions has become a pivotal method for analyzing expansive datasets, particularly in domains where conventional algorithmic approaches prove insufficient. These domains encompass a diverse range of applications, including image recognition, genome analysis, finance, and autonomous driving. The significance of data privacy is evident in these applications, with certain cases subject to regulations such as the GDPR [Com16]. The escalating prevalence of data breaches, especially in sensitive areas like healthcare [FGR16], underscores the critical need for the development and implementation of robust and effective security measures.
In various scenarios, multiple parties collaborate on machine learning projects, necessitating the protection of confidential data from unauthorized access. This scenario is commonly referred to as *secure machine learning*. In fields like medical research, where client data and machine learning models are often kept separate, secure machine learning collaborations can benefit both clients and researchers.
One secure machine learning solution called DASH utilizes garbled circuits (GCs) as its primary cryptographic tool and was developed by Sander et al. [SBBE23a]. It advances previous results by Ball et al. [BMR16, BCM+19]. This thesis aims to build upon DASH's current implementation and introduce improvements to their solution.

1.2 Related Work

Several other secure neural network (NN) inference implementations have utilized GCs before the DASH approach. For example, in 2017, Mohassel et al. [MZ17] incorporated GCs into select components of their *SecureML* framework. Later, Rouhani et al. [RRK18] developed *DeepSecure*, which marked a significant milestone as the first viable GC-only solution for this problem. Additionally, there have been notable hybrid approaches that combine Fully Homomorphic Encryption (FHE) schemes and GCs, focusing on using the latter only for the non-linear NN components such as activation functions. Two such examples are *Gazelle*[JVC18] and *Delphi*[MLS+20]. Furthermore, FHE-exclusive approaches, such as *CryptoNets*[GBDL+16] and its successor *Faster CryptoNets*[CBL+18], have also been explored, aiming to approximate all non-linear NN components.

1.3 Research Questions

The goal of this thesis is to improve the capabilities of the DASH framework by two distinct metrics: *Firstly*, we aim to improve DASH's runtime performance in order to make a further step towards user-friendly secure machine learning inference. *Secondly*, we try to make DASH compatible with more real-world examples of (pre-trained) models by implementing currently unsupported NN layer classes. Such layers currently have to be approximated and replaced by other NN techniques, hindering the real-world applicability of DASH.
Performance benchmarks conducted with realistic deep NN topologies indicated a significant performance bottleneck induced by repeated scaling of discretized feature vectors in DASH, cf. Figure 1.1.

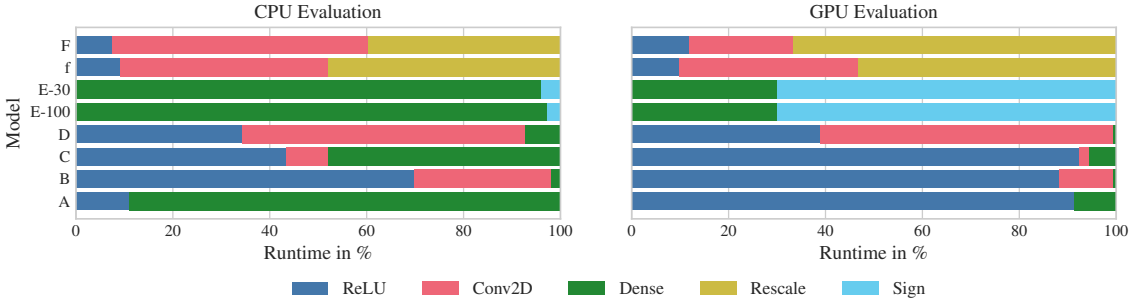

Figure 1.1: Runtime distribution by layer type for various NN topologies. Models F and f are the deepest and require rescaling of feature vectors during inference. Measurements conducted by [SBBE23a].

This limitation is caused directly by a limitation to the scaling factor, as will be discussed in detail in Chapter 3. Therefore, our first research question is:

1. *What techniques could advance runtime performance of the secure machine learning framework DASH? How can we generalize DASH's scaling-by-two operation to scale by a wider range of scaling factors?*

DASH's secure machine learning approach is geared towards convolutional NNs, which, in most cases, deploy a pooling operation after each convolution layer. While in theory, this pooling operation can be circumvented by adjusting the convolutions' stride parameters and retraining the model, supporting a wide range of pre-trained models would improve DASH's applicability. We thus arrive at our secondary research question:

2. *How can we implement a (max-)pooling operation using arithmetic GCs and integrate it in DASH?*

1.4 Thesis Structure

Before explaining the DASH framework in detail in Chapter 3, we will first provide an overview of the theoretical background relevant to our work. This includes discussing the fundamentals of NNs, residue number systems (RNS), and GCs in Chapter 2. After contextualizing the state-of-the-art of the DASH framework, we will then detail our central theoretical result - a novel approach to garbled base extension (BE) for arbitrary moduli - in Chapter 4. An in-depth examination of the implementation details for our solutions to both research questions will be presented in Chapter 5. Finally, in Chapter 6, we will thoroughly evaluate our approach from both a theoretical and experimental perspective before providing a conclusion and discussing possible directions for future work in Chapter 7.

2 Preliminaries

Our work revolves around the application of a cryptographic approach to the context of machine learning. This cryptographic approach combines the GC protocol with residue number representations. In this chapter, we will introduce some general machine learning preliminaries, then discuss RNS on an abstract level, and finally showcase our cryptographic protocol.

2.1 Secure Neural Network Inference

The discipline of secure machine learning constitutes a subset of the broader field of *secure function evaluation*. This field focuses on enabling multiple parties to jointly compute a function without revealing their individual inputs to one another: Alice and Bob each hold one component of an input tuple (a, b) to a function f. They seek to evaluate $f(a, b)$ while not leaking their input contribution to the other computing party. This problem can be generalized for an arbitrary number of parties with respective inputs $a, b, c, \ldots$. In the case of two input parties, this problem is also referred to as *secure two-party computation*, while the general case is referred to as *secure multi-party computation*. In our case, that is in the context of secure machine learning, f may either be

1. the classification function $\Phi(\mathbf{x}, \theta)$, to which Alice provides the trained model θ, and Bob provides an input feature vector $\mathbf{x}$ that shall be classified or
2. the training function dictating how the weight parameters of θ shall be adjusted, given the current state of the trained model θ and the training input data provided by Bob.

In this thesis, we will explore an approach to the first case, where during the classification phase, the input data $\mathbf{x}$ shall remain inaccessible to the public, while its classification result may optionally be queriable by one or more parties. Furthermore, the provider of the model parameters θ (the classifying party) shall never know the contents of $\mathbf{x}$, and neither should any user other than the provider of $\mathbf{x}$. This approach aims to preserve the privacy of the input data while enabling the classification task to be performed in a secure and controlled manner.
In some cases, θ is not met with the same treatment as $\mathbf{x}$ regarding confidentiality. This is, for example, currently the case in DASH, the framework this thesis extends. The original

authors of DASH argued that due to common NN model extraction attacks [CJM20], this goal is unrealistic in most scenarios.

Now let us discuss some formal preliminaries regarding NNs, including the already mentioned model θ and the network's input feature vector $\mathbf{x}$.

Neural Networks

The concept of a *neural network* (NN) serves as the theoretical foundation for a multitude of modern machine learning achievements, including image and face recognition, autonomous driving, and search algorithms. NNs offer a powerful approach to approximating complex, unknown mappings by utilizing an iterative training procedure that generates a range of feature correlations. This iterative training allows NNs to surpass the accuracy and expressivity of traditional training methods, such as linear regression. The objective of this thesis is to apply cryptographic techniques to such NNs, thereby enabling secure machine learning. To accomplish this, we will first need to define the fundamental principles of both fully-connected and convolutional NNs, which are widely used in a variety of machine learning applications.

The core architectural components of a traditional NN are fully connected (dense) layers and activation layers, which are applied in an alternating sequence during network evaluation. Each layer $\Phi^{(1)}, .., \Phi^{(L)}$ processes an input vector $\mathbf{x} \in \mathbb{R}^{N_l}$ and transforms them by various means in *neurons* $\Phi_1^{(l)}, .., \Phi_{N_l}^{(l)}$, with N_l being the number of neurons of the lth fully connected/activation layer and L the total number of such layer pairs.

The fully connected layers perform a linear transformation on the input vector, consisting of a matrix multiplication with a corresponding weight matrix W followed by the addition of a bias vector $b \in \mathbb{R}^{R_l}$. In contrast, the network's activation layers apply a non-linear transformation, using a unary activation function $\varrho : \mathbb{R} \to \mathbb{R}$, to the output of the preceding fully connected layer. A widespread choice for activation is, for instance, the *Rectified Linear Unit* (*ReLU*) defined as $ReLU(x) = \max(0, x)$, which we will later encounter again in an entirely different setting in Section 5.1. The non-linearity of the $ReLU$ function is depicted in Figure 2.1. We now arrive at the following formal definition of (deep) fully connected NNs.

Definition 2.1.1 (Fully Connected Neural Network). A *fully connected neural network* with architecture $N_0, .., N_L$ and activation function ϱ is a mapping $\Phi(\mathbf{x}, \theta) \in \mathbb{R}^{N_L}$ with *input* $\mathbf{x} = (\mathbf{x}_1, .., \mathbf{x}_{N_0})$ and *weight and bias vectors* $\theta = (\theta^{(l)})_{l=1}^{L} = ((W^{(l)}, b^{(l)}))_{l=1}^{L}$.

The result of the mapping is referred to as the *output layer* of the network and is defined

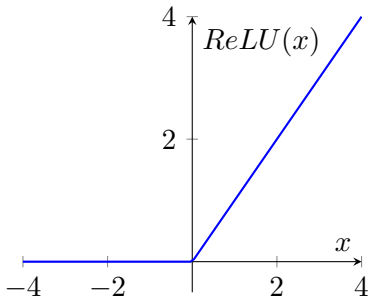

Figure 2.1: Plot of $ReLU(x)$ for $x \in [-4, 4]$.

as $\Phi(\mathbf{x}, \theta) = \Phi^{(L)}(\mathbf{x}, \theta)$, where for all $l \in [L-1]$

$$\Phi^{(1)}(\mathbf{x}, \theta) = W^{(1)}\mathbf{x} + b^{(1)} \qquad (\textit{input layer}), \tag{2.1}$$

$$\overline{\Phi}^{(l)}(\mathbf{x}, \theta) = \varrho(\Phi^{(l)}(\mathbf{x}, \theta)), \qquad (\textit{activation layer}), \tag{2.2}$$

$$\Phi^{(l+1)}(\mathbf{x}, \theta) = W^{(l+1)}\overline{\Phi}^{(l)}(\mathbf{x}, \theta) + b^{(l+1)} \qquad (\textit{fully connected layer}), \tag{2.3}$$

Note that in Equation 2.2 the function $\varrho : \mathbb{R} \to \mathbb{R}$ is applied element-wise. If $L > 3$, the network architecture of Φ is referred to as *deep*. All layers that are neither input nor output layers are referred to as *hidden layers*.

Such networks can be visualised using a graph representation, depicting neurons as vertices and weighted correlations as edges, cf. Figure 2.2.

