Convolutions in Overdrive: Maliciously Secure Convolutions for MPC

Marc Rivinius
University of Stuttgart
Stuttgart, Germany
marc.rivinius@sec.uni-stuttgart.de

Pascal Reisert
University of Stuttgart
Stuttgart, Germany
pascal.reisert@sec.uni-stuttgart.de

Sebastian Hasler
University of Stuttgart
Stuttgart, Germany
sebastian.hasler@sec.uni-stuttgart.de

Ralf Küsters
University of Stuttgart
Stuttgart, Germany
ralf.kuesters@sec.uni-stuttgart.de

ABSTRACT

Machine learning (ML) has seen a strong rise in popularity in recent years and has become an essential tool for research and industrial applications. Given the large amount of high quality data needed and the often sensitive nature of ML data, privacy-preserving collaborative ML is of increasing importance. In this paper, we introduce new actively secure multiparty computation (MPC) protocols which are specially optimized for privacy-preserving machine learning applications. We concentrate on the optimization of (tensor) convolutions which belong to the most commonly used components in ML architectures, especially in convolutional neural networks but also in recurrent neural networks or transformers, and therefore have a major impact on the overall performance. Our approach is based on a generalized form of structured randomness that speeds up convolutions in a fast online phase. The structured randomness is generated with homomorphic encryption using adapted and newly constructed packing methods for convolutions, which might be of independent interest. Overall our protocols extend the state-of-the-art Overdrive family of protocols (Keller et al., EUROCRYPT 2018). We implemented our protocols on-top of MP-SPDZ (Keller, CCS 2020) resulting in a full-featured implementation with support for faster convolutions. Our evaluation shows that our protocols outperform state-of-the-art actively secure MPC protocols on ML tasks like evaluating ResNet50 by a factor of 3 or more. Benchmarks for depthwise convolutions show order-of-magnitude speed-ups compared to existing approaches.

KEYWORDS

secure multiparty computation, convolutions, neural networks

1 INTRODUCTION

Machine learning (ML) and, in particular, deep learning are more and more growing in importance for academia and industry. The performance of an ML model, and hence, its application potential in real-world use cases, strongly depends on the amount and quality of available data. Since many companies are no longer able to generate the necessary data or models themselves, they have to rely on collaborations with competitors and other industry players.

Multi-Party Computation for ML.. Secure multiparty computation (MPC) addresses the challenges related to collaborative privacy-preserving machine learning. MPC allows several parties, e.g., companies, to compute functions on secret inputs and to reveal only the function result and no additional information, in particular no information on the sensitive inputs (beyond what can be inferred from the result).

Indeed, MPC has been shown to be a suitable tool for privacy-preserving ML in tasks like inference/evaluation and training (see, e.g., [16, 32] and Section 1.2). However, most of the MPC protocols that are specifically designed for ML provide security guarantees only in special setups, e.g., they require adversaries to follow the protocol rules (i.e., *passive security*) or limit the number of adversaries (i.e., honest majority). In a mutually distrustful setup, e.g., collaborations between industry competitors on highly sensitive data, these requirements can usually not be guaranteed. We therefore strive for a setup that guarantees to an honest party that their data remains private and that the result is correct even if all other parties are *actively* trying to corrupt the computation or gain sensitive information, e.g., by deviating from the previously agreed upon protocol. MPC protocols that provide this strong form of security are called *actively* or *maliciously* secure.

The currently best MPC protocols in this *dishonest majority* setting with *active security*, are SPDZ [18] or state-of-the-art improvements thereof [3, 30, 31]. The efficiency of *SPDZ-like* protocols relies on a two-phase approach consisting of an offline and an online phase. In the input-independent *offline phase*, different forms of structured random data, e.g., Beaver triples [4], are produced. Then, this random data is used in the *online phase* to speed up the computation on sensitive input data.

Direct Support for Matrix Multiplications and Convolutions. However, these protocols are not optimized for ML applications. In fact, *SPDZ-like* protocols were designed for arithmetic computations on finite field elements and hence each computation, e.g., a matrix multiplication or a convolution, has to be realized in a low-level way with just field addition and multiplication. For our ML operations and especially convolutions this approach usually leads to an unnecessary overhead in communication and computation.

For example, in a convolution of a 2d image $\boldsymbol{a}$ and a 2d filter $\boldsymbol{f}$ as presented in Fig. 1c, the filter runs over the whole image. In particular, each entry of the image (e.g., $\boldsymbol{a}[0,0]$) and each entry of

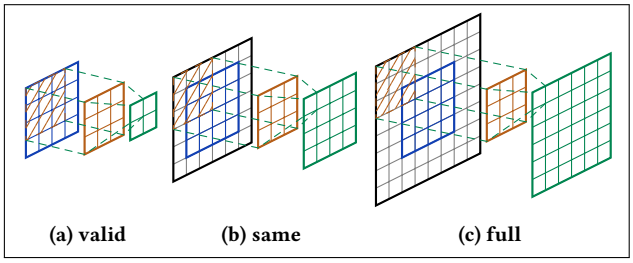

Figure 1: Example of Convolutions with Different Paddings. Here, we visualize how the first component of the output is computed for different padding modes when convolving a 4×4 image (blue) and a 3×3 filter (orange). The black pixels for same and full padding show where the image has to be padded with zero values. The first component of the result (green) is computed by multiplying the pixels of the padded image component-wise with the pixels of the filter at the overlapping positions (symbolized with the hatched copy of the filter) and then summing up these products.

the filter (e.g., $\boldsymbol{f}[0,0]$) is used in several multiplications. Classical protocols like [18] securely multiply these values with so-called Beaver multiplications. This requires both $\boldsymbol{a}[0,0]$ and $\boldsymbol{f}[0,0]$ to be masked with new masks for each multiplication they occur in and parties to send around messages between all MPC parties for each of these maskings. In Fig. 1c, overall 9 maskings of $\boldsymbol{a}[0,0]$ and 16 maskings of $\boldsymbol{f}[0,0]$ are created and sent.

The impact of this overhead (for each single convolution) on the efficiency of the overall ML algorithm is usually significant, given the large number of convolutions used in classical architectures, especially in *convolutional neural networks* (CNNs) [23, 35, 36] but also in *recurrent neural networks* (RNNs) [28, 38, 50] and *transformers* [2, 11, 20].[1]

The aforementioned efficiency issues for ML operations have been addressed by Mohassel and Zhang in [43], who replace the common Beaver triples with more complex structured random data, that is, specially adapted to matrix triples for matrix multiplication and convolution triples for convolutions. Matrix triples and convolution triples are perfectly adapted to the respective operations and no longer suffer from the overhead discussed before with the example of Fig. 1, i.e., they significantly lower communication and computational costs compared to standard Beaver triples.

While the protocol by Mohassel and Zhang is merely passively secure, their construction has been lifted to the actively secure setting by Chen et al. [14]. However, the focus of [14] is on matrix triples and matrix multiplication rather than convolutions. In particular, [14] does not use convolution triples but emulates convolutions with matrix multiplications. This is more efficient than the original approach based on Beaver triples but introduces an overhead linear in the filter size (filter height and width).

Actively Secure Convolutions with Dishonest Majorities. In this paper, we construct an actively secure MPC protocol which directly uses convolution triples and therefore natively supports convolutions. We show that our construction leads to a more efficient evaluation of convolutions than classical actively secure protocols like SPDZ and the matrix multiplication-based protocol [14], namely, the only actively secure protocol with direct support for an operation close to convolutions.

For our protocols we employ the successful two-phase protocol structure common in SPDZ-like protocols. In our case, in the offline phase what we call *convolution triples* are generated, which are then used in the online phase to very efficiently evaluate convolutions of sensitive inputs. In order to construct these convolutions triples in an actively secure offline phase, we employ a *homomorphic encryption scheme* (HE), similar to the currently fastest Beaver triple generation protocol Overdrive [3, 31]. Classically, HE-based offline phases gain most of their efficiency by amortization, i.e., they produce Beaver triples in large batches (with size usually in the range of 2^{12} to 2^{18}) to lower the per triple costs for encryption, zero-knowledge proofs, and other cryptographic tools. This approach is usually very efficient given the large number of Beaver triples needed in most applications, i.e., far more than the batch size. For example, in Fig. 1c we already need $16 \cdot 9 = 144$ Beaver triples for a single (rather small) convolution.

However, direct generalizations of these classical protocols are usually inefficient for ML applications. The reason is the large variety of convolutions of different sizes and the often small total number of convolutions of a specific size in many ML architectures, e.g., in ResNet [23] (cf. Section 7). A naive approach that produces a batch of convolution triples for each specific size will ultimately produce a huge overhead of unused convolution triples and hence will be inefficient.

New Packing Methods. We solve this issue by developing new *packing methods* for convolutions, i.e., we pack the entries of images and filters suitably into plaintexts of the underlying encryption scheme and then use the special multiplicative structure of the plaintext and ciphertext spaces to compute a complete convolution (or multiple convolutions) with a single ciphertext multiplication. In line with some of the most recent packing methods, we avoid costly ciphertext rotations and maskings – primitives used in [14] and most of the related work (cf. Section 1.2). This simplifies the protocols and reduces the computational load for the parties, while still utilizing most of the capacity of ciphertext operations. We embed our method into a new general framework that (apart from our new and other recent packing methods) also directly supports the computation of scalar products, matrix multiplications, different types of convolutions, and potentially other operations. In particular, our general framework describes a wide class of packing methods that might be of independent interest and can potentially be used for other applications outside of ML.

Overall, we build a new flexible offline phase that can be instantiated with both our new and already known packing methods. We solve security issues of recent packings like Bian et al.'s packing [6] in our setting and prove our protocols secure against attacks by active adversaries as long as one party remains honest.

Implementation. We implement our techniques as an extension to MP-SPDZ [29], the currently most efficient implementation of SPDZ-like protocols. Our implementation [47] provides convolution optimized extensions for both the LowGear protocol and the HighGear protocol of Overdrive [31]. This allows our protocols to

[1] As already mentioned in the seminal work on transformers [52], matrix multiplications in a classical transformer can also be viewed as 1×1 convolutions.

be applied both in setups with a small number of parties, where LowGear is most efficient, and a setup with many parties, where HighGear scales better. More precisely, LowGear scales quadratically in the number of parties and uses cheap primitives, which make it well-suited for low-party setups. In contrast, HighGear scales linearly in the number of parties and uses more expensive ZKP and SHE protocols. Settings with both low and high number of parties are realistic and we want to support as many settings as possible. There are application scenarios where the distribution of the data dictates the number of parties, e.g., there are cases where one party holds the inputs and one party holds the model (i.e., a two-party setting) or cases where the data (i.e., model and/or inputs) is naturally distributed among many parties. Another scenario is the client-server setting. Here, a setup with two servers is usually most efficient, but a setup with more servers reduces the risk that *all* of them collude to break security. While we focus on the low-party setup in our evaluation, we also show feasibility of our approach for more than three parties. We remark that related work usually concentrates on the low-party setup only.

We also use the optimized zero-knowledge proofs (ZKPs) introduced in TopGear [3] but extended the ZKPs to also support non-trivial packing methods. By implementing our protocols on top of the already highly optimized MP-SPDZ framework, we receive a better overall performance in ML applications since we get the improved performance for convolutions, while maintaining the currently best performance of MP-SPDZ for all other operations.

We use our implementation to give an extensive evaluation of our methods and compare it to the current state-of-the-art implemented in MP-SPDZ, as well as the state-of-the-art research results of [14]. On benchmarks for ResNet50, we outperform the state-of-the-art by a factor of 3 to 4.8 (depending on the network setup). For depthwise convolutions, our protocols are up to 26× faster than the state-of-the-art.

1.1 Summary of Our Contributions

- We introduce the first actively secure MPC protocol with direct support for *convolutions* (Sections 5 and 6).
- We introduce a new efficient convolution triple generation protocol as part of our offline phase (Sections 6.2 and 6.3).
- The convolution triple production is instantiated with multiple new and recent packing methods (Sections 6.2 and 6.3 and Appendix F.3.2), which might be of independent interest.
- We prove that our online and offline protocols are actively secure even if only one party is honest (Appendices E and F). In particular, we solve existing security issues with recent packings like Bian et al. [6] in the active adversary setup.
- We present new and more efficient packing methods for convolutions (Section 4). This includes the first packing method for depthwise convolutions based on polynomial multiplication in the underlying cyclotomic ring (Section 4.3). Our packings do not use ciphertext rotations or maskings.
- We have implemented our complete protocol (offline phase and online phase) [47], including several packing methods, as an extension of MP-SPDZ [29], which is the state-of-the-art implementation for SPDZ-like protocols (Section 7).

Table 1: Related Work in Secure Evaluation of Convolutions

Reference	Security	Majority	n	Conv. Comp.	Cipher. Ops.[a]
MiniONN [39]	semi	dish.	2	matmul	pc
[26]	semi	dish.	2	mul	cc, rot
GAZELLE [27][b]	semi	dish.	2	mul	pc, rot
LoLa [9]	semi	dish.	2	matmul	pc/cc, rot
[14]	mal.	dish.	any	matmul	cc, rot
[16, 32]	any	any	any	matmul	–
CryptGPU [49]	semi	hon.	3	conv	–
APAS [6]	semi	dish.	2	conv	pc[c]
CrypTen [34]	semi	dish.	any	matmul	–
TenSEAL [5]	semi	dish.	2	matmul	pc, rot
Cheetah [25]	semi	dish.	2	conv	pc
HeLayers [1]	semi	dish.	2[d]	matmul	pc/cc, rot
Ours	mal.	dish.	any	conv	pc/cc

[a] plaintext-ciphertext multiplications (pc), ciphertext-ciphertext multiplications (cc), and rotations (rot)
[b] used and/or extended in DELPHI [42], CrypTFlow2 [46], GALA [54], HEAR [33], and [37]; same setting and operations as CHET [19] and HEMET [41]
[c] matrix-vector multiplication of a plaintext matrix and an encrypted vector
[d] one model owner and a compute server in addition to any number of clients

- We have evaluated our implementation against generic SPDZ as well as [14], the state-of-the-art actively secure protocol for matrix multiplications (Section 7). Our results show that our specialized operations significantly improve the online and offline runtime compared to the related work. Our advantage in the offline phase is 4.82× in the LAN setting and 3.01× in the WAN setting for convolutions as in ResNet50. For depthwise convolutions, our approach is up to 18.59× (LAN) or up to 26.53× (WAN) faster. We also observe improvements of up to 40.15× (LAN) or 41.84× (WAN) in the online phase.

1.2 Related Work

In Table 1, we summarize recent MPC protocols and focus on the realization of secure convolutions. As can be seen, most research focuses on a very specific setting: 2-party or 3-party computations with passive (semi-honest) security. In contrast, our protocols aim at active security rather than passive security and allow for a dishonest majority of malicious parties, similar to the setup of, e.g., [14]. Table 1 also includes an overview of technical realizations of convolutions and ciphertext operations used in the different protocols. Convolutions are usually reduced to either field multiplications (mul), matrix multiplications (matmul), or computed directly as convolutions (conv). Most protocols realize these with different ciphertext operations and packing methods. Element-wise (SIMD) multiplication of encrypted data and ciphertext rotations (to align the data encoded in ciphertexts for multiplications) are used almost exclusively.

Note that these rotations come with two downsides, which we want to avoid with our protocols. Firstly, ciphertext rotations are computationally expensive and require additional key material. For

example, for [14] one has to generate around 24.72 GB of non-trivial data in an actively secure way. Secondly, if plaintext rotations are used (e.g., in a LowGear-style protocol) one has to make sure that the plaintexts are rotated correctly. This might require additional ZKPs or similar constructions to guarantee security in the presence of misbehaving parties. The same is true for packing methods that require parties to tile and/or replicate data in a specific way (e.g., [1]).

Furthermore, some recent works [6, 33, 37] aim to even perform multiple convolutions in parallel using specialized packing methods. A notably unique HE technique [6] uses multiplications of plaintext matrices with ciphertexts for this. There are also exceptions that do not (necessarily) use HE, such as CryptGPU [49], which uses field multiplication without the use of HE, and [16, 32, 34], which build their protocols generically on matrix multiplications (which might in turn be realized with HE but other techniques are possible as well).

Orthogonal to securely computing convolutions, there is also work on verifiable convolutions, i.e., proving in zero-knowledge that convolutions are performed correctly [40, 51]. We note that our protocols guarantee correct computation of convolutions towards the parties who participate in the protocol. For a discussion on other privacy-preserving technologies for ML we refer the reader to [10].

2 PRELIMINARIES AND NOTATION

We use the notation $[a..b) \coloneqq \{a, a+1, \dots, b-1\}$ for ranges of integers. We also use the shorthand $[..b) \coloneqq [0..b)$. For (image) domains, we abuse the notation and write $h \times w$ instead of $[0..h) \times [0..w)$. Furthermore, we index vectors (and matrices and tensors) with brackets, i.e., $\boldsymbol{v}[i]$ is the i-th element of vector $\boldsymbol{v}$. Polynomials or vectors of polynomials are depicted sans-serif, e.g., x or $\mathbf{v}$.

2.1 Integer Polynomials and Multiplication

Let $\mathcal{R} = \mathbb{Z}[X]/\Phi_m(X)$ be the integer polynomials modulo the m-th cyclotomic polynomial $\Phi_m(X) = X^N + 1$ for $m = 2N$ a power of two. Let $\boldsymbol{a} \in \mathbb{Z}^N$ be the vector of coefficients of $\mathsf{a} \in \mathcal{R}$, i.e., $\mathsf{a} = \sum_{i=0}^{N-1} \boldsymbol{a}[i] \cdot X^i$. Analogously define $\boldsymbol{b} \in \mathbb{Z}^N$ for $\mathsf{b} \in \mathcal{R}$. Then the vector of coefficients $\boldsymbol{c} \in \mathbb{Z}^N$ to the product $\mathsf{c} = \mathsf{a} \cdot \mathsf{b} \in \mathcal{R}$ can be computed with a negacyclic convolution:

$$\boldsymbol{c}[i] = (\boldsymbol{a} \circledast \boldsymbol{b})[i] \coloneqq \sum_{j=0}^{N-1} (-1)^{i-j \operatorname{div} N} \cdot \boldsymbol{a}[j] \cdot \boldsymbol{b}[i-j \bmod N]. \quad (1)$$

To verify this equation recall that $X^N \bmod \Phi_m(X) = -1$. To simplify notation we will usually identify $\mathcal{R}$ and $\mathbb{Z}^N$. This allows us to compute (negacyclic) convolutions with encryption schemes that support the homomorphic multiplication of encrypted polynomials, i.e., $\boldsymbol{a}$ or $\boldsymbol{b}$ (or both) are encrypted and we are able to obtain an encrypted product $\boldsymbol{c}$.

2.2 Convolutions in Machine Learning

To simplify the exposition, we restrict ourselves to two-dimensional convolutions, which is very common in image processing [35, 38, 44, 53]. However, note that our results also carry over to the one-dimensional case and to higher dimensions (e.g., 3d convolutions).

Let R be a commutative ring and denote by R^D the functions $\mathbb{Z}^k \to R$ with support in the finite *domain* $D \subset \mathbb{Z}^k$ ($k \in \mathbb{N}$), i.e., functions that are zero outside of D. A discrete 2d convolution $* : R^{D'} \times R^D \to R^{D''}$ with domain D'' of an image $\boldsymbol{a} \in R^D$ ($D = h \times w$) and a filter $\boldsymbol{f} \in R^{D'}$ ($D' = h' \times w'$) is defined as

$$(\boldsymbol{f} * \boldsymbol{a})[y, x] \coloneqq \sum_{y'=0}^{h'-1} \sum_{x'=0}^{w'-1} \boldsymbol{f}[y', x'] \cdot \boldsymbol{a}[y - y', x - x'], \quad (2)$$

for $(y, x) \in D''$. We call $*$ a convolution with

- *valid padding* if $D'' = [h' - 1..h) \times [w' - 1..w)$, i.e., (2) accesses only indices of $\boldsymbol{a}$ such that $(y - y', x - x') \in D$ for all $(y', x') \in D'$. This is the case where the filter and the image overlap completely (e.g., in Fig. 1a).
- *same padding* if $D'' = [\lfloor h'/2 \rfloor..h + \lfloor h'/2 \rfloor) \times [\lfloor w'/2 \rfloor..w + \lfloor w'/2 \rfloor)$, i.e., $|D| = |D''|$ are of the same size and a suitable number of zero values of $\boldsymbol{a}$ outside of D are accessed by (2). As can be seen in Fig. 1b, this means that the image is extended by roughly half the size of the filter in each direction.
- *full padding* if $D'' = (h + h' - 1) \times (w + w' - 1)$, i.e., D'' is chosen such that all (possibly) non-zero summands in (2) are accessed. This is the case where the filter and the image overlap in at least one entry. For this, the image is extended by the filter size (minus one) in each direction. Figure 1c visualizes this.

For our packing schemes in Section 4, we will use $\operatorname{up}(D'') = \operatorname{up}([l..h'') \times [v..w'')) \coloneqq (h'', w'')$, i.e., the smallest upperbound for D'' in each spatial direction that is not included in D''.

Note that the *valid* output of the convolution is (in general) smaller than the input image and with full padding, the output is larger than the input image. However, for $h' = w' = 1$, these three types of convolution are equivalent. A simple way to compute arbitrary convolutions is to compute full convolutions and simply discarding some parts of the output to get results with same or valid padding. The same is true for strided convolutions, where we only want the results for, e.g., every second coordinate.