Training

The accuracy of a NN directly depends on how its model parameters θ are selected. To increase an NN's accuracy, one must find values for θ that yield results closer to the correct solution of the problem at hand, such as image classification that is statistically more often correct. To quantify this error, a data set $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})_{i=1}^{\widetilde{m}}$ consisting of *input data* and *label* pairs is required. Otherwise, no notion of the correct solution could be derived. This data set must then be split into *training data* $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})_{i=1}^{m}$ and *testing data* $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})_{i=m}^{\widetilde{m}}$: The training data will be used to determine a better choice for θ, while the testing data set serves to verify this result. The exact choice of m is a matter of preference, though generally, the training data set is larger than the testing data set, i.e., $m > \frac{\widetilde{m}}{2}$. The idea of what constitutes a better set of parameters, i.e., a better θ, must now be more formally defined. This whole process is an optimization task that can be reduced to decreasing the *empirical risk* of failure, as defined below. Specifically, the empirical risk refers to the measure of how well the model performs on the observed training data, and the objective

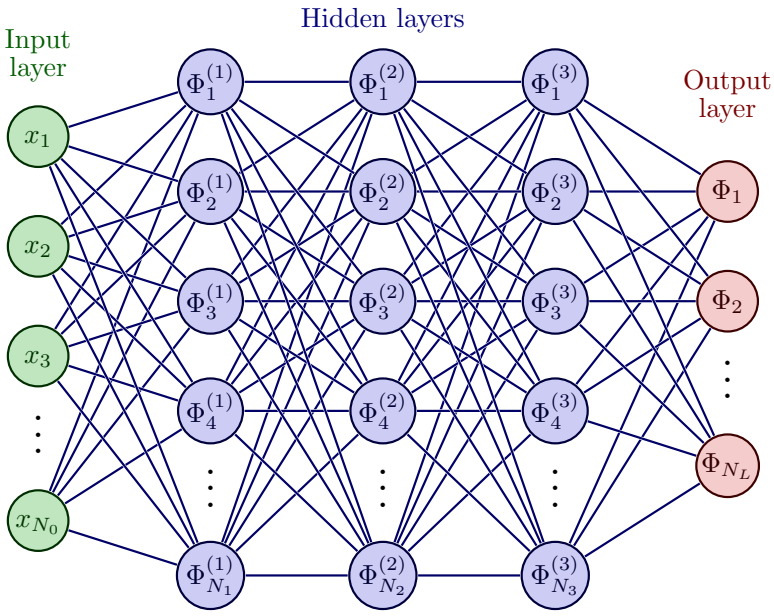

Figure 2.2: Graph representation of deep fully connected NN Φ with $L = 4$. Activation layers ($\overline{\Phi}$) are not displayed.

is to find the optimal parameters that minimize this empirical risk.

Definition 2.1.2 (Empirical Risk). The *empirical risk* $\mathcal{R}$ of a neural network Φ with weights and biases θ is defined as

$$\mathcal{R}(\Phi, \theta) = \frac{1}{m} \sum_{i=1}^{m} \left(\Phi(\mathbf{x}^{(i)}, \theta) - \mathbf{y}^{(i)} \right)^2 \tag{2.4}$$

where $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})_{i=1}^{m}$ is the training data set of Φ.

The metric $\mathcal{R}$ can be considered the mean squared error between the model output $\Phi(\mathbf{x}^{(i)}, \theta)$ and the target values $\mathbf{y}^{(i)}$. To evaluate the accuracy of the newly trained network, we determine to what extent the goal of our network

$$\forall i = m+1, ..., \widetilde{m} : \Phi(\mathbf{x}^{(i)}, \theta) \approx \mathbf{y}^{(i)} \tag{2.5}$$

is attained. There are different ways of solving the above-mentioned empirical risk-based optimization problem. Traditionally, gradient descent on the risk function $\mathcal{R}$ or a variation of it that does not take into consideration all point-wise derivatives but only a subset, called *stochastic gradient descent* [RM51], is used here. Such derivatives can accurately be determined by a numerical algorithm called *backpropagation*, details on which can be

found in [GW08]. As we will not be training the model parameters θ ourselves, we will not further discuss these optimization algorithms in detail.

Convolutional Neural Networks

Convolutional NNs (CNNs) form a new class of NNs and are what DASH's secure inference is built for. We are no longer limiting feature vectors to be one-dimensional, i.e. generalize $\mathbf{x}$ to be a *tensor*. You can also apply this generalization to FCNNs, for instance when operating on two-dimensional image data. In literature, such tensors are still called feature vectors regardless of dimensionality, and therefore we stick to this nomenclature.
When identifying local attributes in input data, such as in the previously mentioned image classification example, it becomes apparent that dense layers provide correlations that may prove inconsequential during the inference process. Reducing these correlations would, while not enhancing the neural network's expressive capacity, significantly reduce the size of θ, enabling the deployment of deeper and wider network topologies. In numerous cases, local patches or *kernels* of limited size are sufficient to establish a relationship between an input neuron and other neurons in its immediate vicinity. Furthermore, the features detected in this manner exhibit invariance to primitive transformations like rotation or translation.
A widely utilized operation in this context is *convolution*, which originates from the field of image processing. For a pair two-dimensional tensors $w = (w_{-a,-b}, ..w_{a,b})$ and $= (x_{1,1}, ..x_{c,d})$ (e.g. a weighted kernel and an image) it is defined as

$$(w * x)(x, y) = \sum_{(dx,dy)=(-a,-b)}^{(a,b)} (w_{dx,dy}) \cdot (x_{x-dx,y-dy}). \tag{2.6}$$

The use of convolutions in NNs was first proposed by Fukushima in 1980 [Fuk80] and then significantly advanced by LeCun et al. in 1989 [LBD+89], who introduced the modern architecture we will now explain in further detail. Usually, the goal is not to extract a single local property using convolutional filtering but rather to capture a list of features. This is modeled by a list of kernels $k_1, .., k_C$, where $C \in \mathbb{N}$ represents the number of *channels* or *feature maps* in that layer. In that case, the above mentioned kernel list replaces the weight and bias pair $\theta^{(l)}$, i.e. for a convolutional layer $\theta^{(l)} = (k_i)_{i=1}^{C}$.
This convolutional operation is then followed by a *pooling operation* p reducing the size of each individual feature map in all dimensions by either averaging, finding the maximum element or other means of reducing multiple elements (e.g. 2×2 entries) to one. The pooling operation is parametrized in terms of *kernel sizes* per dimensionality and a *stride* value that determines the step size by which this kernel moves across the input data.

For example, if the stride value is smaller than the pooling operation's kernel sizes, the kernels overlap. As we will later discuss our implementation of (garbled) max-pooling in Chapter 5.1, the corresponding function p_{max} for two-dimensional input and kernel sizes $K_X, K_Y = 2$ in visualized in Figure 2.3.

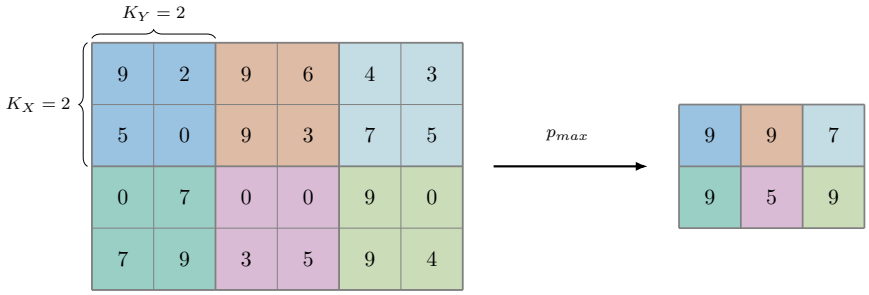

Figure 2.3: Visual representation of two-dimensional p_{max} for $K_X, K_Y = 2$ on examplary 2×3 integer input. Strides are identical to kernel sizes and are not explicitly depicted.

Down-sampling of the convolution's feature maps serves the purpose of making the network more robust to slight variations in input data patterns. For example, in image recognition, pooling layers even out small translations (e.g. rotation, shift) during image feature extraction. Furthermore, it limits layer input size expansion. This leads us to the following definition of a convolutional NN:

Definition 2.1.3 (Convolutional Layer and CNN). In a neural network, a *convolutional layer* $\Phi^{(l)}$ with $C \in \mathbb{N}$ channels and kernel list $\theta^{(l)}$ is defined as

$$\Phi^{(l)}(\mathbf{x}, \theta) = (p(\overline{\Phi}^{(l)}(\mathbf{x} * k_i, \theta)))_{i1}^{C}, \tag{2.7}$$

where $k_1, .., k_C \in \theta^{(l)}$ and p is a pooling operation. A *convolutional neural network (CNN)* is a neural network containing one or more convolutional layers.

Typically, a CNN comprises multiple convolutional layers, which are then flattened, reducing the multi-dimensional collection of feature maps to a one-dimensional vector. This vector is subsequently fed into a standard, fully connected neural network, as outlined in Section 2.1.1. Analogously to the visual depiction of an FCNN found above in Section 2.1.1, a visual representation of a convolutional NN is depicted in Figure 2.4.

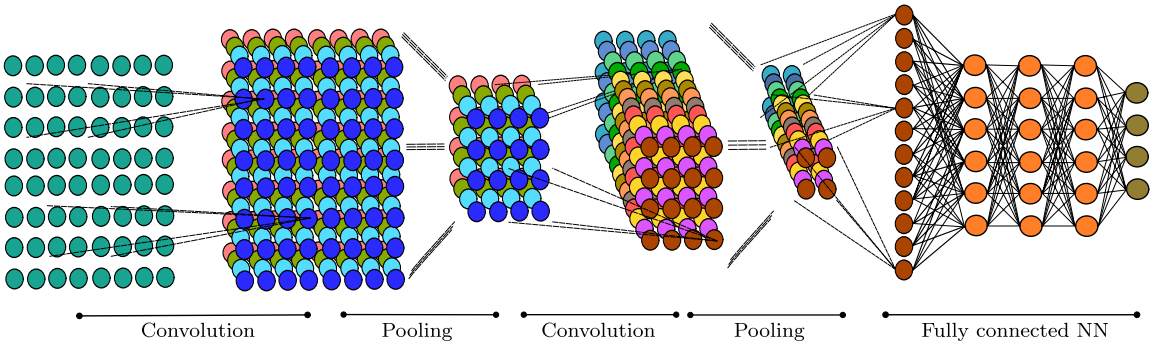

Figure 2.4: Graph representation of a deep convolutional neural network. Image taken from [BGKP21].

2.2 Residue Number Systems

In our secure machine learning solution we apply cryptographic means to discretized integer representations x of elements in feature vectors denoted by us as $\mathbf{x}$. While the exact protocol details will be given later in Section 2.3, it suffices to know that we map a large number of cleartext integers to encrypted ciphertexts, which expand drastically for larger x. It would thus be beneficial if we could instead of encrypting one large x, opt for several smaller $x_1, ..x_k$ that store identical information. In order to avoid confusion, we reiterate: $\mathbf{x}$ are feature vectors, x are discretized *elements* of such a feature vector, and now we try to express one x by storing its information in multiple smaller x_i.

The Chinese Remainder Theorem

Luckily, a well-known result in number theory dating back to at least the fifth century allows us to do just that by utilizing *modular residues*. Let P_k be the k-th *primal modulus*, i.e. the product of the first k prime numbers. We can represent all members of $\mathbb{Z}/P_k\mathbb{Z} =: \mathbb{Z}_{P_k}$ using their modular residue respective 2,3,...,p_k where p_k is the k-th prime number:

Theorem 1 (Chinese Remainder Theorem). For arbitrary coprime *moduli* $p_1, .., p_k$ and respective *residues* $[x]_1, .., [x]_k$, there exists a unique x such that

1. $0 < x < \prod_{i=1}^{n} p_k$ and
2. $[x]_i = x \mod p_i$ for all $i = 1, .., n$.