A related operation is the cross-correlation

$$(\boldsymbol{f} \star \boldsymbol{a})[y, x] \coloneqq \sum_{y'=0}^{h'-1} \sum_{x'=0}^{w'-1} \boldsymbol{f}[y', x'] \cdot \boldsymbol{a}[y + y', x + x'] \quad (3)$$

(for real-valued $\boldsymbol{f}$), which is equivalent to a convolution of a $\boldsymbol{a}$ and a mirrored $\boldsymbol{f}$ (see Appendix C.1). Therefore, we will only talk about convolutions in the following, even if we might want to compute cross-correlations from time to time.

In ML applications, slightly more complex operations built on 2d convolutions are considered. For 4d tensors $\boldsymbol{a}$ and $\boldsymbol{f}$ with domains $D = b \times h \times w \times d$ and $D' = d' \times d \times h' \times w'$, respectively, we define

$$\operatorname{conv2d}(\boldsymbol{a}, \boldsymbol{f})[i, \cdot, \cdot, j] \coloneqq \sum_{k=0}^{d-1} \boldsymbol{f}[j, k, \cdot, \cdot] \star \boldsymbol{a}[i, \cdot, \cdot, k] \quad (4)$$

for each $(i, j) \in b \times d'$. The padding modes (full, same, valid; zero-padding) and strides then apply to the individual 2d cross-correlations (convolutions), i.e., to the finite $D''_{ijk} \subset \mathbb{Z}^2$ such that $\boldsymbol{f}[j, k, \cdot, \cdot] \star \boldsymbol{a}[i, \cdot, \cdot, k] \in R^{D''_{ijk}}$ for $(i, j, k) \in b \times d' \times d$. Usually, all

D''_{ijk} are the same, and we can simply define $\mathrm{up}(D'') \coloneqq \mathrm{up}(D''_{ijk})$ for the 4d domain D'' of the output. In addition to this, there is also the so-called *depthwise (separable) convolution*

$$\mathrm{dconv2d}(\boldsymbol{a}, \boldsymbol{f})[i, \cdot, \cdot, j] \coloneqq \boldsymbol{f}[j, \cdot, \cdot] \star \boldsymbol{a}[i, \cdot, \cdot, j] \tag{5}$$

where $\boldsymbol{f}$ is now a 3d tensor with domain $D' = d \times h' \times w'$. The latter is used, for example in [2, 15, 24, 48], to reduce the computational load and the number of trainable parameters compared to conv2d.

2.3 MPC, Secret-Sharing, and SPDZ

The currently most efficient actively secure MPC protocols are based on the fundamental results in SPDZ [18]. By now there is a vast amount of work that builds on and extends the original SPDZ protocol, e.g., [3, 14, 17, 31] (cf. [45] for an overview). We see our work as an extension to the SPDZ framework or as a *SPDZ-like protocol*. What follows is a short overview of the most important concepts necessary for this work.

2.3.1 Secret-Sharing. For security against a dishonest majority, i.e., in our setup all but one party might be corrupted, SPDZ uses a full-threshold *additive secret-sharing*. For this work, we restrict ourselves to a finite prime field $\mathbb{F}_p$. We call $[x]_i$ the share of the secret $x \in \mathbb{F}_p$ and party P_i. We have that $x = \sum_{i=0}^{n-1} [x]_i$ for n parties, where (usually) all but one share are chosen uniformly at random – the last share is chosen such that the sum of all shares *reconstructs* the secret. *Reconstruction* or *opening* of a share requires the parties to broadcast their share and each party computes the sum locally: $x = \mathrm{open}([x]_i) = \sum_{i=0}^{n-1} [x]_i$. This secret-sharing scheme is additive, i.e., $[x+y]_i = [x]_i + [y]_i$, $[c \cdot x]_i = c \cdot [x]_i$, and $[x+c]_i = [x]_i + c \cdot [1]_i$ for all sharings of $x, y \in \mathbb{F}_p$, constant $c \in \mathbb{F}_p$, and $[1]_i = \delta_i$ the Kronecker delta ($\delta_i = 1$ if $i = 0$ and zero otherwise).

For active security, this secret-sharing is enhanced by so-called SPDZ MACs. An *authenticated share* is then defined as $[\![x]\!]_i \coloneqq ([x]_i, [\alpha \cdot x]_i)$ for a (secret) MAC key α. Addition of authenticated shares is done analogously to the addition of shares above, but for scalar addition $[\![x + c]\!]_i = ([x]_i + c \cdot [1]_i, [\alpha \cdot x]_i + c \cdot [\alpha]_i)$, each party P_i additionally requires a share $[\alpha]_i$ of the MAC key. The MAC shares are used in a MAC check procedure (cf. Fig. 10 in Appendix A.4) to verify that the parties correctly computed and opened shares. The MAC key is not revealed during a (successful) MAC check and many MAC checks can be combined into a single check [17].

2.3.2 Beaver Multiplication. To multiply two shares $[\![x]\!]_i, [\![y]\!]_i$, one can no longer only perform local operations on shares (as for linear operations above). Instead, Beaver's multiplication technique [4] is used: Given a triple $([\![a]\!]_i, [\![b]\!]_i, [\![c]\!]_i)$ with $c \coloneqq a \cdot b$ and $u \coloneqq x - a, v \coloneqq y - b$, we can compute a share of the product as

$$[\![x \cdot y]\!]_i = [\![c]\!]_i + [\![a]\!]_i \cdot v + u \cdot [\![b]\!]_i + u \cdot v. \tag{6}$$

The values u and v are obtained by reconstruction of the shares $[x]_i - [a]_i, [y]_i - [b]_i$ where the correctness of the openings is guaranteed by the MAC check. The Beaver triple multiplication can be generalized to arbitrary bilinear operations, i.e., we can replace the multiplication in (6) (and in the definition of c) by other bilinear operations like matrix multiplications or convolutions [14, 43].

2.4 Homomorphic Encryption and BGV

Similar to other SPDZ-like protocols [3, 17, 18, 31], we base our offline phase on the BGV encryption scheme of Brakerski et al. [8]. We will therefore shortly repeat the basics of the BGV scheme, while more details can be found in Appendix A.1.

Let $\mathcal{R}_p = \mathbb{F}_p[X]/\Phi_m(X) = \mathcal{R}/p\mathcal{R}$ for p a prime with $p = 1 \bmod m$. Let $p < q$, q a prime, and identify $\mathcal{R}_p$ with a subset of $\mathcal{R}_q$ in the usual way (cf. [18]). Let $(\mathrm{pk}, \mathrm{sk}) \in \mathcal{R}_q^2 \times \mathcal{R}_q$ be a BGV public key/private key pair, $C \coloneqq \mathcal{R}_q^2$, $\mathrm{enc}_{\mathrm{pk}} : \mathcal{R}_p \times \mathcal{R}_q^3 \mapsto C$ the encryption function, and $\mathrm{dec}_{\mathrm{sk}} : C \mapsto \mathcal{R}_p$ the decryption function. We use the following notation for encrypted values: $\langle \mathsf{x} \rangle$, e.g., $\langle \mathsf{x} \rangle_{\mathrm{pk}} = \mathrm{enc}_{\mathrm{pk}}(\mathsf{x}, \mathbf{r})$, where we omit the explicit dependency on the key if it is clear from the context.We also define homomorphic operations on ciphertexts and denote them with operations of the same semantics, e.g., $\mathsf{x} \cdot \langle \mathsf{y} \rangle$ for plaintext-ciphertext multiplication. For further details, e.g., the definition of the encryption and decryption functions, how encryption randomness has to be chosen, and how ciphertext operations (addition and multiplication) are defined, see Appendix A.1

3 CONVOLUTION PACKING

There is an obvious similarity between the multiplicative structure of $\mathcal{R}$ depicted in (1) and (1d versions of) the convolutions shown in (2) and (3). Indeed, with the zero-padding and a large enough N, we have

$$(\boldsymbol{f} * \boldsymbol{a})[i] = (\boldsymbol{f} \circledast \boldsymbol{a})[i] \tag{7}$$

and – as mentioned before – we can easily express cross-correlations as convolutions.

This similarity can be used to represent convolutions as operations on $\mathcal{R}$ and then use the homomorphic multiplication of BGV ciphertexts in SPDZ-like protocols to securely (and efficiently) compute convolutions in our MPC protocol. For (7), one could, for example, compute $\boldsymbol{f} * \boldsymbol{a} = \mathrm{dec}(\langle \boldsymbol{f} \rangle \cdot \langle \boldsymbol{a} \rangle)$.

However, two problems remain before we can use this in practice. Firstly, (7) holds a priori only for 1d convolutions but we need support for higher-dimensional (e.g., 2d) convolutions. Fortunately, there is a standard way to represent higher dimensional convolutions in terms of 1d convolutions. This construction is described in Section 3.2.1 and allows us to restrict ourselves to the case of 1d convolutions in most cases. Secondly, N is often quite large in MPC protocols. In order to use the full potential of $\mathcal{R}$, we therefore need to utilize a large fraction of the convolved slots, usually by performing multiple convolutions at once. Both problems can be simultaneously addressed by so called *packing methods*. We describe a general framework for packing methods next.

3.1 General Framework for Convolution Packing

Before we describe concrete packing methods for convolutions, we establish a general framework for packing methods that can be used to evaluate arbitrary bilinear operations like simple field multiplications, matrix products, or convolutions. For this, let $\mathrm{op} : R^D \times R^{D'} \to R^{D''}$ be a R-bilinear operation, i.e., for $\boldsymbol{f}, \boldsymbol{f'} \in R^{D'}, \boldsymbol{a}, \boldsymbol{a'} \in R^D$, and $s \in R$ we have $\mathrm{op}(\boldsymbol{a} + s \cdot \boldsymbol{a'}, \boldsymbol{f}) = \mathrm{op}(\boldsymbol{a}, \boldsymbol{f}) + s \cdot \mathrm{op}(\boldsymbol{a'}, \boldsymbol{f})$

and $\mathrm{op}(\boldsymbol{a}, \boldsymbol{f} + s \cdot \boldsymbol{f}') = \mathrm{op}(\boldsymbol{a}, \boldsymbol{f}) + s \cdot \mathrm{op}(\boldsymbol{a}, \boldsymbol{f}')$. Similarly, let $\mathrm{op}_{\mathcal{R}} : \mathcal{R}^u \times \mathcal{R}^v \to \mathcal{R}^w$ be an R-bilinear map for some $u, v, w \in \mathbb{N}$.

Definition 3.1. Let $\mathrm{packi} : R^D \to \mathcal{R}^u$, $\mathrm{packf} : R^{D'} \to \mathcal{R}^v$, $\mathrm{unpackr} : \mathcal{R}^w \to R^{D''}$. We say (packi, packf, unpackr) is a *packing method* w.r.t. $(\mathrm{op}, \mathrm{op}_{\mathcal{R}})$ if for any $\boldsymbol{a} \in R^D$, $\boldsymbol{f} \in R^{D'}$

$$\mathrm{op}(\boldsymbol{a}, \boldsymbol{f}) = \mathrm{unpackr}(\mathrm{op}_{\mathcal{R}}(\boldsymbol{g}, \boldsymbol{b})), \tag{8}$$

where $\boldsymbol{b} = \mathrm{packi}(\boldsymbol{a})$ and $\boldsymbol{g} = \mathrm{packf}(\boldsymbol{f})$, i.e., $\boldsymbol{a}$ is packed with packi, $\boldsymbol{f}$ is packed with packf, then the bilinear operation $\mathrm{op}_{\mathcal{R}}$ is evaluated on the packed vectors $\boldsymbol{b}$ and $\boldsymbol{g}$ and the result is then unpacked with unpackr.

To avoid confusion we will often add additional arguments for the bilinear operations, e.g., write $\mathrm{packi}(\mathrm{op}, D, D', \boldsymbol{a})$ instead of $\mathrm{packi}(\boldsymbol{a})$. Additionally, most of the discussed packing schemes use the standard choice of $\mathrm{op}_{\mathcal{R}} = \circledast$ on $\mathcal{R} \simeq \mathbb{Z}^N$ and $u = v = w = 1$, i.e., we express the operation op as a negacyclic convolution or as a polynomial multiplication in $\mathcal{R}$. The latter can be performed securely with homomorphic encryption (e.g., BGV; cf. Section 2.4) and packing (or unpacking) can be performed before the encryption (or after the decryption, respectively). Note that (8) then becomes

$$\mathrm{op}(\boldsymbol{a}, \boldsymbol{f}) = \mathrm{unpackr}(\boldsymbol{g} \circledast \boldsymbol{b}) \tag{9}$$

with $\boldsymbol{b}, \boldsymbol{g} \in \mathcal{R}$ for the standard case.

Remark 3.1. While not required by our definition, please note that all packing methods treated in this paper have the additional feature that one can also directly unpack $\mathrm{packi}(\boldsymbol{a})$ and $\mathrm{packf}(\boldsymbol{f})$ again, i.e., there are maps $\mathrm{unpacki} : \mathcal{R}^u \to R^D$ and $\mathrm{unpackf} : \mathcal{R}^v \to R^{D'}$ with $\mathrm{unpacki} \circ \mathrm{packi} = \mathrm{id}_{R^D}$, $\mathrm{unpackf} \circ \mathrm{packf} = \mathrm{id}_{R^{D'}}$.

An important subclass of packing methods are those induced by injective functions on the index sets, which we describe next. Let $\iota : D \to F$ be an injective map and $\pi : E \coloneqq \mathrm{image}(\iota) \to D$ the surjective (left) inverse, i.e. $\pi \circ \iota = \mathrm{id}_D$. To π we can associate the natural pullback map $\pi^* : R^D \to R^E$ on the corresponding function spaces. This means, $\pi^*(\boldsymbol{a})[\boldsymbol{i}] \coloneqq \boldsymbol{a}[\pi(\boldsymbol{i})]$ for all $\boldsymbol{i} \in E$ and $\pi^*(\boldsymbol{a})[\iota(\boldsymbol{x})] = \boldsymbol{a}[\boldsymbol{x}]$ for $\boldsymbol{x} \in D$. Note that R^E embeds into R^F by extending function by zero and hence we can interpret π^* as a map to R^F, i.e., $\pi^*(\boldsymbol{a})[\boldsymbol{j}] = 0$ for $\boldsymbol{j} \in F \setminus E$. Similarly, define the pushforward $\iota_* : R^F \to R^D$ by $\iota_*(\boldsymbol{b})[\boldsymbol{x}] \coloneqq \boldsymbol{b}[\iota(\boldsymbol{x})]$ for $\boldsymbol{x} \in D$. We say π^* and ι_* are induced by the injective map ι.

Definition 3.2. If (packi, packf, unpackr) is a packing method and the procedures $\mathrm{packi} : R^D \to \mathcal{R}^u$, $\mathrm{packf} : R^{D'} \to \mathcal{R}^v$, $\mathrm{unpackr} : \mathcal{R}^w \to R^{D''}$ are induced by injective maps $\mathrm{mapi} : D \to [..N)^u$, $\mathrm{mapf} : D' \to [..N)^v$, $\mathrm{mapr} : D'' \to [..N)^w$ then we say that the packing method is induced by (mapi, mapf, mapr).

In other words, mapi, mapf, mapr correspond to ι above and packi, packf are defined as pullbacks π^* while unpackr is defined as pushforward ι_*.

Remark 3.2. One of the most important examples of a packing method *not* induced by functions is the CRT packing discussed in Appendix B. In particular, this encoding is used in the generation of Beaver triples where we use our general framework in the special case $D = D' = D'' = [..N)$, $R = \mathbb{F}_p$ a finite field, $\mathrm{op} = \odot$ component-multiplication, and $\mathrm{op}_{\mathcal{R}}$ the standard choice of polynomial multiplication as described above.

3.2 Recent Packing Methods

In the following we want concentrate on induced packing methods recently introduced in the literature. Additionally, we describe Bian et al.'s packing method in Appendix C.2. In Section 4 we present new packing methods – partially based on existing methods and completely new ones.

3.2.1 Multidimensional Convolution Packing. We first want to show how we can include 2d convolutions in our framework. This will also be used as part of all other packing methods. Therefore, our description is similar to the packing methods presented in [6, 25]. However, the version presented here is not bound to a specific padding but rather supports all popular padding methods (with zero-padding) discussed above.

Theorem 3.3. *Let $\boldsymbol{a}$ be a $D = h \times w$ (2d) image and $\boldsymbol{f}$ a $D' = h' \times w'$ (2d) filter. Choose D'' according to the padding mode and let $(h'', w'') = \mathrm{up}(D'')$. For N with $h'' \cdot w'' \le N$ define $\phi : h'' \times w'' \to [..N), (y, x) \mapsto y \cdot w'' + x$. Then the packing of $\boldsymbol{a}$ and $\boldsymbol{f}$ as (1d) N-vectors $\boldsymbol{b} = \mathrm{packi}(*, D, D', \boldsymbol{a})$ and $\boldsymbol{g} = \mathrm{packf}(*, D, D', \boldsymbol{f})$ satisfies*

$$(\boldsymbol{f} * \boldsymbol{a})[y, x] = \mathrm{unpackr}(*, D, D', \boldsymbol{g} \circledast \boldsymbol{b})[y, x] \tag{10}$$

for $(y, x) \in D''$, where the packing method (packi, packf, unpackr) *is induced by* $\mathrm{mapi}(\boldsymbol{x}) = \mathrm{mapf}(\boldsymbol{x}) = \mathrm{mapr}(\boldsymbol{x}) = \phi(\boldsymbol{x})$.

Recall that the output domains D'' corresponds to the popular padding modes in Section 3. A proof for Theorem 3.3, as well as the corresponding version for cross-correlations, can be found in Appendix C.1.

3.2.2 Huang et al.'s Convolution Packing. Huang et al. present a packing method in [25] with the goal to perform a whole conv2d operation at once (given a large enough N). Let $\phi(i, j, y, x) = ((i \cdot d + j) \cdot h + y) \cdot w + x$ be the canonical indexing into a (flattened 4d) $d' \times d \times h \times w$ tensor. Huang et al. encode a $1 \times h \times w \times d$ image[2] $\boldsymbol{a}$ and a $d' \times d \times h' \times w'$ filter $\boldsymbol{f}$ as $\boldsymbol{b}[\phi(0, j, y, x)] \coloneqq \boldsymbol{a}[0, y, x, j]$ and $\boldsymbol{g}[\phi(d' - 1 - i, d - 1 - j, h' - 1 - y, w' - 1 - x)] \coloneqq \boldsymbol{f}[i, j, y, x]$ (with $\boldsymbol{b}[\,\cdot\,] = 0$ and $\boldsymbol{g}[\,\cdot\,] = 0$ at other positions), respectively. Then, $\mathrm{conv2d}(\boldsymbol{f}, \boldsymbol{a})[0, y, x, i] = (\boldsymbol{g} \circledast \boldsymbol{b})[\varphi]$ for $(i, y, x) \in d' \times (h - h' + 1) \times (w - w' + 1)$ and $\varphi = \phi(d' - 1 - i, d - 1, h' - 1 + y, w' - 1 + x)$. This corresponds to a conv2d for a batch size $b = 1$ and valid padding (with shifted x and y indices of the output compared to the above description). In the framework of Section 3.1, we would have $\mathrm{mapi}(0, y, x, j) = \phi(0, j, y, x)$, $\mathrm{mapf}(j', j, y, x) = \phi(d' - 1 - j', d - 1 - j, h' - 1 - y, w' - 1 - x)$, and $\mathrm{mapr}(0, y, x, j') = \phi(d' - 1 - j', d - 1, h' - 1 + y, w' - 1 + x)$.

In Section 4.2, we present a more efficient generalization which can use the same $\boldsymbol{g}$ in multiple batches. This will be particularly useful when the packing is applied to encrypted versions of $\boldsymbol{g}$ and $\boldsymbol{b}$, since then the encryption of $\boldsymbol{g}$ has to be sent only once. We remark that sending ciphertexts and proving their correctness with zero-knowledge proofs is expensive and our approach reduces these costs, both the bandwidth and the runtime, compared to the original version where each batch is handled as a completely new and unrelated conv2d operation.

[2] Huang et al. use a $h \times w \times d$ image and we extended this trivially to a 4d tensor to be compatible with our framework.

4 NEW PACKING METHODS

Here, we present new packing methods for convolutions. This includes the first packing method for depthwise convolutions that can be realized with only homomorphic polynomial multiplications. Appendix D contains the proofs of correctness for each of the packing methods.

4.1 Simple Convolution Packing

Our first simple convolution packing is based on the multidimensional packing of Section 3.2.1. For a complete conv2d computation, we deviate slightly from the standard choice $\mathrm{op}_{\mathcal{R}} = \bar{*}$ and use instead a linear combination of d partial convolutions in (8), i.e., $\mathrm{op}_{\mathcal{R}} : \mathcal{R}^d \times \mathcal{R}^d \to \mathcal{R}, ((\boldsymbol{g}_k)_{k\in d}, (\boldsymbol{b}_k)_{k\in d}) \mapsto \sum_{k=0}^{d-1} \boldsymbol{g}_k \bar{*} \boldsymbol{b}_k$ with $(\boldsymbol{b}_k)_{k\in d} = \mathrm{packi}(\mathrm{op}, D, D', \boldsymbol{a})$ and $(\boldsymbol{g}_k)_{k\in d} = \mathrm{packf}(\mathrm{op}, D, D', \boldsymbol{f})$. This allows for the simple convolution packing below.