Note that for the Chinese Remainder Theorem (CRT) it suffices that all p_i are coprime, which is obviously the case for our choice of moduli. This *CRT representation* or *residue representation* $([x]_i, .., [x]_k)$, often referred to as a member of a *residue number system* (RNS)

$(p_1, .., p_k)$, can be transformed back to its integer representation x efficiently as seen in Equation 2.8:

$$x = \sum_{i=1}^{k} \alpha_i \cdot [x]_i \mod P_k, \tag{2.8}$$

where $\alpha_i = A_i^{-1} \cdot A_i$ and $A_i = \frac{P_k}{p_i}$. An examplary RNS with $k = 3$ can be found in Table 2.1.

x	$[x]_1$	$[x]_2$	$[x]_3$
0	0	0	0
1	1	1	1
2	0	2	2
3	1	0	3
⋮	⋮	⋮	⋮
26	0	2	1
27	1	0	2
28	0	1	3
29	1	2	4

Table 2.1: Residue representation of $x = 0, .., 29$ with moduli $p_1 = 2, p_2 = 3, p_3 = 5$.

Residue Arithmetic

Modular addition and multiplication on (all of) these residues can be trivially constructed by mapping the integer operation to the integers' residue representations. Let $\star \in \{+, \cdot\}$. Then for some x, y with respective residues $([x]_1, .., [x]_k)$ and $([y]_1, .., [y]_k)$

$$x \star y \mod P_k \longleftrightarrow ([x]_1 \star [y]_1 \mod p_1, .., [x]_k \star [y]_k \mod p_k), \tag{2.9}$$

where $\longleftrightarrow$ means equivalence by the means of the CRT. This also covers subtraction, as modular subtraction by y is equivalent to adding $P_k - y$, i.e the additive modular inverse of y.

One central drawback in using residue representations is that many other operations are not as trivially constructable. Most importantly for our purposes is the difficulty of modeling division in RNS. In this thesis, we will bring the special case of *scaling*, that is division by arbitrary p_i, to DASH's arithmetic GCs in order to enable downsampling of intermediate values in NN inference. Further detail and motivation behind (garbled) scaling will be given when discussing DASH's current scaling capabilities in Chapter 3.

Associated Mixed-Radix Systems

We will now describe a central building block of our solution regarding the problem of generalized BE, namely *mixed-radix systems* (MRS). Note that this section will not contain its application, i.e. the BE algorithm, as this will be done later when describing our approach in Chapter 4.

Usually, number systems share one radix across all digits of a number, for instance 2 in the binary number system or 10 in the decimal one. Even though it may seem odd at first, a multi-digit representation of some integer x can just as well be expressed using potentially varying radices $R_1, ..R_{N-1}$ where N is the number of digits of x in *mixed radix* form. More precisely, any $x < \prod_{i=1}^{N} R_i - 1$ can be expressed as

$$x = a_N \prod_{i=1}^{N-1} R_i +, .., +a_3 R_1 R_2 + a_2 R_1 + a_1, \tag{2.10}$$

where a_i are the mixed radix digits of x for some radices R_i. Generally, radices act as a carry-over from less significant digit positions $1, .., i-1$ to the respective i. This becomes intuitively clear when picturing the above mentioned everyday examples of $R_1, R_2, .. = 2$ (the binary number system) or $R_1, R_2, .. = 10$ (the decimal number system). An example for nonidentical $R_1 = 2, R_2 = 3, R_3 = 5$ can be found in Table 2.2.

x	a_1	a_2	a_3
0	0	0	0
1	1	0	0
2	0	1	0
3	1	1	0
⋮	⋮	⋮	⋮
26	0	1	4
27	1	1	4
28	0	2	4
29	1	2	4

Table 2.2: Mixed radix representation of $x = 0, .., 29$ with radices $R_1 = 2, R_2 = 3, R_3 = 5$.

For each RNS $(p_1, ..p_k)$ there exists a unique MRS with $R_i = p_i$ called the *associated* MRS for this RNS. For instance, the MRS depicted in Table 2.2 is the associated MRS of a RNS with $k = 3$. We can construct its digits $a_1, ..a_k$ as follows [ST67]:

$$a_i = \begin{cases} [x]_1, & \text{for } i = 1, \\ \frac{x - a_{i-1}}{p_{i-1}} \mod p_i, & \text{otherwise.} \end{cases} \tag{2.11}$$

This general recursive property presented in Equation 2.11 is the foundational idea of the BE algorithm later discussed in Section 4.

2.3 Garbled Circuits

The central cryptographic protocol behind DASH and many alternative solutions to secure machine learning inference is the *garbled circuit* (GC) protocol. As this is where all possible optimizations to the current framework lie, an overview of the protocol and its continous advancements during the last 35 years will now be given.

Yao's Protocol

One solution to secure MPC was proposed by Yao[Yao86] in 1986: Given some operation in binary circuit representation, one party (the *garbler*) transforms the non-secure variant of each of the circuit representation's gates $f : \mathbb{Z}_2 \to \mathbb{Z}_2$ to a secure (*garbled*) variant. For each of the two input wires two encrypted *input labels* l^0 and l^1 are chosen randomly representing 0 and 1. In similar fashion, *output labels* for the gate's output wire are chosen. Following this, the garbler produces an encrypted truth table using the input wires as symmetric encryption keys by computing $\text{Enc}_{l^a_{in_1}, l^b_{in_2}}(l^{f(a,b)}_{out})$ for all $a, b \in \mathbb{Z}_2$ where in_1, in_2 are the garbled gate's input wires and out is its output wire.
The central idea is that during evaluation, only the output label corresponding to a given (encrypted) label input can be decrypted. To prevent cleartext output deduction, the truth table ordering must be randomized. During the actual multi-party computation, the GC is traversed by an *evaluator* party that only learns the (garbled) output corresponding to one input, but not the other parties' input or other possible outcomes of the evaluation, i.e. the circuit's underlying function. GCs can only be evaluated once, since otherwise the evaluator may be able to deduct which labels correspond to 0 or 1. Note that from now on we will assume that each (binary) GC only consists of AND and XOR gates, as arbitrary boolean functions can be constructed from these two operations.
During evaluation, the two parties usually communicate via a 1-out-of-2 *oblivious transfer* (OT), where neither party learns the (binary) input choice of the other.

Garbled Circuit Optimizations

Several optimizations to the above mentioned initial protocol proposed by Yao have been made to make GCs more efficient while remaining secure. While for our purposes the most central change is the generalization from binary GCs to arithmetic GCs, we first need to describe three central advancements for the former:

1. *Point-And-Permute* (Beaver et al. [BMR90]) While the ordering of (encrypted) truth table entries in the original protocol is not known to the evaluator, this technique allows to determine which of the four truth table entries to decrypt by appending a *color bit* $p \in \mathbb{Z}_2$ to the gate's input labels.

2. *Free-XOR* (Kolesnikov et al. [KS08]) By choosing $l_1 = l_0 \oplus R$ for some pre-defined circuit wide constant R, we can evaluate garbled XOR gates by simply executing a single XOR operation on garbled input labels without any additional computational overhead, i.e. compute $x \oplus y$ by evaluating $l_x^0 \oplus l_y^0$.

3. *Half-Gates* (Zahur et al. [ZRE15]) This optimization allowing for AND gates to only require two garbled ciphertexts (distributed between the garbling and evaluating parties) is the current state-of-the-art in regards to efficient AND gate evaluation in GCs. Unlike Free-XOR, this optimization does not result in free operations, meaning that AND gates still present a bottleneck for binary GCs.

Arithmetic Garbled Circuits

Arithmetic GCs are derived from traditional (binary) GCs by generalizing these three central advancements. The motivation for this generalization is that the major workload of NNs consists of arithmetic operations. Even though arithmetic GCs do not allow for arithmetics over $\mathbb{R}$, *quantization* techniques (as introduced in Section 3.2) enable us to map floating point values to integer rings. The following generalization was first proposed by Ball et al. [BCM+19].

While the binary circuit representation of arithmetic operations over $\mathbb{Z}_n$ produces high fan-in gates for larger $n \in \mathbb{Z}$, arithmetic circuits allow us to utilize residue number representations, reducing gate size expansion even for large n very effectively: By generalizing labels from sequences of $\mathbb{Z}_2$ to sequences of $\mathbb{Z}_{P_k}$, we are required to not only define l^0 and l^1, but l^a for all $a \in \mathbb{Z}_{P_k}$. We observe that the size of each input- and output label l_a is no longer invariant, but is affected by the size of P_k: Let λ be the global security parameter, e.g. usually 128 for AES-128, then we observe that $l^a \in \mathbb{Z}_m^{\lambda/\log_2 P_k+1}$, where the one additional entry stems from generalizing the point-and-permute technique for arithmetic circuits, i.e. using a color digit in $\mathbb{Z}_{P_k}$ instead of a binary color digit for labeling. This results in logarithmic asymptotic label size expansion.

The additive operator $\oplus$ over $\mathbb{Z}_2$ is analogous to $+$ over arbitrary residue rings in $\mathbb{Z}$, allowing us to follow the Free-XOR construction by Kolesnikov et al. [KS08]: Free addition is enabled by choosing

$$l^a = l^0 + aR_{P_k} \tag{2.12}$$

where the circuit-wide constant R_{P_k} is now an element of $\mathbb{Z}_{P_k}$, analogous to $R \in \mathbb{Z}_2$ in binary circuits, as this simplifies addition to:

$$l_x^a + l_y^b \mod P_k = (l_x^0 + l_y^0) + (a+b)R_{P_k} \mod P_k \tag{2.13}$$

Ball et al. showed that this construction allows for free multiplication by a constant as well [BCM+19]:

$$c \cdot l^a \mod P_k = c \cdot l_x^0 + c \cdot aR_{P_k} \mod P_k \tag{2.14}$$

Unfortunately, other functions are not free but must be constructed using the approach outlined earlier in Section 2.3.1, i.e. one must encrypt function results in a truth table using input labels as encryption keys. For any arbitrary unary function $\phi : \mathbb{Z}_m \mapsto \mathbb{Z}_n$ we can construct a modular *projection* gate that garbles the result for each possible garbled input value $l^a : a \in \mathbb{Z}_m$ using $\text{Enc}_{l^a}(l^{\phi(a)})$ to create the garbled output. Here we can see why Sander et al. chose to utilize Ball et al.'s approach of leveraging residue representations to express a, as otherwise the truth table becomes unpractical in size for larger m.

Several non-linear components of NNs can be built from the *sign* function. For example, activation layers commonly utilize $ReLU(x) = x \cdot sign(x)$. Luckily, the garbled *sign* function can be approximated efficiently in a manner first described by Ball et al. [BCM+19]: We first look at the standard procedure on how to map an integer in residue representation $([x]_1, ..., [x]_k)$ back to $\mathbb{Z}$ (cf. Section 2.2):

$$x = \sum_{i=1}^{k} \alpha_i \cdot [x]_i \mod P_k. \tag{2.15}$$

It follows that some factor $q \in \mathbb{Z}$ must exist, such that

$$x = qP_k + \sum_{i=1}^{k} \alpha_i [x]_i \iff \frac{x}{P_k} = q + \frac{\sum_{i=1}^{k} \alpha_i [x]_i}{P_k}. \tag{2.16}$$

As $q \in \mathbb{Z}$ and $P_k \in \mathbb{N}$, the fractional part of $\frac{x}{P_k}$ and $\frac{\sum_{i=1}^{k} \alpha_i [x]_i}{P_k}$ must be equal. Since x was derived from a residue representation that is upper-bounded by P_k, $x < P_k$ and therefore $q = 0$. Let us now set an equivalent definition of the *sign* operation used in DASH (as will be detailed later in Section 3.2) for the case that we are working on discretized floating point numbers in $\mathbb{Z}_{P_k}$:

$$sign(x) := \begin{cases} 1, & \text{if } x \geq \frac{P_k}{2} \iff \text{if } \frac{x}{P_k} \geq \frac{1}{2}, \\ 0, & \text{otherwise.} \end{cases} \tag{2.17}$$

From the above we conclude:

$$\frac{x}{P_k} \geq \frac{1}{2} \iff \text{fractional part of } \frac{\sum_{i=1}^{k} \alpha_i[x]_i}{P_k} \geq \frac{1}{2} \tag{2.18}$$

We can now calculate $sign$ by computing $\alpha_i[x]_i/P_k$ for all residues. It is possible approximate the result for performance reasons. This approximation relies on choice of *discretization level* M and radices $m_1, ..m_t$ (s.t. $\prod_{i=1}^{t} m_i = M$) to construct mixed-radix approximations of the input residue components of x. Consequently we do not compare with $\frac{1}{2}$ but with $\frac{M}{2}$ when evaluating Equation 2.18. We use this mixed-radix approximation, as it allows for an arithmetic generalization of the Free-XOR technique mentioned earlier [KS08] as presented by Malkin et al. [MPS15], i.e. free integer addition. Furthermore, for large M, it is significantly more runtime and memory efficient to represent that integer value using a computationally attained mix of larger radices [BCM+19] This follows a similar motivation as to why we use residue representations for GC labels. So in conclusion, we perform three steps:

I: Compute $\alpha_i[x]_i/P_k$ for all $i \leq k$.