Theorem 4.1. *Let $\boldsymbol{a}$ be a (4d) $D = b \times h \times w \times d$ tensor and let $\boldsymbol{f}$ be a (4d) $D' = d' \times d \times h' \times w'$ tensor. Choose D'' according to the padding mode and let $(h'', w'') = \mathrm{up}(D'')$. Let $\phi(i, j, y, x) = ((i \cdot d' + j) \cdot h'' + y) \cdot w'' + x$ be the canonical indexing into a (flattened 4d) $b \times d' \times h'' \times w''$ tensor. Let*

$$\mathrm{mapi}(i, y, x, j) = (\phi(i, 0, y, x), j) \in N \times d$$
$$\mathrm{mapf}(j', j, y, x) = (\phi(0, j', h' - 1 - y, w' - 1 - x), j) \in N \times d$$
$$\mathrm{mapr}(i, y, x, j') = \phi(i, j', y, x).$$

For the induced packing (packi, packf, unpackr) *and $\boldsymbol{b}_k, \boldsymbol{g}_k$ as above,*

$$\mathrm{conv2d}(\boldsymbol{a}, \boldsymbol{f}) = \mathrm{unpackr}(\mathrm{conv2d}, D, D', \sum_{k=0}^{d-1} \boldsymbol{g}_k \bar{*} \boldsymbol{b}_k). \tag{11}$$

Compared to Theorem 3.3 we have two additional dimensions index by i and j'. For each index pair (i, j') we map to a disjoint subset of $[..N)$ and then apply a modified version of Theorem 3.3 for $\star$ instead of $*$ (cf. Remark C.1). A single $\boldsymbol{g}_k \bar{*} \boldsymbol{b}_k$ then yields cross-correlations for each batch (b) and output channel (d') and a fixed input channel (d). We can then simply sum up all d sets of individual cross-correlations to get the full conv2d result. For details on the proof of Theorem 4.1 we refer to Appendix D.1. A visual example can be seen in Fig. 2. There, we abstract away the spatial dimensions with blocks, each representing an $h'' \times w''$ slice (by Theorem 3.3), and only focus on the remaining dimensions.

4.2 Generalization of Huang et al.'s Convolution Packing

Here, we present a slightly different (but more intuitive) extension to Huang et al.'s packing method [25] described in Section 3.2.2.

Theorem 4.2. *Let $\boldsymbol{a}$ be a (4d) $D = b \times h \times w \times d$ tensor and let $\boldsymbol{f}$ be a (4d) $D' = d' \times d \times h' \times w'$ tensor. Choose D'' according to the padding mode and let $(h'', w'') = \mathrm{up}(D'')$. Let $\phi(i, j, k, y, x) = (((i \cdot d' + j) \cdot d + k) \cdot h'' + y) \cdot w'' + x$ be the canonical indexing into a (flattened 5d) $b \times d' \times d \times h'' \times w''$ tensor. Let $\boldsymbol{b} = \mathrm{packi}(\mathrm{conv2d}, D, D', \boldsymbol{a})$ and $\boldsymbol{g} = \mathrm{packf}(\mathrm{conv2d}, D, D', \boldsymbol{f})$. Then,*

$$\mathrm{conv2d}(\boldsymbol{a}, \boldsymbol{f}) = \mathrm{unpackr}(\mathrm{conv2d}, D, D', \boldsymbol{g} \bar{*} \boldsymbol{b}) \tag{12}$$

for the packing method (packi, packf, unpackr) *induced by* $\mathrm{mapi}(i, y, x, j) = \phi(i, 0, j, y, x)$, $\mathrm{mapf}(j', j, y, x) = \phi(0, j', d - 1 - j, h' - 1 - y, w' - 1 - x)$, *and* $\mathrm{mapr}(i, y, x, j') = \phi(i, j', d - 1, y, x)$.

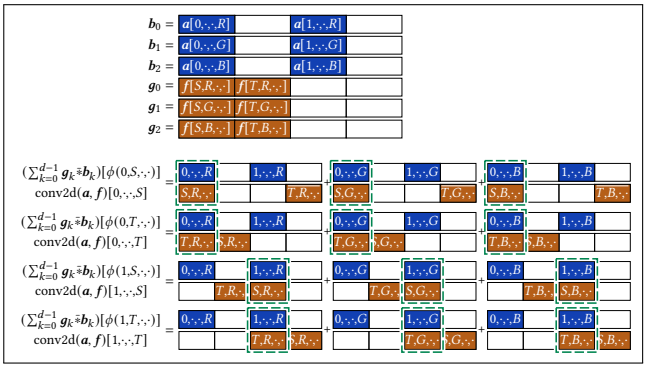

Figure 2: Example for the Simple Convolution Packing with $b = 2, d = 3, d' = 2$ (cf. (11)). The packing of the image $\boldsymbol{a}$ (blue) and the filter $\boldsymbol{f}$ (orange) is shown at the top. For better readability, we identify the input channels with the symbols R, G, B and the output channels with S, T. The computation of the convolution result $\mathrm{conv2d}(\boldsymbol{a}, \boldsymbol{f}) = \mathrm{unpackr}(\sum_{k=0}^{d-1} \boldsymbol{g}_k \bar{*} \boldsymbol{b}_k)$ is shown below. There, image and filter components are identified by their indices, i.e., we drop "$\boldsymbol{a}$" and "$\boldsymbol{f}$"; negated components of $\boldsymbol{f}$ are shown with bars on top. Values of $\boldsymbol{a}$ and $\boldsymbol{f}$ that are aligned on top of each other are convolved and the colored (green) boxes show where the partial convolution is non-zero. These (partial) convolutions are summed up as part of $\bar{*}$ and as part of the sum over k.

The intuition of this packing is similar to the simple packing (cf. Section 4.1): For a pair (i, j'), the image and filter are mapped to disjoint subsets of $[..N)$ such that (partial) convolutions for different batches or output dimensions do not overlap. Additionally, the index j along the input depth dimension d is chosen so the image and filter for the same input channel intentionally overlap. By the structure of the (negacyclic) convolution, the filter has to be reversed along this dimension. Then, the partial convolutions are summed up and the result can be obtained in the last slot along the d dimension. A proof can be found in Appendix D.2 and an example can be seen in Fig. 3. As for Fig. 2, we only ignore the spatial dimensions in the figure as this is handled by Theorem 3.3. An example for Huang et al.'s original packing would look similar (with the before mentioned limitations of only supporting $b = 1$ and valid padding) but the encoding of the filter and decoding of the result would both be reversed along the d' axis, as can be seen when comparing the equations for the packing methods.

With (12), we are then able to pack a whole conv2d operation into a single convolution. However, if the length of vectors (on which we can operate) is limited, e.g., when we work with homomorphic encryption with a fixed N, we cannot perform the whole operation. Instead, we should split a conv2d operation into smaller operations (which can be computed as in (12)). This is possible along all dimensions (batch dimension b, x dimension, y dimension, input depth dimension d, and output depth dimension d'; see [25] for their version and Section 6.5.2 for ours). Here, our generalization not only allows (previously impossible) direct realization of convolutions (including $b > 1$ and not only valid padding), but it also improves efficiency as we can move spatial splits (splitting along the x or y dimensions) into the batch dimension. For example,

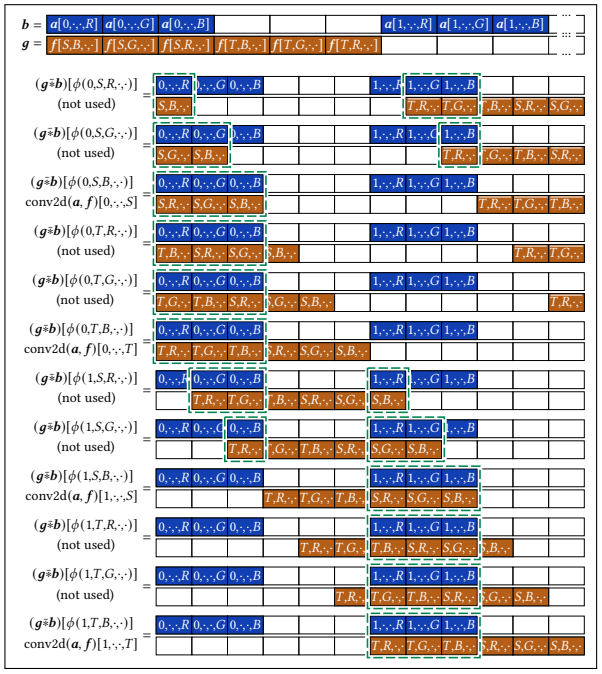

Figure 3: Example for the Generalization of Huang et al.'s Convolution Packing with $b = 2, d = 3, d' = 2$ (cf. (12)). The illustration of the image $\boldsymbol{a}$ (blue) and the filter $\boldsymbol{f}$ (orange) is similar to Fig. 2. The computation of the convolution result $\mathrm{conv2d}(\boldsymbol{a}, \boldsymbol{f}) = \mathrm{unpackr}(\boldsymbol{g} \bar{*} \boldsymbol{b})$ is shown below the packing of $\boldsymbol{a}$ and $\boldsymbol{f}$. Values of $\boldsymbol{a}$ and $\boldsymbol{f}$ that are aligned on top of each other are convolved and the colored (green) boxes show where the partial convolution is non-zero. These (partial) convolutions are summed up as part of $\bar{*}$.

in our evaluation, we compute the convolution of a $1 \times 224 \times 224 \times 3$ image $\boldsymbol{a}$ with a filter $\boldsymbol{f}$ as a convolution of a $4 \times 112 \times 112 \times 3$ image with the same filter $\boldsymbol{f}$ (recombining the four batches to a larger convolution triple afterwards). This is still a convolution of a single image with a single filter and therefore we only need ciphertexts for a single filter instead of four, as would be the case when we represent this as four separate $1 \times 112 \times 112 \times 3$ convolutions.

4.3 Depthwise Convolution Packing

With the exception of Bian et al. [6], no prior work seems to be able to pack convolutions in a way suitable for depthwise convolutions (see Appendix C.2 for a description of Bian et al.'s packing). Here, we present an alternative to [6] that represents multiple independent cross-correlations in a single convolution (instead of representing a depthwise convolution as matrix-vector product). We remark that our approach (unlike [6]) can also be used with encryption schemes, e.g., BGV and BFV, that natively only support the homomorphic multiplication of polynomials in $\mathcal{R}$.