II: Compute the sum of all $\alpha_i[x]_i/P_k$.

III: Compare the result to $\frac{M}{2}$.

These individual steps will be important later when comparing our scaling solution to Sander et al.'s [SBBE23b] in terms of ciphertexts required per garbled computation in Chapter 6.

3 Secure Neural Network Inference in DASH

In this thesis we work on improving the DASH framework, an already existing solution to secure machine learning inference provided by Sander et al. in 2023 [SBBE23b]. Before outlining our approach, we must therefore give an overview of the current state of DASH.

3.1 Overview

Similar to previous solutions (cf. Section 1.2), DASH offers secure NN inference by utilizing arithmetic GCs. However, certain distinctive features of DASH enable it to outperform such previous solutions both in terms of quantitative performance measures and feature set richness. Among them, most central are:

1. *Arithmetic Garbled Circuits*. As described earlier in Section 2.3.3, abstractions to FreeXOR in binary GCs can be made, enabling efficient garbling of circuits composed of arithmetic gates. DASH utilizes such advancements, operating on arithmetic circuits which evaluate large integers in residue representation.

2. *CUDA / GPU support*. DASH supports efficient CPU- and GPU-based parallelization during circuit evaluation. When looking at the prevelance of massive parallelization in non-secure machine learning today, it is logical to pursue this development in secure machine learning as well.

3. *Single Round of Communication*. While other solutions struggle with infeasible runtimes due to large communication overheads for increasingly deep NN models, DASH can handle such models easily due to only needing one round of communication per inference regardless of model depth.

4. *ONNX-based interoperability*. DASH supports the widely used ONNX format to import pre-trained NN models. This allows for greater interoperability with already existing (non-secure) solutions.

5. *TEE support*. DASH utilizes Trusted Execution Environments (TEE), enabling a two-level hierarchy of security properties between non-critical operations that can be offloaded to non-secure hardware and critical garbling operations that are securely executed within a TEE. This allows DASH to achieve security against the malicious attacker model without expanding the (arithmetic) GC protocol.

6. *Rescaling of inputs* Furthermore, DASH allows to deploy a scaling operation during quantization of floating-point inputs to integers (in CRT representation), which we will now discuss in detail.

3.2 Garbled Scaling by Two

While formally operating over (tensors of) $\mathbb{R}$ as outlined in Section 2.1.1, most NN implementations use floating-point representations during training and inference. As arithmetic GCs operate on integer rings, DASH opts for a different choice: All model information as well as the actual inference input are quantized to $\mathbb{Z}_{P_k}$ either by the means of simple rounding (after multiplying by a small circuit-wide quantization constant α, i.e. performing $round(x \cdot \alpha)$) or, more interestingly, by the means of first *scaling* and then rounding during quantization. The first approach is denoted as *SimpleQuant*, while second one is denoted as *ScaleQuant* by DASH's authors [SBBE23a].
The latter option allows for garbled inference on significantly larger NN models: Via *downsampling* of hidden layer feature vectors, inputs quantized to larger integer ring domains can be processed in a garbled manner, while the accuracy tradeoff remains negligable. This allows for overall better garbled inference results, as more complex network topologies outweigh this accuracy loss in practice.
Implementing a garbled scaling operation is non-trivial, as our arithmetic generalization of binary GCs operate on CRT representations, for which general division is impractical. DASH currently supports scaling by *scaling factor* $s = 2$, i.e. transforms residue representations of $x \in \mathbb{Z}$ to $y := \lfloor \frac{x}{s} \rfloor$ for $s = 2$.
Let $([y]_1, .., [y]_k)$ be the residue representation of y, analogously to x. In order to map from both positive and negative integers to the residue representation's value range $\mathbb{Z}_{P_k}$, the actual scaling step is preceded by a *shift up* operation and succeeded by a *shift down* operation as depicted in Figure 3.1.
During scaling, Sander et al. utilize a per-residue computation proposed by Jullien [Jul78] to determine $[y]_2, .., [y]_k$:

$$[y]_i = ([x]_i - ([x]_i \mod 2)) \cdot 2^{-1} \mod p_i. \tag{3.1}$$

The remaining component $[y]_1$ can then be deduced as follows:

$$[y]_1 = sign(x') := \begin{cases} 0, & \text{if } x' \text{ is negative in integer representation,} \\ 1, & \text{otherwise.} \end{cases}, \tag{3.2}$$

where $x' = (0, [y]_2, .., [y]_k)$ in residue representation. This final step is called *base exten-*

sion (BE), because we extend the CRT base of the RNS from $(3, .., p_k)$ to $(2, 3, .., p_k)$. The resulting circuit is visualized in Figure 3.2.
Note that this *sign*-based BE cannot be generalized for $s > 2$. This poses a problem as scaling by higher scaling factors must therefore be constructed by repeated application of $s = 2$ scaling layers in DASH, i.e. scaled quantization is limited to

$$ScaleQuant(x, l) = round(x \cdot 2^{-l}), \tag{3.3}$$

where l is the number of scaling layer repetitions. In DASH's current state, their limitations result in scaling layers presenting a runtime performance bottleneck, especially for more complex NN topologies. Furthermore, this limits the overall scaling factor to be a power of two.

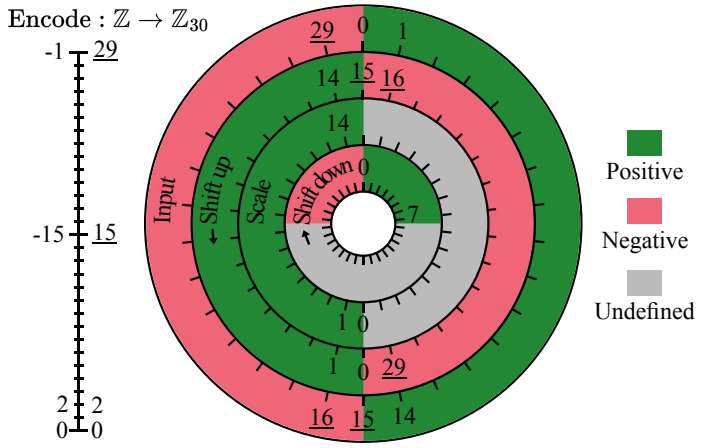

Figure 3.1: DASH's scaling of integers in residue representation for $P_k = 30$ and $s = 2$. Image taken from [SBBE23a].

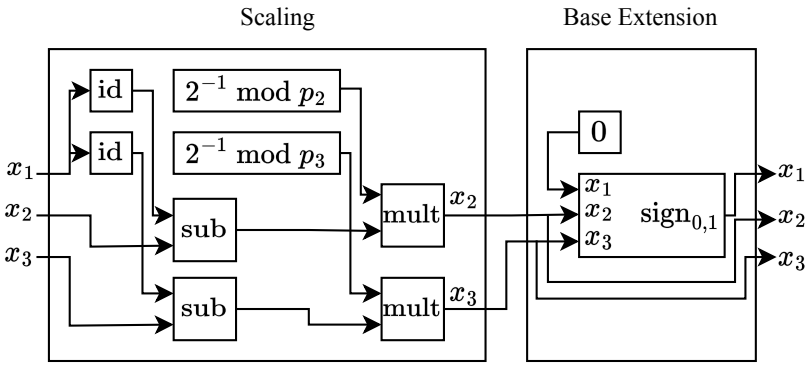

Figure 3.2: Arithmetic circuit used for scaled quantization with $s = 2$ in DASH. Image taken from [SBBE23a].

3.3 Security Discussion

Sander et al. [SBBE23a] define three central security goals for DASH:

1. *Input Privacy*. The input shall be inaccessible to all MPC parties except for the respective input provider, even against malicious and co-operating adversaries. This property follows from what Sandel et al. call the *garbling assumption* and *device assumption*: The former states that the evaluation of garbled protocols is passively secure. This assumption is reasonable, they argue, as it is based on well-established cryptographic assumptions. The latter states that within DASH's TEEs, i.e. during garbling, computation is secure.

2. *Integrity of the Inference*. It must be ensured that the garbled NN accurately simulates the underlying NN model and computes the correct result of the inference. This property directly follows from the GC protocol.

3. *Output Privacy*. The output of the NN inference must hold to the same security standards as the initial input. Following the garbling assumption and the device assumption, this property is assured as well.

As discussed earlier when defining secure machine learning, DASH does not guarantee *model privacy*. The authors argue that this security goal is generally hard to reach due to existing model extraction attacks on NNs. Furthermore, it is important to note that because DASH does not support techniques such as the cut-and-choose protocol [LP12], input privacy under the malicious attacker model is only achieved when deploying TEEs, which we do not support for our two contributions. This is why for our purposes, DASH provides security against the semi-honest model, as is the standard for the (arithmetic) GC protocol. The difference between these two adverserial models is that during computation the malicious (or active) adversary may deviate from the protocol, while in a semi-honest (or honest-but-curious) setting, the adversary tries to learn additional information while following the protocol.

4 Generalized Modulus-Based Base Extension

A per-layer runtime comparison for DASH conducted by Sander et al. [SBBE23a] yielded the insight that in deeper NN topologies, the scaling operation introduced earlier currently presents a runtime performance bottleneck. Our hypothesis is that this is caused by the current limitation of $s = 2$ and that a general solution for arbirary s would be more efficient compared to the current solution of stacking scaling layers with scaling parameter $s = 2$. This generalization is, however, non-trivial as we need to find a way to generalize the BE to residues with modulus greater than two. This will be our approach to the first research question proposed earlier in Chapter 1.

By applying Szabo and Tanaka's 1967 BE algorithm [ST67] to Sander et al.'s arithmetic GC-based secure machine learning solution DASH [SBBE23a], we generalize DASH's scaling-by-two approach to enable scaling by an arbitrary prime factor.

As pointed out in Chapter 3, Sander et al. leveraged Jullien's approach to residue number scaling [Jul78], which we introduced earlier for the special case of scaling by $p_0 = 2$ in Equation 3.1. For clarity, let $s =: p_e \in \{2, 3, ..p_k\}$ denote our prime scaling factor and $[x]_e$ the respective residue. Then, we can generalize Sander et al.'s approach to

$$[y]_i = ([x]_i - ([x]_e \mod p_i)) \cdot p_e^{-1} \mod p_i. \tag{4.1}$$

By conducting this generalization, we also generalize the problem of BE: While the former approach found in Equation 3.1 allows us to scale x in residue representation for all residues except $[x]_1$, i.e., except for the residue corresponding to the prime factor $p_1 = 2$, we now can only scale x in residue representation for $\forall [x]_i : i \neq e$.

Therefore, we will first describe how to tackle this generalized problem in Section 4.1 and then apply our approach to arithmetic GCs in DASH in Section 5.2.1. Finally, we describe how to deploy this new BE implementation in a novel garbled scaling layer which scales by p_e in Section 5.2.2.