Theorem 4.3. *Let $\boldsymbol{a}$ be a (4d) $D = b \times h \times w \times d$ tensor and let $\boldsymbol{f}$ be a (3d) $D' = d \times h' \times w'$ tensor. Choose D'' according to the padding mode and let $(h'', w'') = \mathrm{up}(D'')$. As in Theorem 4.2, let $\phi(i, j, k, y, x) = (((i \cdot d + j) \cdot d + k) \cdot h'' + y) \cdot w'' + x$ be the*

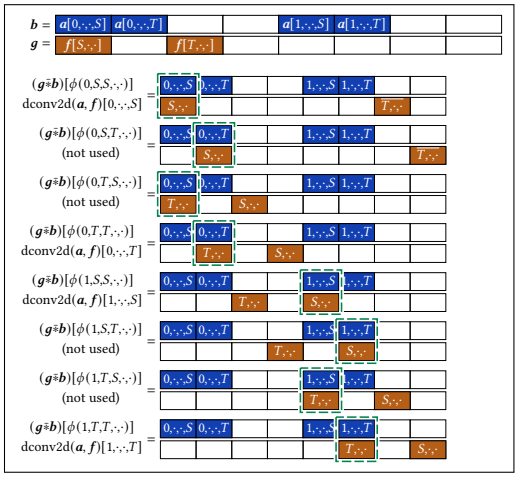

Figure 4: Example for the Depthwise Convolution Packing with $b = d = 2$ (cf. (13)). The illustration of the image $\boldsymbol{a}$ (blue) and the filter $\boldsymbol{f}$ (orange) is similar to Fig. 2. For better readability, we identify the input and output channels with the symbols S, T. The computation of the convolution result $\mathrm{dconv2d}(\boldsymbol{a}, \boldsymbol{f}) = \mathrm{unpackr}(\boldsymbol{g} \bar{*} \boldsymbol{b})$ is shown below the packing of $\boldsymbol{a}$ and $\boldsymbol{f}$. Values of $\boldsymbol{a}$ and $\boldsymbol{f}$ that are aligned on top of each other are convolved and the colored (green) boxes show where the partial convolution is non-zero. These (partial) convolutions are summed up as part of $\bar{*}$.

canonical indexing into a (flattened 5d) $b \times d \times d \times h'' \times w''$ tensor.[3] *Let $\boldsymbol{b} = \mathrm{packi}(\mathrm{dconv2d}, D, D', \boldsymbol{a})$ and $\boldsymbol{g} = \mathrm{packf}(\mathrm{dconv2d}, D, D', \boldsymbol{f})$. Then,*

$$\mathrm{dconv2d}(\boldsymbol{a}, \boldsymbol{f}) = \mathrm{unpackr}(\mathrm{dconv2d}, D, D', \boldsymbol{g} \bar{*} \boldsymbol{b}) \tag{13}$$

for the packing method (packi, packf, unpackr) *induced by* $\mathrm{mapi}(i, y, x, j) = \phi(i, 0, j, y, x)$, $\mathrm{mapf}(j, y, x) = \phi(0, j, 0, h' - 1 - y, w' - 1 - x)$, *and* $\mathrm{mapr}(i, y, x, j) = \phi(i, j, j, y, x)$.

Again, we construct the packing such that each batch i is mapped to disjoint region of $[..N)$. For the depth dimension, we pack the image and filter such that the $\boldsymbol{g} \bar{*} \boldsymbol{b}$ yields the convolution for each channel j of the image and channel j' of the filter. For the output, we simply select the partial convolution with $j = j'$. While this might seem wasteful, especially for small images, it is more efficient than emulating dconv2d with conv2d (cf. Section 7). A proof for Theorem 4.3 can be found in Appendix D.3.

5 MALICIOUSLY SECURE CONVOLUTIONS: THE ONLINE PHASE

As mentioned before in Section 2.3.2, given a convolution triple $(\llbracket \boldsymbol{a} \rrbracket_i, \llbracket \boldsymbol{f} \rrbracket_i, \llbracket \boldsymbol{c} \rrbracket_i)$ for uniformly random $\boldsymbol{a}, \boldsymbol{f}$ and $\boldsymbol{c} = \mathrm{conv2d}(\boldsymbol{a}, \boldsymbol{f})$, we can compute a (maliciously secure) convolution of a secret-shared image $\llbracket \boldsymbol{x} \rrbracket_i$ (with the same shape as $\boldsymbol{a}$) and a filter $\llbracket \boldsymbol{y} \rrbracket_i$ (with the same shape as $\boldsymbol{f}$) as a linear combination of the triple and

[3] Compared to Section 4.2, we have $d' = d$.

opened values $\boldsymbol{u} := \boldsymbol{x} - \boldsymbol{a}, \boldsymbol{v} := \boldsymbol{y} - \boldsymbol{f}$ (analogously to (6)), i.e.,

$$[\![\text{conv2d}(\boldsymbol{x}, \boldsymbol{y})]\!]_i = [\![\boldsymbol{c}]\!]_i + \text{conv2d}([\![\boldsymbol{a}]\!]_i, \boldsymbol{v}) + \text{conv2d}(\boldsymbol{u}, [\![\boldsymbol{f}]\!]_i) + \text{conv2d}(\boldsymbol{u}, \boldsymbol{v}). \quad (14)$$

With this, we obtain a share of the convolution result and can inductively compute an arbitrary function on shares – in particular any convolutional neural network – as we can compute all necessary operations on shares in a maliciously secure way (scalar multiplications and additions [18], fully connected layers and matrix multiplications [14], ReLUs and max pooling [12, 21], etc.). The security follows from the security of the individual operations (e.g., linear operations to compute (14)) and the bilinearity of conv2d [14, 43]. The same can be done for depthwise convolutions by simply replacing conv2d by dconv2d above. Strided convolutions and different paddings can be handled analogously.

The full protocol for the online phase Π_{online}, as well as the corresponding functionality $\mathcal{F}_{\text{online}}$ and the security proof of the following theorem can be found in Appendix E. Assuming the existence of $\mathcal{F}_{\text{offline}}$ – an ideal functionality for the offline phase that generates triples (cf. Section 6) – and $\mathcal{F}_{\text{rand}}$ that allows parties to sample random values in $\mathbb{F}_p$ (used in the MAC check, Fig. 10), we obtain the following theorem.

Theorem 5.1. *The online protocol Π_{online} securely implements the ideal functionality $\mathcal{F}_{\text{online}}$ in the $(\mathcal{F}_{\text{offline}}, \mathcal{F}_{\text{rand}})$-hybrid model.*

6 MALICIOUSLY SECURE CONVOLUTIONS: THE OFFLINE PHASE

The convolution triples used in the online phase are generated in the input-independent offline phase. Different design patterns for the offline phase can lead to drastically different performance characteristics of MPC protocols in different application setups (few parties, low latency communication; many parties, high latency communication; etc.) and they heavily influence the practicality of certain approaches. In order to be applicable to these different setups we instantiate our generic computation methods for convolutions discussed in Section 4 in multiple ways. Since all of the presented new offline protocols are based on homomorphic encryption we first describe the common pattern of these approaches in Section 6.1. We then introduce specialized protocols for the standard choice (9) in the case of a low number of parties based (similar to Overdrive's LowGear protocol [31]) in Section 6.2 and a larger number of parties (similar to Overdrive's HighGear protocol [31] or TopGear [3]) in Section 6.3. The protocols can be trivially extended to support the simple packing of Section 4.1. Additionally, we provide a linear homomorphic offline phase for Bian et al.'s packing technique in Appendix F.3.2.

6.1 General Construction

In Sections 3 and 4 we have seen how convolutions can be packed and then evaluated by a polynomial multiplication in a cyclotomic ring. We first restrict to a single polynomial multiplication and discuss the case of convolutions that cannot be represented as a single polynomial multiplication (without increasing N) in Section 6.5.2. To securely realize a polynomial multiplication we use the homomorphic properties of the BGV encryption scheme common in SPDZ-like protocols. Once we can compute convolutions in a secure way, we can construct the non-trivial third entry $\boldsymbol{c} = \text{conv2d}(\boldsymbol{a}, \boldsymbol{f})$ or a convolution triple. We remark that since our general framework also support simple field multiplication, we can also generate classical Beaver triples.

Sacrifice$(\text{op}, [\![\boldsymbol{a}]\!]_i, [\![\boldsymbol{f}]\!]_i, [\![\boldsymbol{f'}]\!]_i, [\![\boldsymbol{c}]\!]_i, [\![\boldsymbol{c'}]\!]_i)$**:** Generalized sacrificing to generate a triple with the correct correlation or fail.

1. All parties sample r with $\mathcal{F}_{\text{rand}}$.
2. Open $[\![\boldsymbol{u}]\!]_i = [\![r \cdot \boldsymbol{f} - \boldsymbol{f'}]\!]_i$ to obtain $\boldsymbol{u}$.
3. Open $[\![\boldsymbol{v}]\!]_i = [\![r \cdot \boldsymbol{c} - \boldsymbol{c'} - \text{op}(\boldsymbol{a}, \boldsymbol{u})]\!]_i$ to obtain $\boldsymbol{v}$.
4. Abort if $\boldsymbol{v} \neq \boldsymbol{0}$. Perform the MAC check for the openings of $\boldsymbol{u}$ and $\boldsymbol{v}$ and abort if the check fails.
5. Return $([\![\boldsymbol{a}]\!]_i, [\![\boldsymbol{f}]\!]_i, [\![\boldsymbol{c}]\!]_i)$.

ZKP(I)**:** Given an index set $I \subseteq [..N)$, compute ciphertexts for plaintexts $\boldsymbol{x}_i$ with $\boldsymbol{x}_i[l] = 0$ for $l \in I$ for each party and prove in ZK that a suitable witness exists. Output the own plaintext and the ciphertext for each party.

See Fig. 11 in Appendix A.4.

Figure 5: Utilities for the Offline Phases at Party P_i

Multiplication with homomorphic encryption schemes in our protocols follows the following pattern. First each party generates (random) shares locally and encrypts them with their public key. In order for a ciphertext to be used in a multiplication protocol, a party first has to show with a zero-knowledge proof (ZKP) that they know a plaintext witnesses and that the plaintext is well-formed. In particular, our ZKPs show that the plaintexts are valid packings, which reduces to showing that certain coefficients (depending on packi and packf) are zero. This in turn will imply that the sum of the shares, i.e., the shared secret, will have the same zero coefficients and therefore represents a valid packing.

Next, the parties multiply their shares with standard multiplication techniques, which are base on either linear homomorphic encryption in Section 6.2 or somewhat homomorphic encryption in Section 6.3. Additionally, the shares are authenticated in the usual way, i.e., by multiplication with ciphertexts of (shares of) the MAC key α. Furthermore, the parties check that the original shares were authenticated and that no error was introduced in the multiplication (or resharing for SHE-based protocols). The latter is done using a new extended *sacrificing* technique which we will introduce in Section 6.4.

6.2 Linear Homomorphic Offline Phase

In Fig. 7, we present the (convolution) triple generation of our protocol based on linear homomorphic encryption. Additional subprotocols can be found in Figs. 5 and 6. The construction is based on Overdrive's LowGear protocol [31] but extends it to generate triples for any bilinear operations that can be represented with the framework of Section 3.1. (Fig. 7 restricts this to the standard case (9) for simplicity.)

Analogously to Overdrive LowGear, parties first generate their shares for $\boldsymbol{a}$ and $\boldsymbol{f}$. Here, only one of them (i.e., $\boldsymbol{a}$) requires ZKPs that prove correctness of encrypted shares. These shares are sent to all parties. Then, the parties can multiply these ciphertexts with (packings of) their own share of $\boldsymbol{f}$ to obtain ciphertexts of pairwise

Multiply$(\mathsf{a}_i, \mathsf{b}_i, \langle \mathsf{b}_0 \rangle_{\mathsf{pk}_0}, \ldots, \langle \mathsf{b}_{n-1} \rangle_{\mathsf{pk}_{n-1}})$**:** Compute $[\mathsf{c}]_i$ such that $\mathsf{c} = (\sum_{j=0}^{n-1} \mathsf{a}_j) \cdot (\sum_{j=0}^{n-1} \mathsf{b}_j)$.

1. For $j \in [..n) \setminus \{i\}$ do the following (in parallel).
1.1. Sample an uniformly random $\mathsf{m}_{i,j} \in \mathcal{R}_p$.
1.2. Compute $\langle \mathsf{c}_{i,j} \rangle_{\mathsf{pk}_j} = \mathsf{a}_i \cdot \langle \mathsf{b}_j \rangle_{\mathsf{pk}_j} - \mathsf{enc}'_{\mathsf{pk}_j}(\mathsf{m}_{i,j})$ where enc' is encryption with large *drowning* noise (larger than normal encryption randomness; cf. Appendix A.1.1).
1.3. Send $\langle \mathsf{c}_{i,j} \rangle_{\mathsf{pk}_j}$ to P_j and receive $\langle \mathsf{c}_{j,i} \rangle_{\mathsf{pk}_i}$ in return.
1.4. Decrypt $\langle \mathsf{c}_{j,i} \rangle_{\mathsf{pk}_i}$ to $\mathsf{c}_{j,i}$ with $\mathsf{dec}_{\mathsf{sk}_i}$.
2. Compute $[\mathsf{c}]_i = \mathsf{a}_i \cdot \mathsf{b}_i + \sum_{j \neq i} (\mathsf{c}_{j,i} + \mathsf{m}_{i,j})$.

Figure 6: Utilities for the Linear Homomorphic Offline Phase Used in $\Pi_{\text{offline-LHE}}$ (cf. Fig. 7) at Party P_i

Protocol $\Pi_{\text{offline-LHE}}$

Triples(op, D, D')**:** Generate a triple for the bilinear map op.

1. Run **ZKP**$([..N) \setminus \mathrm{image}(\mathrm{mapi}(\mathrm{op}, D, D', \cdot)))$ to obtain $(\mathsf{b}_i, \langle \mathsf{b}_0 \rangle_{\mathsf{pk}_0}, \ldots, \langle \mathsf{b}_{n-1} \rangle_{\mathsf{pk}_{n-1}})$. Sample uniformly random $\mathsf{g}_i, \mathsf{g}'_i \in \mathcal{R}_p$ such that the k-th coefficients are zero with $k \in [..N) \setminus \mathrm{image}(\mathrm{mapf}(\mathrm{op}, D, D', \cdot))$. Define $[\boldsymbol{a}]_i := \mathrm{unpacki}(\mathrm{op}, D, D', \mathsf{b}_i)$, $[\boldsymbol{f}]_i := \mathrm{unpackf}(\mathrm{op}, D, D', \mathsf{g}_i)$, $[\boldsymbol{f'}]_i := \mathrm{unpackf}(\mathrm{op}, D, D', \mathsf{g}'_i)$.
2. Run **Multiply** with $\mathsf{g}_i, \mathsf{b}_i$, and the ciphertexts for b_j to obtain $[\mathsf{c}]_i$. Analogously, obtain $[\mathsf{c}']_i$ for $\mathsf{g}'_i, \mathsf{b}_i$, and the ciphertexts.
3. Run **Multiply** with b_i, $[\alpha]_i$, and the ciphertexts of the MAC key shares to obtain $[\alpha \cdot \mathsf{b}]_i$. Analogously, obtain $[\alpha \cdot \mathsf{g}]_i$, $[\alpha \cdot \mathsf{g}']_i$, $[\alpha \cdot \mathsf{c}]_i$, and $[\alpha \cdot \mathsf{c}']_i$. Unpack these shares to get $[\alpha \boldsymbol{a}]_i$, $[\alpha \boldsymbol{f}]_i$, $[\alpha \boldsymbol{f'}]_i$, $[\alpha \boldsymbol{c}]_i$, and $[\alpha \boldsymbol{c'}]_i$, respectively.
4. Return **Sacrifice**$(\mathrm{op}, [\![\boldsymbol{a}]\!]_i, [\![\boldsymbol{f}]\!]_i, [\![\boldsymbol{f'}]\!]_i, [\![\boldsymbol{c}]\!]_i, [\![\boldsymbol{c'}]\!]_i)$.

Figure 7: Protocol for the Linear Homomorphic Offline Phase at Party P_i

shares. These pairwise shares are also re-randomized and sent back to the party that originally sent the encrypted share and holds the corresponding private key. After receiving all encrypted pairwise shares, this party can decrypt them and combine them to obtain a share of the overall product of packings, e.g., an encoding of the convolution of $\boldsymbol{a}$ and $\boldsymbol{f}$. Finally, all shares are authenticated (by multiplying with encrypted MAC key shares as in LowGear) and parts of the triples can be sacrificed to guarantee the correct relation between authenticated triples (cf. Section 6.4).

Note that this construction is much closer to LowGear than, for example, [25]. In [25], a protocol similar to our **Multiply** subprotocol (cf. Fig. 6) is used. However, their version does not drown the ciphertext containing the pairwise product $\mathsf{c}_{i,j}$. Instead, [25] computes this product and extracts (LWE) ciphertexts for all coefficients of the product's (RLWE) ciphertext that are later required for the shares of the conv2d result. We opted to not follow this approach for the following reasons. (i) We use larger BGV parameters for drowning ciphertexts for (scalar) Beaver triple generation, so avoiding drowning does not improve the parameters. (ii) The technique comes with additional computational overhead. (iii) It is unclear if maliciously crafted (LWE) ciphertexts might break the security as [25] only considered semi-honest adversaries. (iv) The technique could not be reproduced since the reference [13] pointed to in [25] does not discuss how to obtain LWE ciphertexts from RLWE ciphertexts (only vice versa). (v) The noise hiding technique of [25] is not well suited for our protocol, since it introduces a (probabilistic) 1 bit error in the result.

The following theorem captures the security of our LHE-based offline phase. A security proof can be found in Appendix F. To follow the security proofs in [31], the functionalities $\mathcal{F}_{\text{auth-linear}}$ (for linear operations on shares) and $\mathcal{F}_{\text{auth-MPC}}$ (for linear operations and triple generation) are used instead of a more traditional offline functionality $\mathcal{F}_{\text{offline}}$. Additionally, we assume standard functionalities for sampling random values ($\mathcal{F}_{\text{rand}}$), committing and decommitting to values ($\mathcal{F}_{\text{commit}}$), and generating encryption keys and shares of the MAC key ($\mathcal{F}_{\text{setup}}$).

Theorem 6.1. *The offline protocol $\Pi_{\text{offline-LHE}}$ securely implements the ideal functionality $\mathcal{F}_{\text{auth-MPC}}$ in the $(\mathcal{F}_{\text{auth-linear}}, \mathcal{F}_{\text{commit}}, \mathcal{F}_{\text{rand}}, \mathcal{F}_{\text{setup}})$-hybrid model with rewinding if the used BGV cryptosystem achieves enhanced CPA-security [31].*

Remark 6.1. Please note that the use of rewinding is a standard tool in these type of protocols (cf. LowGear protocol [31]).

6.3 Somewhat Homomorphic Offline Phase

In Appendix F (Fig. 19), we present a (convolution) triple generation based on somewhat homomorphic encryption. The construction is based on Overdrive's HighGear protocol [31].

Similarly to the linear homomorphic case (cf. Section 6.2), all parties sample their own shares of $\boldsymbol{a}$ and $\boldsymbol{f}$ and encrypt them. However, in the SPDZ-like SHE approach, the share for both $\boldsymbol{a}$ and $\boldsymbol{f}$ are encrypted. Utilizing a HighGear/TopGear-style ZKP, the parties prove that the sum of their encrypted shares is a valid ciphertext of the sum of the shares, i.e., of the shared value $\boldsymbol{a}$ or $\boldsymbol{f}$. Therefore, all parties have a valid ciphertext of (the packing of) $\boldsymbol{a}$ and $\boldsymbol{f}$. These can be multiplied homomorphically with a somewhat homomorphic encryption scheme to obtain a ciphertext of the product, e.g., of the encoding of a convolution of $\boldsymbol{a}$ and $\boldsymbol{f}$. Analogously to the original approach by [18] the parties can (distributively) decrypt the product ciphertext, *reshare* the product and authenticate it. Finally, sacrificing is used to guarantee that the correlation of the triple is satisfied (cf. Section 6.4).

Please note, that again the main changes to the HighGear (or TopGear) protocol are the use of ZKPs that ensure correct packing, local (un)packing operations, and the adapted sacrificing for convolutions. The security of our SHE-based offline phase is given by the following theorem and the proof in Appendix F. Again, we assume the availability of standard functionality for (de)committing, randomness generation, and a key/MAC setup.

Theorem 6.2. *The offline protocol $\Pi_{\text{offline-SHE}}$ securely implements the ideal functionality $\mathcal{F}_{\text{offline}}$ in the $(\mathcal{F}_{\text{commit}}, \mathcal{F}_{\text{rand}}, \mathcal{F}_{\text{setup}})$-hybrid model if the used BGV cryptosystem achieves CPA-security and has an algorithm for meaningless public key generation [18].*

6.4 Sacrificing

While Chen et al. presented a generalization of the Beaver multiplication approach for arbitrary bilinear operations in [14], they

did not generalize the sacrificing in the same way.[4] As described in Section 6.1, sacrificing is necessary in our protocols to ensure that the produced triple is correctly authenticated. In Fig. 5, we show a generalization of the sacrificing presented in [30]. Its security follows directly from [30]. However, the efficiency of the sacrificing can greatly depend on the type of bilinear operation that we consider. The reason for this is the inherent asymmetry of the optimized sacrificing of [30] (compared to the original technique used in [18]). This is especially true for LowGear-style protocols that only require expensive ZKPs for one of the triple elements.