Note that this generalization also allows us to scale by the product of multiple prime moduli $s = p_{e_1} \cdot p_{e_2} \cdot ...$ within one scaling layer. In this case, one scaling operation is followed by multiple generalized BE operations, i.e., by extending the output's base first by p_{e_1}, then by p_{e_2}, and so on. This possibility is discussed further in Section 5.2.2 as well.

4.1 The Base Extension Algorithm

In order to improve the readability of the following algorithms, let $[x]_m^{-1}$ denote $x^{-1} \mod p_m$. Furthermore, we make RNS to integer conversion less explicit and more readable: $x = ([x]_1, .., [x]_k)$ shall be equivalent to $x \longleftrightarrow ([x]_1, .., [x]_k)$.
For the implementation of modular scaling, we deployed an algorithm to perform BE proposed by Szabo and Tanaka in 1967 [ST67] that combines

1. modular division with remainder zero and
2. mixed-radix systems (MRS) (cf. Section 2.2.3).

As noted earlier, general division in RNS is a hard task. However, we will give a construction for the special case of division with remainder zero, i.e., division where the dividend x is an integer multiple of the divisor y. We assumed the division can not be simplified, i.e., x and y are coprime. For such x, y we can construct

$$\left(\frac{x}{y} \mod P_k\right) \cdot (y \mod P_k) = x \mod P_k. \tag{4.2}$$

From Equation 2.9 we know that this is equivalent ($\longleftrightarrow$) to the following in residue representation:

$$\left(\frac{[x]_i}{[y]_i} \mod p_i\right)_{i=1}^{k} \cdot ([y]_i \mod p_i)_{i=1}^{k} = ([x]_i \mod p_i)_{i=1}^{k}. \tag{4.3}$$

Since the multiplicative inverse of each $[y]_i$ is unique, it follows from Equation 4.3 that

$$\frac{[x]_i}{[y]_i} \mod p_i = [y]_i^{-1}[x]_i \mod p_i, \tag{4.4}$$

and therefore as a result for all RNS divisions with remainder zero

$$\frac{x}{y} \mod P_k = ([y]_i^{-1}[x]_i \mod p_i)_{i=1}^{k}. \tag{4.5}$$

This result implies free division with remainder zero in our arithmetic GC setting: During scaling, we only divide by (plaintext) RNS moduli p_i, making the generally hard task of finding a modular inverse cryptographically free. Hence, the computation is equivalent to performing one (free) multiplication by a constant.
Given $i = 1, ..., k$, let $B = (p_i)_{i \neq e}$ be our RNS base before BE and $B_E = (p_i)_i$ be our RNS base after BE. The general approach for an arbitrary prime BE given by Szabo and Tanaka [ST67] is to generate an MRS representation for x that is associated with our extended

RNS base B_E in order to find a_e and subsequently $[x]_e$. Note that when performing the BE, values of all $[x]_i$ for $i \neq e$ are known.

We treat $[x]_e$ as a variable and generate an associated MRS derived directly from Equation 2.11. This method finds a linear expression for a_e relative to our variable $[x]_e$. If we assume a_e to be the most significant digit in the MRS associated with B_E, i.e., assume $e = k$, we know that $a_e = 0$: Since x is within the CRT domain of the un-extended CRT base B, x can be put to mixed radix form using only the moduli of B as radices, implying $a_e = 0$. Since now (with the elimination of a_e as a variable) the aforementioned linear expression only contains one unknown variable, namely $[x]_e$, we can easily find a solution for $[x]_e$. This, however, limits our choice of e to $e = k$ and therefore s to $s = p_k$, which poses no problem as it merely demands re-ordering our extended CRT base B_E before and after BE, which will be discussed later in Section 5.2.1.

We will now illustrate this algorithm with an example. First, we provide an abstract description and a numerical example of Szabo and Tanaka's MRS generation algorithm. This first step is crucial as it offers a practical understanding of our algorithmic starting point. Then, in a second numerical example, we perform a BE to illustrate the procedure. A pseudocode implementation of the entire BE algorithm is then provided in Algorithm 1.

Example: MRS Generation

Let $B = (8, 5, 7, 3)$ be our choice of a coprime RNS base. We aim to find the associated MRS coefficients $a_1, ..a_4$ for a given residue representation $x = (3, 4, 2, 1)$. Note that the motivation behind generating this MRS is purely that the BE algorithm described afterward builds upon its general procedure: We are *not* interested in the resulting MRS itself. Recall Section 2.2.3 for notational details.

Generating an associated MRS representation consists of the iterative subtraction of a_i, followed by a remainder-zero division by p_i. Since $x - a_i$ is divisible by p_i by construction (a_i is the leftover *residue* when performing $\frac{x}{p_i}$), this division is equivalent to residue-wise multiplication by p_i's modular inverse (cf. Equation 4.1). This process was introduced earlier in the recursive formula presented in Equation 2.11.

Getting back to our example, we know from Equation 2.11 that $a_1 = [x]_1 = 3$, so we begin by subtracting a_1 from x, both in CRT representation:

$$x - a_1 = (3, 4, 2, 1) - (3, 3, 3, 0) = (0, 1, 6, 1). \tag{4.6}$$

In order to determine a_2, we then continue to multiply by the modular inverse of $p_1 = 8$

(respective a_2's modulus $p_2 = 5$) to construct the expression found in Equation 2.11:

$$\begin{aligned} a_2 &= [(x - a_1) \cdot p_1^{-1}]_{p_2} \\ &= [(0, 1, 6, 1) \cdot (0, 2, 1, 2)]_5 \\ &= [(0, 2, 6, 2)]_5 \\ &= 2. \end{aligned} \tag{4.7}$$

From now on, we iteratively repeat this process: First, a subtraction from our intermediary result:

$$(0, 2, 6, 2) - a_2 = (0, 2, 6, 2) - (2, 2, 2, 2) = (0, 0, 4, 0). \tag{4.8}$$

Followed by a multiplication by a modular inverse:

$$\begin{aligned} a_3 &= [((0, 2, 6, 2) - a_2) \cdot p_2^{-1}]_{p_3} \\ &= [(0, 0, 4, 0) \cdot (0, 0, 3, 2)]_7 \\ &= [(0, 0, 5, 0)]_7 \\ &= 5. \end{aligned} \tag{4.9}$$

Only a_4 is now left to be determined. We repeat the procedure once more:

$$(0, 0, 5, 0) - a_3 = (0, 0, 5, 0) - (5, 0, 5, 2) = (0, 0, 0, 1), \tag{4.10}$$

$$\begin{aligned} a_4 &= [((0, 0, 5, 0) - a_3) \cdot p_3^{-1}]_{p_4} \\ &= [(0, 0, 0, 1) \cdot (0, 0, 0, 1)]_3 \\ &= [(0, 0, 0, 1)]_3 \\ &= 1. \end{aligned} \tag{4.11}$$

The algorithm terminates and we know: The coefficients of x in the associated MRS of our RNS are $(a_1, a_2, a_3, a_4) = (3, 2, 5, 1)$. Notice that during the computation of any a_i, we never use the information of $[x]_j$ for any $j < i$, i.e., less significant RNS components of modular index j do not affect the computational outcome for more significant indices i. From now on, we will consequently not compute such $[x]_j$, as seen in line 5 of our BE's algorithmic description found later in Algorithm 1.

Example: Base Extension

But first, let us expand the example above to Szabo and Tanaka's BE algorithm and illustrate it with another example. Here, we're working with a smaller $B = (2, 3, 5)$, adding 7 as an extra modulus. Choosing the same x as before, we now have $(1, 1, 3)$ as its residue representation in this 3-modulus CRT base. We perform a BE adding 7 as a modulus, i.e., $B_E := (2, 3, 5, 7)$. In the extended base, $x = (1, 1, 3, [x]_7)$. We begin by executing the algorithm for generating the associated MRS coefficients (a_1, a_2, a_3, a_4) as we just described in Eqs. 4.6 to 4.10. This results in $a_1 = 1, a_2 = 0, a_3 = 2$ and, most importantly,

$$a_4 = [2^{-1}3^{-1}5^{-1}[x]_7 + 4]_7 = [30^{-1}[x]_7 + 4]_7 = [2^{-1}[x]_7 + 4]_7. \tag{4.12}$$

Because x was within the domain of the original CRT base B, that is, within the interval $[0, (2 \cdot 3 \cdot 5) - 1] = [0, 29]$, we know that it can be expressed as a mixed-radix number (a_1, a_2, a_3) in B's associated three-digit MRS. This implies that in the four-digit MRS associated with B_E, $a_4 = 0$. This allows us to find $[x]_7$:

$$\begin{aligned} a_4 =& [2^{-1}[x]_7 + 4]_7 = 0 \\ \iff& [x]_7 + [8]_7 = 0 \\ \iff& [x]_7 = [-8]_7 = 6 \end{aligned} \tag{4.13}$$

Finding the residue for our newly added modulus $p_4 = 7$ concludes Szabo and Tanaka's algorithm, which is, as they put it, essentially an "MRS conversion with an additional step" [ST67].

4.2 Non-Garbled Base Extension Pseudocode

Before implementing a garbled version of this algorithm in DASH, we designed a non-garbled version aiming to reduce expressions that are expensive to garble in arithmetic GCs, i.e., reduce the number of projection gates. An in-depth analysis of projection gate count and a comparison to the current state-of-the-art implementation in DASH can be found later in Chapter 6. The non-garbled algorithm is presented in Algorithm 1.

Algorithm 1 Algorithmic description of the non-garbled prototype implementation of Szabo and Tanaka's BE algorithm. $\mathbf{x} = (\mathbf{x}_1, .., \mathbf{x}_k)$ is the operation's input tensor. The algorithm returns the BE result in $\mathbf{x}_e$ and assumes $s = p_k$, i.e., that the last residue must be base-extended. The resulting re-ordering of B_E may imply that p_k is not its largest prime, which is we must introduce $p_{max} := \max(B_E)$.

procedure BASE EXTENSION
  $w \leftarrow \mathbf{x}$
  $ext \leftarrow 0$
  for $i \leftarrow 0$ **to** $k - 1$ **do**
    for $j \leftarrow 0$ **to** $k - i - 1$ **do**
      if $i = 0$ **then**
        $[w]_{i+j+1} \leftarrow [w]_{i+j+1} - ([x]_0 \mod p_{i+j+1})$
      else
        $[w]_{i+j+1} \leftarrow [w]_{i+j+1} - (ext \mod p_{i+j+1})$
      end if
      $[w]_{i+j+1} \leftarrow [w]_{i+j+1} \cdot \left(\left(\prod_{p_l \in B_E : l > i} p_i^{-1} \mod p_l\right) \mod p_{i+j+1}\right)$
    end for
    $ext = [w]_{i+1} \mod p_{max}$
  end for
  return $-ext \mod p_e \cdot \left(\left(\prod_{p \in B}[p]^{-1} \mod p_e\right)^{-1} \mod p_e\right)$
end procedure

This algorithm performs the iterative approach for determining MRS coefficients we just introduced: The central loop of Algorithm 1 finds an expression for the coefficient corresponding to our additional modulus, followed by the two operations performed on $[4]_7$ in our BE example: The expression is multiplied by the (modular) *inverse* of $\left(\prod_{p \in B}[p]^{-1} \mod p_e\right)$ (Equation 4.13, Line 2) and then multiplied by -1 (Equation 4.13, Line 3).

5 Implementation

Now that we have introduced the theoretical ideas behind our garbled modulus-based BE approach, we can go over to discuss implementation details of our solutions to both research questions presented in Chapter 1.
We propose a garbled max-pooling implementation for DASH to tackle the second research question. Pooling layers find widespread application in real-world NN topologies and are currently unsupported by DASH. Instead, DASH models are restricted to increasing stride parameters in convolutional layers to achieve downsampling, possibly implying a loss in accuracy or runtime performance.
As some implementation details are best understood by first describing our design of garbled max-pooling layers and then going over the more extensive subject of our scaling generalization, we will discuss our approach to the second research question and then cover the first.

5.1 Garbled Max-Pooling Layer Implementation

We will now describe in detail our approach, which was given a brief overview above, starting with the addition of a max-pooling operation for DASH. Discussing this topic first offers the opportunity to delve into the intricacies of layer implementation in DASH, which will form a basis for our other endeavors as well. Therefore, we will now showcase the implementation process of additional layers in DASH from a general perspective in Section 5.1.1, and then further explore newly added software components required for this specific layer in Section 5.1.2. We then conclude this Section on garbled max-pooling by detailing input size limitations in relation to the choice of CRT base in Section 5.1.3 and covering concurrent execution in Section 5.1.4.

Layer Design in DASH

Designing layers in DASH is based on drawing parallels between garbled- and non-garbled versions of components: Implementations of the *Layer* class interface serve as non-garbled versions of NN layers. These non-garbled layers are managed by the singleton *Circuit* class, essentially representing a non-garbled NN. This singleton *Circuit* is then used during initialization of the (also singleton) *GarbledCircuit* class, representing a gar-

bled NN: For each *Layer*, find the corresponding implementation of the *GarbledLayer* class interface, then add it to the *GarbledCircuit*.
Knowing this, we can construct an exemplary garbled NN containing a single garbled max-pooling layer showcasing the end-user convenience of this approach:

Listing 5.1: Garbled NN in DASH containing a garbled max-pooling layer. The pooling layer expects input size 16×16, channel count $C = 3$, and kernel size $K_X, K_Y = 2$. MRS base for approximated garbled *sign* was chosen as proposed by Ball et al. [BCM+19] for CRT base size $k = 3$.

```
const vector<crt_val_t> crt_base{2, 3, 5};
const vector<mrs_val_t> mrs_base{26, 6, 3, 2};

auto circuit = new Circuit{new MaxPool2d(16, 16, 3, 2, 2)};
auto gc = new GarbledCircuit(circuit, crt_base, mrs_base);
```

Given some $16 \times 16 \times 3$ input data, we can now proceed with the GC protocol. As outlined in Section 2.3, the two parties following the protocol now

1. garble the input data, then
2. evaluate the GC, and finally
3. decode the garbled output data.

These two parties can either communicate via OT or be (as is the case for this simple example) on the same host machine, referred to as the *one-server setting*. This translates to the contents of Listing 5.2:

Listing 5.2: Garbling of input data, evaluation, and decoding of garbled output of the GC constructed in Listing 5.1.

```
// 1. Garbler garbles inputs:
const auto g_inputs{gc->garble_inputs(inputs)};
// 2. Evaluator evaluates the GC:
const auto g_outputs{gc->cpu_evaluate(g_inputs)};
// 3. Either party decodes the garbled output:
const auto outputs{gc->decode_outputs(g_outputs)};
```

This same separation of garbling and then evaluating can be found internally in individual layers as well. The *GarbledLayer* interface requires the implementation of a member function for both garbling and evaluation. While the former is used to describe the arithmetic circuit components to be garbled, the second describes how these garbled components shall be evaluated during circuit evaluation. These two are algorithmically very similar.

Algorithm 2 Algorithmic description of the *Garble* function found in garbled max-pooling: *current_max* and *comp* are local auxiliary variables, *output_width*, *output_height*, C, K_X and K_Y are global constants, *max_gadgets* is a class member, and **x** is the operation's (three-dimensional) input label tensor. For better readability, this procedure abstracts from internal CRT style label representations.

```
1:  procedure GarbledMaxPool2d::Garble()
2:      for i ← 0 to output_width do
3:          for j ← 0 to output_height do
4:              for k ← 0 to C do
5:                  current_max ← x(i, j, k)
6:                  for l ← 0 to K_X do
7:                      for m ← 0 to K_Y do
8:                          comp ← x(i + l, j + m, k)
9:                          Append new MaxGadget to max_gadgets.
10:                         current_max ← max_gadgets.back().Garble(current_max, comp)
11:                     end for
12:                 end for
13:                 Copy current_max to pre-allocated output label at index i, j, k.
14:             end for
15:         end for
16:     end for
17: end procedure
```

Let us, therefore, first discuss the *Garble* function in detail and then showcase what differentiates it from *Evaluate*.

As seen in Algorithm 2, the approach can be broken down to a singular iteration over all output labels, for each of which a local two-dimensional kernel in each input channel is traversed. We then utilize a *MaxGadget*, allowing garbled computation of the maximum of two garbled (single-element) input labels. What precisely a *gadget* is in this context and how the *MaxGadget* operates will be discussed in detail in Section 5.1.2.

Now, how does the garbled pooling implementation differ during *evaluation* round? Our modular gadget-based design allows us to isolate large parts of the protocol's cryptographic details and not show up in layer implementations. The difference between garbling and evaluation therefore lies in how we interface with elements of *max_gadgets*: During evaluation (i.e. when calling *GarbledMaxPool2d::Evaluate*), we do not allocate new gadget instances, but rather reference the ones previously created during garbling (cf. Algorithm 2, Line 9) by passing *comp* and *cur_max* to *MaxGadget::Evaluate* where *MaxGadget::Garble* was called during garbling (Algorithm 2, Line 10).

When later discussing the inner workings of the *MaxGadget* in Section 5.1.2, a similar pattern will emerge: By combining DASH's various gadgets with some basic arithmetic op-

erations implemented as garbled gates, operations such as *Max* or *ReLU* can be modeled similarly to how max-pooling layers were brought to DASH.

Auxiliary Gadgets

Gadgets fill a design gap between garbled network layers and garbled gates. While the former implements modular end-to-end garbled neuron transformation functionality, the latter implements single-input garbled operations. We thus require a third category of *gadgets* that operate on batches of labels while not being restricted to the end-user interface garbled layers provide. The first such gadget in DASH was the *SignGadget* provided by Sander et al. [SBBE23b], implementing the *sign* operation defined earlier in Equation 3.2.

For our max-pooling implementation discussed in Section 5.1.1, the *max* operation is crucial. Therefore, we first implemented a *ReLUGadget* and built a *MaxGadget* from there. To do so, we utilized the following equation provided by Ball et al. [BCM+19]:

$$max(x, y) = x + ReLU(y - x) \tag{5.1}$$

where $ReLU(x) = x \cdot sign(x)$ (cf. Section 2.3.3). This allows us to reduce the *max* operation to computing *ReLU*, while computing *ReLU* can be reduced to applying Sander et al.'s *SignGadget*. Algorithmically, we arrive at Algorithms 3 and 4 defining the *Garble* functionality of our new *ReLUGadget* and *MaxGadget* respectively. The corresponding *Evaluate* functions are not given in detail here, as their marginal difference follows the same pattern mentioned earlier for the difference between *GarbledMaxPooling2d::Garble* and *GarbledMaxPooling2d::Evaluate* in Section 5.1.1.

Algorithm 3 Algorithmic description of the *Garble* function of DASH's *ReLUGadget*. Again, **x** is the input label tensor; *out_sign* and *out_mult* are auxiliary label tensors of equivalent dimensionality as the input. The singular *SignGadget* is assigned to *sign_gadget* before execution, and the |**x**| *MixedModulusHalfGates* are assigned to *mm_hgs* during execution.

```
1: procedure ReluGadget::Garble
2:     out_sign ← sign_gadget.Garble(x)
3:     for i ← 0 to |x| do
4:         Append new MixedModHalfGate to mm_hgs
5:         out_mult[i] ← mm_hgs.back().Garble(x[i], output_sign[i])
6:         Copy out_mult[i] to pre-allocated output label at index i.
7:     end for
8: end procedure
```

Algorithm 4 Algorithmic description of the *Garble* function of DASH's *MaxGadget*. This time, we have two single-entry input label tensors $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$ (corresponding to x and y in Equation 5.1). The *ReluGadget* is assigned to *relu_gadget* before execution. Addition ($+$) and subtraction ($-$) implies DASH's implementation of free garbled addition (cf. Equation 2.12).

1: **procedure** MAXGADGET::GARBLE
2: *out_subtraction* $\leftarrow \mathbf{x}^{(2)} - \mathbf{x}^{(1)}$
3: *out_relu* $\leftarrow$ *relu_gadget.Garble*(*out_subtraction*)
4: *out_addition* $\leftarrow \mathbf{x}^{(1)} +$ *out_relu*
5: Copy *out_addition* to pre-allocated output label of size 1.
6: **end procedure**

Correct Choice of CRT Base

Since we utilize modular congruence classes over $\mathbb{Z}_{P_k}$ to model our garbled values, the collision-free domain is limited by the choice of P_k. The first obvious limitation is, as reasoned in Section 2.2, that all compared values must be within the boundaries of our RNS, i.e.,

$$\left\lfloor -\frac{P_k}{2} \right\rfloor \leq x, y \leq \left\lfloor \frac{P_k}{2} - 1 \right\rfloor . \tag{5.2}$$

It does not suffice, however, to simply look at the domain limits of the discretized NN input vectors x, y, as in some cases, preliminary results during layer evaluation may be larger in size than the input or the computation's result. Note that this is the case for our garbled pooling implementation: In Equation 5.1 we conduct a per-residue subtraction of two input values. This means that in order for this preliminary result to not cause modular collision, we require not only the above but furthermore that

$$\left\lfloor -\frac{P_k}{2} \right\rfloor \leq x - y \leq \left\lfloor \frac{P_k}{2} - 1 \right\rfloor . \tag{5.3}$$

As x, y can hold arbitrary residue values within their residual bounds, Eqs. 5.2 and 5.3 imply

$$\left\lfloor -\frac{P_k}{4} \right\rfloor \leq x, y \leq \left\lfloor \frac{P_k}{4} - 1 \right\rfloor . \tag{5.4}$$

The same argument applies to the addition in Equation 5.1, leading to the same conclusion of choosing our CRT base according to the upper and lower bounds presented in Equation 5.4.

Parallel Evaluation

Our implementation supports parallel execution during evaluation. We deployed a depth-three OpenMP [DM98] parallelization instruction over the pooling layer's central for loop (cf. i, j, k in Algorithm 2). As individual pooling kernels do not interfere with one another, this reproduces single-thread execution results consistently. We chose this approach over parallelizing within kernel evaluation (i.e., over l, k in Algorithm 2) as our gadget-based solution would make eliminating race conditions between concurrent $\max$ evaluations challenging.

5.2 Garbled Scaling by Arbitrary Scaling Factors

Building upon the above implementation details of DASH and the theoretic groundwork of Chapter 4, we are ready to describe our solution to the first research question.

Garbled Base Extension

As mentioned earlier, we aim to minimize the amount of projection gates used in our implementations. Projection gates are required for all operations that are not cryptographically free, that is, all operations except modular additions (which include modular subtraction) and multiplications by a constant.
Fortunately, Szabo and Tanaka's BE algorithm largely relies upon these two free operations. The only problem induced by their iterative approach is that it requires several *modular base change* projection operations computing

$$x \mod p_i \mapsto (x \mod p_i) \mod p_j \tag{5.5}$$

for some $i, j \leq k$. As each label in an arithmetic GC corresponds to one CRT modulus, the central loop of Szabo and Tanaka's approach necessarily implies the usage of modular base change projections: In each iteration, an intermediate result in the calculation of ext is derived from a CRT component of w, namely w_{i+1} (Algorithm 1, Line 13). In order to prevent information loss in this process, ext must be a residue of equivalent or larger modulus than each component of w. To still minimize the number of ciphertexts required during projections, we chose the modulus of ext to be $\max(B_E)$, i.e., the maximum modulus of all elements of w.
In addition to projections required when writing intermediate results to ext, we must also perform projections when writing *from* that label to labels of differing moduli:

1. During the i-th iteration of Szabo and Tanaka's BE algorithm, all CRT entries of w with an index larger than i are updated (cf. Algorithm 1, Lines 7 and 9). Here, the

current value of ext is subtracted from labels of varying moduli, making modular base change operations on ext necessary.

2. After performing the algorithm's final arithmetic operations (cf. Equation 4.13), the final result must be projected to be of modulus p_e, i.e., the modulus for which we perform the BE.

While the underlying *SignGadget* developed by Sander et al. [SBBE23b] deploys projection gates to perform non-free garbled computations (cf. comparison of ciphertext counts later in Chapter 6), our previously discussed implementation of garbled max-pooling did not require the manual management of such gates. All computation was based on a SIMD approach combining various *gadgets*, which differs from this more complex contribution of garbled BE: All projection gates are allocated, garbled, and evaluated per input.
Similarly to the parallelization of our first contribution, we parallelized the online phase (i.e., circuit evaluation) of our garbled BE implementation via OpenMP [DM98]. While the iterative steps in Szabo and Tanaka's approach build upon the previous steps and can thus not be computed concurrently, concurrency over the input domain poses no problem. A runtime analysis of this approach can be found later in Chapter 6.
The entire BE computation is wrapped in a new gadget for DASH, the *BaseExtensionGadget*. During the instantiation of this gadget, three pre-computations are performed before the gadget is garbled or evaluated:

1. At the end of each algorithmic iteration, the intermediate result of ext is multiplied by the product of several modular inverses of CRT base entries (cf. Algorithm 1, Line 1). As all CRT entries are public and not stored in garbled labels but in cleartext (which is why multiplying ext with this product of modular inverses is cryptographically free in the first place), the pre-computation of these products can be conducted before entering the garbling phase.

2. Analogously, the aforementioned $\prod_{p \in B}[p]^{-1} \mod p_e$, i.e., the product of inverses of the total CRT base, utilized in the final step of Szabo and Tanaka's algorithm (cf. Equation 4.13), is pre-computed during instantiation.

3. The order of CRT moduli in B_E is reversed. This is done to integrate the algorithm's requirement of the extra modulus to be at the very end of B_E. At the same time, DASH's scaling operation always assumes the first entry of B_E to be the scaling factor s and therefore to be the extra modulus p_e, requiring a reversal.

The *BaseExtensionGadget* is deployed during rescaling where Sander et al. [SBBE23b] previously deployed their *SignGadget*. This allows for garbled scaling by an arbitrary (prime) scaling factor $s = p_e$, which we will now discuss in further detail.

Integration in DASH's *RescaleLayer*

As outlined earlier in Chapter 3, DASH currently supports scaling by $s = 2$. A dedicated *RescaleLayer* performs this operation by first *shifting up* the signed integer representations of quantized neuron inputs x of value range $\left[-\frac{P_k}{2}, \frac{P_k}{2}\right]$ to the corresponding unsigned value range $[0, P_k]$. This is achieved via constant addition of $\frac{P_k}{2}$. After then applying the scaling operation (cf. Equation 3.1) including Sander et al.'s $sign$-based BE, the result is *shifted down* back to a new signed domain: Because we scaled down by the factor $s = 2$, we cannot simply subtract the previously added $\frac{P_k}{2}$, but must subtract $\frac{P_k}{2s} = \frac{P_k}{4}$ instead. Again, cf. Figure 3.1 for a visual depiction of this procedure.
Due to the modular design of our *BaseExtensionGadget*, the procedure can easily be generalized as follows for any $s \in 2, .., p_k$ by deploying our new modulus-based BE instead:

1. Shift up x by $\frac{P_k}{2}$.
2. Let $[x]_e$ be the residue corresponding to modulus s in the CRT representation of x. For all residue components $[x]_i \neq [x]_e$, perform Equation 4.1.
3. Apply the new *BaseExtensionGadget* in order to determine the extra modulus $[x]_e$.
4. Shift down by $\frac{P_k}{2s}$

By utilizing the generalized scaling formula found in Equation 4.1 proposed by Jullien [Jul78], replacing Sander et al.'s *SignGadget* with our new *BaseExtensionGadget* and adjusting the shifting down factor according to s, garbled scaling by an arbitrary prime factor finally becomes possible.
While not yet implemented in our solution, scaling by a *product of* several prime factors is also supported by this design: Let $s = p_{e_1} \cdot p_{e_2} \cdot ...$ be our scaling factor. Then the following slight variation of the above would scale x by s:

1. Shift up x by $\frac{P_k}{2}$.
2. For each $[x]_i$ with corresponding to modulus p_i that is *not* a factor in $s = p_{s_1} \cdot p_{s_2} \cdot ...$, perform Equation 4.1.
3. Apply a separate *BaseExtensionGadget* for each $s_1, s_2, ...$ in order to determine the extra moduli.
4. Shift down by $\frac{P_k}{2s}$

This result enables scaling not just by arbitrary *prime* scaling factors, but by any $s \leq P_k$.

6 Evaluation

In this chapter, we will analyze our approach. We will first conduct an experimental evaluation of our novel BE implementation based on Szabo and Tanaka's MRS-based solution [ST67]. We will not investigate the accuracy of this implementation, as Szabo and Tanaka's algorithm does not approximate and hence does not affect this metric. This experimental analysis is then followed by an analysis of ciphertext counts required when garbling in our solution, comparing it to Sander et al.'s *sign*-based approach.

6.1 Experiments

This section will evaluate the runtime performance of our novel modulus-based BE implementation. It will be challenged by the current state-of-the-art provided by Sander et al. and their *sign*-based BE implementation for DASH. All measurements were conducted in the one-server setting on a single Intel Xeon Gold 5415+ CPU with base clock speed of 2.90 GHz. As our new implementation does not currently support CUDA, we did not perform any experiments on GPUs.
We used P_8 (i.e., $k = 8$) as the CRT representations' primal modulus for all benchmarks. All input data consists of randomly generated integer values sampled from a uniform integer distribution of range $[0, 255]$. We used the Mersenne Twister algorithm *mt19937* [MN98] as our pseudo-random number generation engine. Each measurement was repeated ten times.
We only measure circuit evaluation time, i.e., the online phase of the GC protocol. This metric is the primary (runtime) performance metric pursued in DASH's design, as the circuit's usually more expensive garbling happens prior to any communication and can be seen as a pre-computation in a secure machine learning as a service setting.
Since Sander et al.'s [SBBE23a] BE implementation is restricted to $s = 2$, it was logical to compare how our implementation fares in that scenario first. This is the best-case scenario for their implementation, as higher scaling factors would have to be translated to multiple successive *RescaleLayer* evaluations, which we will cover in the following experiment. Even in the $s = 2$ scenario, evaluation runtimes remain very similar between their and our solution for various input sizes N and for various thread counts, as depicted in Figure 6.1. In each case, two NNs containing a single *RescaleLayer* were evaluated individually, one of which utilizing their implementation and one utilizing ours.

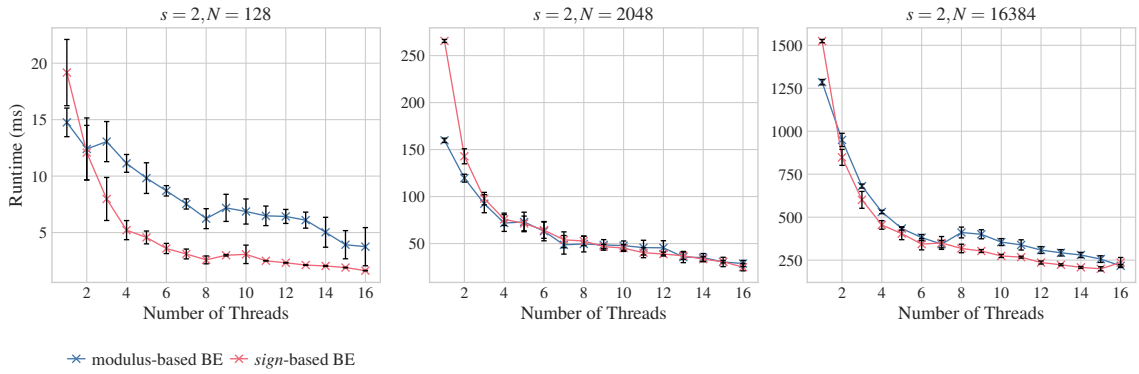

Figure 6.1: Runtime comparison of scaling by $s = 2$ for $N = 128, N = 2048$, and $N = 16384$ in 1 to 16 concurrent threads.

When looking at scenarios with larger s, our solution outperforms Sander et al.'s state-of-the-art solution by construction: As the only option with their *sign*-based BE is to concatenate additional $s = 2$ scaling layers, the total runtime expands logarithmically for larger s, i.e., one rescale layer for $s = 2$, two rescale layers for $s = 4$, three rescale layers for $s = 8$ etc. Our implementation does not increase in runtime for larger s: By reordering our CRT base to have p_e as the first modulus, we automatically scale by that prime number. Overall, we measued runtimes for $s = 2, 4, 8, 16$ utilizing the classical *sign*-based BE and $s = 3, 5, 7, 11, 13, 17, 19$ utilizing our approach.

As we restricted our measurements for the most realistic case of 16-thread parallel execution for this second experiment, runtime comparison for input sizes smaller than $N = 2048$ proved too high in variance (%RSD > 300 for $N = 128$). Therefore, only the empirically conclusive scenarios of $N = 2048$ and $N = 16384$ are depicted in Figure 6.2.

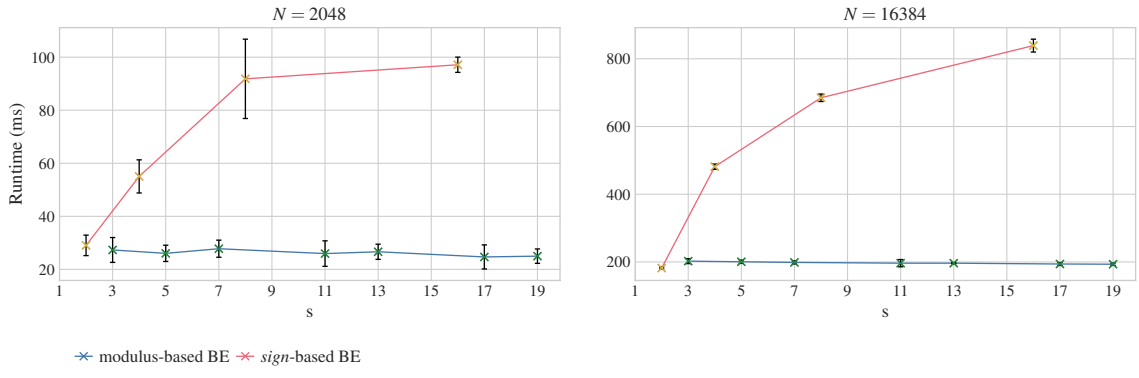

Figure 6.2: Runtime comparison of scaling with increasing s for $N = 2048$ and $N = 16384$, executed in 16 concurrent threads.

6.2 Space Complexity

In this section, we will compare the number of ciphertexts required to garble our modulus-based BE solution with the previously proposed $sign$-based approach. Sander et al.'s $sign$-based solution utilizes MRS to efficiently garble the approximated garbled $sign$ operation detailed earlier in Section 2.3.3. This use of MRS introduces a new parameter set $\prod_{i=1}^{t} m_i = M$, on which the degree of approximation, the ciphertext count and the resulting (garbling) runtime performance of their implementation depends. We choose optimized values proposed by Ball et al. [BCM+19] for M and all m_i such that perfect accuracy is achieved for various CRT base sizes k. They are depicted in Table 6.1.

k	$\prod_{i=1}^{t} m_i$	M
3	2^5	32
4	$26 \cdot 3$	78
5	$54 \cdot 4 \cdot 3$	648
6	$60 \cdot 5^3$	7500
7	$86 \cdot 7 \cdot 6^2 \cdot 5$	108360
8	$92 \cdot 7 \cdot 6 \cdot 5^3 \cdot 4$	1932000

Table 6.1: Optimal MRS parameters for exact sign garbled sign computation proposed by Ball et al. [BCM+19]

According to Sander et al., their approximated garbled $sign$ operation requires a total of

$$\underbrace{t\sum_{i=1}^{k} p_i}_{\text{Section 2.3.3, Step I}} \underbrace{+ 2k\sum_{j=2}^{t} m_j + 2(k-1) +}_{\text{Section 2.3.3, Step II}} \underbrace{m_1}_{\text{Section 2.3.3, Step III}} \tag{6.1}$$

ciphertexts per garbled input. Prior to this operation, the input must undergo a base change projection to modulus 2, resulting in another $2(k-1)$ ciphertexts per input. In comparison, our modulus-based solution requires

$$\underbrace{p_1 k}_{\text{Alg. 1, Line 7}} \underbrace{+ \sum_{i=1}^{k-1} p_k(k-i) +}_{\text{Alg. 1, Line 9}} \underbrace{\sum_{j=0}^{k-1} p_{j+1}}_{\text{Alg. 1, Line 13}} + \underbrace{p_k}_{\text{Alg. 1, Line 15}} \tag{6.2}$$

ciphertexts for one garbled input. Both solutions scale linearly for increasing input counts, which is why we restrict this analysis to $N = 1$. Our solution outperforms the old approach for all k for which Ball et al. provided optimized MRS parameters, namely $3 \leq k \leq 11$, cf. Figure 6.3

During garbled scaling by any s, all steps other than the BE at the end require no addi-

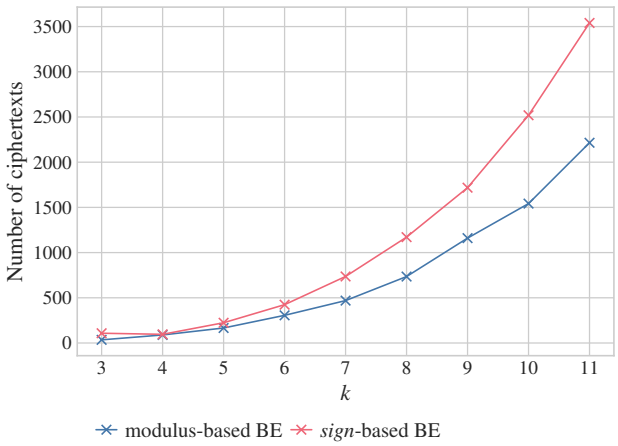

Figure 6.3: Ciphertexts required for garbled BE on one input ($N = 1$) in DASH using the two different approaches for various CRT base sizes k. t and m_i were selected according to Table 6.1.

tional ciphertexts, as they only contain cryptographically free additions and multiplications by a constant. Since scaling by larger s with the classical *sign*-based approach results in a logarithmically growing amount of *RescaleLayers*, the ciphertext count expands accordingly. At the same time, the ciphertexts required by our solution remain constant, as visualized in Figure 6.4.

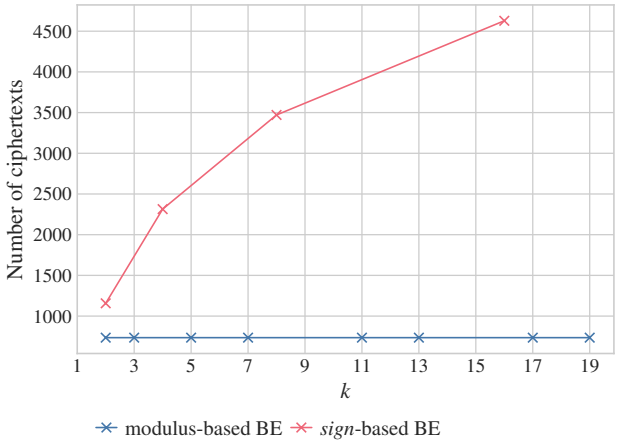

Figure 6.4: Ciphertexts required for garbled scaling on one input ($N = 1$) in DASH using the two different approaches for various scaling factors $s \in [2, 19]$. $k = 8$ for all calculations, t and m_i were selected according to Table 6.1.

7 Conclusions

In this final chapter, we present a concluding summary of our approach and discuss the extent to which the research questions presented in Chapter 1 were answered. Furthermore, we provide a detailed outlook on how our work can be continued and expanded upon in the future.

7.1 Summary

To tackle the first research question, we developed a replacement for Sander et al.'s [SBBE23a] *sign*-based BE that supports not only scaling by $s = 2$ but by any prime number (limited by choice of CRT base size k, i.e, $s \leq p_k$). This enables deploying a singular *RescaleLayer* in DASH where previously multiple layers, each scaling by $s = 2$, where utilized.
Our new approach applies an MRS-based algorithm for BE presented by Szabo and Tanaka in 1967 [ST67] in the context of arithmetic GCs. Even though their algorithm is relatively complex, we leveraged the fact that addition and multiplication by cleartext constants are free in DASH's optimized arithmetic GCs, allowing us to implement the algorithm efficiently.
Therefore, our solution requires fewer ciphertexts per garbled input (cf. Figures 3.1 and 3.2), implying faster garbling times and improved memory efficiency. In the classical $s = 2$ scenario, evaluation runtimes between our solution and the old *sign*-based solution are comparable and scale evenly for larger thread counts (cf. Figure 6.1). For $s > 2$, our solution outperforms the old implementation very clearly: As our modulus-based BE does not increase in evaluation runtime for larger s, scaling times remain constant. In contrast, the old approach by Sander et al. requires a logarithmically increasing number of *RescaleLayers*, which results in longer scaling times, as shown in Figure 6.2.
Regarding the second research question, we found a way to implement a max-pooling layer that directly builds upon Sander et al.'s approximated garbled *sign* implementation without requiring further projection gates and, therefore, no additional ciphertexts during garbling. We achieved this by expressing the *ReLU* function using Sander et al.'s *SignGadget* and several arithmetic operations that are cryptographically free in DASH's optimized arithmetic GCs, and then reducing the max-pooling's *max* operation to computing *ReLU*, as proposed earlier by Sander et al. [SBBE23b].

7.2 Future Work

Regarding the improved scaling operation, the next logical step is to support scaling by a product of multiple prime moduli. The theoretical foundation for this approach is detailed in Section 5.2.2. Enabling scaling by a product of prime moduli would give us more options in selecting the scaling factor s and would make the choice of the CRT base size k less dependent on specific scaling factors used across the NN.
Secondly, our new implementation must be extensively tested not just in the controlled and isolated environments used in Chapter 6, but also in realistic application scenarios that more closely mirror real-world conditions. Our starting point for generalized scaling was model benchmarking experiments conducted by Sander et al. (cf. Figure 1.1). It would be interesting to investigate to what extent *RescaleLayers* still present a performance bottleneck now that our optimizations can reduce their count substantially. Furthermore, the reduced ciphertext counts during the garbling process imply less overall memory consumption, as discussed earlier. This may, in turn, allow for the garbling of deeper and more complex NN topologies within the DASH framework.
Currently, only the online phase of our two contributions to DASH offers CPU parallelization. While the offline phase of the GC protocol may be less interesting than the online phase in terms of runtime performance in most applications, we still think optimizing this metric during the offline phase using OpenMP [DM98] would be worth pursuing.
We did not conduct any experimental evaluation of our new max-pooling operation for DASH. Investigating the runtime efficiency and inference accuracy implications of this new feature could lead to interesting insights. By deploying this new garbled layer and leveraging the practical possibilities created by faster scaling, exciting new opportunities emerge regarding the deployment of deeper garbled CNNs.
Lastly, GPU support via CUDA is a central feature of DASH. It will be essential to ensure that our contributions become compatible with CUDA, especially since the scaling performance bottleneck becomes even more pronounced when offloading circuit evaluation to a GPU, cf. Figure 1.1.

References

[BCM+19] Marshall Ball, Brent Carmer, Tal Malkin, Mike Rosulek, and Nichole Schimanski. Garbled neural networks are practical. *Cryptology ePrint Archive*, 2019.

[BGKP21] Julius Berner, Philipp Grohs, Gitta Kutyniok, and Philipp Petersen. The modern mathematics of deep learning. *arXiv preprint arXiv:2105.04026*, 2021.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 503–513, 1990.

[BMR16] Marshall Ball, Tal Malkin, and Mike Rosulek. Garbling gadgets for boolean and arithmetic circuits. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 565–577, 2016.

[CBL+18] Edward Chou, Josh Beal, Daniel Levy, Serena Yeung, Albert Haque, and Li Fei-Fei. Faster cryptonets: Leveraging sparsity for real-world encrypted inference. *arXiv preprint arXiv:1811.09953*, 2018.

[CJM20] Nicholas Carlini, Matthew Jagielski, and Ilya Mironov. Cryptanalytic extraction of neural network models. In *Annual international cryptology conference*, pages 189–218. Springer, 2020.

[Com16] European Commission. Art. 9 EU General Data Protection Regulation (GDPR). https://gdpr-info.eu/art-9-gdpr/, 2016.

[DM98] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[FGR16] Travis Floyd, Matthew Grieco, and Edna F Reid. Mining hospital data breach records: Cyber threats to us hospitals. In *2016 IEEE Conference on Intelligence and Security Informatics (ISI)*, pages 43–48. IEEE, 2016.

[Fuk80] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193–202, 1980.

[GBDL+16] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International conference on machine learning*, pages 201–210. PMLR, 2016.

[GW08] Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

[Jul78] Jullien. Residue number scaling and other operations using rom arrays. *IEEE Transactions on Computers*, 100(4):325–336, 1978.

[JVC18] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. {GAZELLE}: A low latency framework for secure neural network inference. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1651–1669, 2018.

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free xor gates and applications. In *Automata, Languages and Programming: 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II 35*, pages 486–498. Springer, 2008.

[LBD+89] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[LP12] Yehuda Lindell and Benny Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. *Journal of cryptology*, 25:680–722, 2012.

[MLS+20] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. Delphi: A cryptographic inference system for neural networks. In *Proceedings of the 2020 Workshop on Privacy-Preserving Machine Learning in Practice*, pages 27–30, 2020.

[MN98] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[MPS15] T Malkin, V Pastro, and A Shelat. The whole is greater than the sum of its parts: Linear garbling and applications. In *Workshop talk at Securing Computation Workshop in Berkley*, page 29, 2015.

[MZ17] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE symposium on security and privacy (SP)*, pages 19–38. IEEE, 2017.

[RM51] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

[RRK18] Bita Darvish Rouhani, M Sadegh Riazi, and Farinaz Koushanfar. Deepsecure: Scalable provably-secure deep learning. In *Proceedings of the 55th annual design automation conference*, pages 1–6, 2018.

[SBBE23a] Jonas Sander, Sebastian Berndt, Ida Bruhns, and Thomas Eisenbarth. Dash: Accelerating distributed private machine learning inference with arithmetic garbled circuits. *Unreleased Second Version*, 2023.

[SBBE23b] Jonas Sander, Sebastian Berndt, Ida Bruhns, and Thomas Eisenbarth. Dash: Accelerating distributed private machine learning inference with arithmetic garbled circuits. *arXiv preprint arXiv:2302.06361*, 2023.

[ST67] Nicholas S Szabo and Richard I Tanaka. Residue arithmetic and its applications to computer technology. *(No Title)*, 1967.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pages 162–167. IEEE, 1986.

[ZRE15] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole: Reducing data transfer in garbled circuits using half gates. In *Advances in Cryptology-EUROCRYPT 2015: 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II 34*, pages 220–250. Springer, 2015.

Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Masterstudiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellen- verzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe.

Hamburg, 13. August 2024