In general, one of the triple inputs (i.e., $\boldsymbol{a}$ or $\boldsymbol{f}$ of a triple $(\boldsymbol{a}, \boldsymbol{f}, \boldsymbol{c})$) might be more expensive to compute. Therefore, one should consider a reversed version of the sacrificing presented in Fig. 5 taking shares of $\boldsymbol{a}, \boldsymbol{a'}, \boldsymbol{f}, \boldsymbol{c}, \boldsymbol{c'}$ instead. Technically, this can be achieved by using $(\mathrm{op}'(\boldsymbol{y}, \boldsymbol{x}) = \mathrm{op}(\boldsymbol{x}, \boldsymbol{y}), [\boldsymbol{f}]_i, [\boldsymbol{a}]_i, [\boldsymbol{a'}]_i, [\boldsymbol{c}]_i, [\boldsymbol{c'}]_i)$ as inputs to **Sacrifice**.

6.5 Modifications and Optimizations

While the above MPC protocols are very general (being able to compute triples for any bilinear function that can be represented with the standard case (9) of the framework of Section 3.1), small modification can be used to also support the non-standard bilinear forms $\mathrm{op}_{\mathcal{R}}$ in Eq. (8) (e.g., the simple packing of Section 4.1; cf. Section 6.5.1), handle any size of convolution (cf. Section 6.5.2), utilize ciphertexts more efficiently (cf. Section 6.5.3), or handle convolutions with strides larger than 1 and/or non-zero padding (cf. Section 6.5.4).

6.5.1 Modification for the Simple Convolution Packing. In this paragraph we discuss how packing images and filters in multiple ciphertexts (as in Section 4.1) is handled. The overall result then is a sum of several homomorphic ciphertext products. Extending the protocols of Sections 6.2 and 6.3 is straightforward. To see that these extended protocols are still secure, notice that the intermediate steps only produce shares of intermediate results (as well as ciphertexts that do not leak any information as they are either blinded in the LHE protocol or locally computed in the SHE protocol). These intermediate shares are summed up to obtain the overall triple. Security of the extended proof then directly follows from the security results from Sections 6.2 and 6.3 and the properties of the secret sharing scheme.

6.5.2 Handling Large Convolutions. Recall that in Sections 3.1 and 4, we usually had to choose N large enough to support packing of all tensor dimensions, e.g., $b \cdot d' \cdot h'' \cdot w'' \leq N$ with the simple packing of Section 4.1 or when $b \cdot d' \cdot d \cdot h'' \cdot w'' \leq N$ with the generalization of Huang et al.'s packing (cf. Section 4.2). The choice of N on the other hand affects other parameters, e.g., of the encryption scheme, and can slow down the offline phase significantly if N gets to big. To avoid this blow-up of N and possible parameter changes to the encryption scheme, we split large convolutions into smaller ones and thereby extend the approach by [25] from the passively secure setting to the actively secure setting. While splitting along the batch dimension (b) or output depth dimension (d') is straightforward (even in the actively secure setting), splitting convolutions along the spatial dimensions or the input depth dimension (d) often lead to an overhead and should then be avoided. The technical reason for this behavior are the ciphertext sums in these dimensions that come with our packing methods and convolution protocols. For *irregular splittings*, i.e., summands of different dimensions (e.g., splitting $d = 11$ in parts with $d = 6$ and $d = 5$), we can then no longer use the full amortization potential of the BGV scheme and the associated ZKPs, which we need in the actively secure setting. For example, in the worst case we need an additional ZKP for 40 ciphertexts for each single ciphertext that encodes a different dimension – hence ZKPs for 80 ciphertexts for the splitting of $d = 11$ in parts with $d = 6$ and $d = 5$. This large overhead can be reduced by trivially increasing the ciphertexts for small dimensions to a common larger dimension, i.e., use the same dimensions in each part and set certain parts of the ciphertexts to zero (in our example we get then twice $d = 6$). Nevertheless, a certain overhead due to the zero coefficients remains. We therefore preferably split on dimensions where these problems do not occur and apply irregular splittings only as a last resort.

6.5.3 Combining Ciphertexts for Sacrificing. Finally, we want to discuss an optimal use of the sacrificing technique in our setup. As mentioned in Section 6.4 our sacrificing protocol produces, similar to MASCOT [30], shares of tuples $(\boldsymbol{a}, \boldsymbol{f}, \boldsymbol{f'}, \boldsymbol{c}, \boldsymbol{c'})$ and then discards, i.e., sacrifices, $\boldsymbol{f'}$ and $\boldsymbol{c'}$ to check that $\boldsymbol{c} = \mathrm{conv2d}(\boldsymbol{a}, \boldsymbol{f})$. Now instead of generating $\boldsymbol{c}$ and $\boldsymbol{c'}$ separately, e.g., by using two invocations of **Multiply** in Fig. 7, we can generate them more efficiently by combining $\boldsymbol{f}$ and $\boldsymbol{f'}$ into a single large filter and then multiply only once to get both $\boldsymbol{c}$ and $\boldsymbol{c'}$. For example, we can encode a single convolution of a $d \times h \times w \times d$ image with a $2d' \times d \times h' \times w'$ filter (of twice the output depth dimension d') in the ciphertext multiplication. After the multiplication and unpacking, the share of the result (and of the filter) with doubled output depth can be simply split in half along the d' dimension to get a 5-tuple for sacrificing: one image, two filters, and the result of convolving the image with two filters. The analogous doubling technique can also be applied to the batch dimension (b; for conv2d or dconv2d).

Please note that this optimization is orthogonal to the splitting in Section 6.5.2. We use both optimizations in our implementation.

6.5.4 Special Convolutions. For (depthwise) convolutions with non-zero padding, e.g., when one expands the (blue) image in Fig. 1 with non-zero values (usually constant values or replicas of the border pixels), or convolutions with strides of 2 or more, we do not offer special constructions with our protocols. This is because the used packing methods that homomorphically compute negacyclic convolutions require zero-padding so the constructions are correct (cf. Appendix D) and compute all pixels of the result (i.e., with a stride of 1). These convolutions can still be computed by expressing them as a (larger) convolution with zero-padding or by discarding parts of the result, respectively. Note that the conv1@7x7 convolution discussed in Section 7 has a stride of 2 and our protocols outperform the related work, even though our protocol discards parts of the result.

[4] They instead switched to a encryption scheme that allows for additional ciphertext-ciphertext multiplications and thus no sacrificing is required in [14].

7 IMPLEMENTATION AND EVALUATION

We have implemented our protocols [47] on top of MP-SPDZ [29] by adding support for secure convolutions and depthwise convolutions. Our implementation extends the online phase with convolution tuples for faster convolutions, as well as the corresponding convolution triple generation with both LowGear-style and HighGear-style protocols in the offline phase. The implementation is fully-featured as we can use the wide range of other (non-convolution) operations that are already part of MP-SPDZ.

In the remainder of this section, we show the results for the empirical evaluation of the protocols developed in this work. We evaluate our technique for convolutions with images and filters of typical shapes. We use ResNet50 as a reference for this. Additionally, we benchmark depthwise convolutions for images of different sizes to show the benefit of our specialized handling of depthwise convolutions. Note that our protocols do not affect the accuracy of ML models. The accuracy stays the same as, e.g., in [16, 32], who perform secure inference or training on MPC. Therefore we here do not measure accuracy as part of the evaluation.

Evaluation Setup. We ran the benchmarks on a virtual server (AMD EPYC™ 7443 processor @ 2.85 GHz, 4 to 8 cores) emulating different network settings: LAN with 10 ms network delay and 1 Gbit/s network bandwidth; and WAN with 35 ms delay and 320 Mbit/s. These network settings allow us to compare our results to the state-of-the-art way of computing convolutions as matrix multiplications [14].[5] Our benchmarks utilize only a single thread per party for computations. The benchmarks use $n = 2$ parties for LowGear-style (LHE-based) offline phases (on 4 cores) and $n = 4$ parties for HighGear-style (SHE-based) offline phases (on 8 cores). We benchmark our protocol in the same setting as [14] for SPDZ-like protocols: 128 bit of computational security, 40 bit of statistical security, and plaintext modulus of 128 bit. In the following, we analyze the performance of our protocol in the online phase and in the offline phase.

Runtime in the Offline Phase for Convolutions. In Table 2, we compare the runtime of the classical SPDZ-based MPC computation with field multiplications (LowGear) and [14] to our implementation (simple packing and generalization of Huang et al.'s packing). These benchmarks are for all non-1×1 convolutions in ResNet50 [23], shown separately for each group of layers. The simple packing performs best overall, while the generalization of Huang et al.'s packing is a close second place. The simple packing is clearly superior to the LowGear protocol without any optimizations for convolutions (LAN: 27.52× faster; WAN: 39.66× faster) but also to the matrix multiplication of [14] (LAN: 4.82× faster; WAN: 3.01× faster). The results for [14] show the least increase in runtime when the network gets more limited (comparing LAN to WAN) but the computational overhead of the HE operations used in their offline phase are still too costly to outperform our protocol in the WAN setting. If we compare our results to the protocols in the semi-honest setting (e.g., Huang et al. [25] perform the ResNet50 conv1@7x7 convolution in around 2 s with a smaller plaintext modulus in the WAN setting without any online-offline separation), we can see that there is still a large gap in the performance between actively and passively secure protocols. However, using our convolution packings noticeably improves upon the state-of-the-art in our actively secure setting.

Comparing our protocol's HighGear variant to HighGear shows a 13.43× speed-up for the simple packing and a 12.23× speed-up for the generalization of Huang et al.'s packing. This was measured for the 4-party WAN setting; detailed results can be found in Appendix G. Chen et al. do not evaluate the runtime for their protocol [14] with more than two parties.

Communication Cost for Convolution Triple Generation. For the above computation of ResNet50 convolutions, each party needs to send 2.187 TB of data for the LowGear protocol, 138.672 GB for our protocol with the simple packing, and 134.635 GB with our generalization of Huang et al.'s packing, respectively, in the offline phase. We estimate Chen et al.'s [14] communication cost to be 21.020 GB. As we see above, the low communication cost of [14] does not translate to a faster protocol as we clearly outperform theirs in the evaluated setting. This shows that we can successfully trade communication cost for faster protocols by avoiding expensive ciphertext rotations with our packing methods. A more detailed analysis of the communication costs can be found in Appendix G.

Table 2: Runtime Results for conv2d **Operations in the Offline Phase (in Seconds).** Our protocols here are LowGear-based (cf. Section 6.2). Runtime is given for convolutions of ResNet50 [23]. The layer conv1@7x7 is a convolution of a $1 \times 224 \times 224 \times 3$ image with a $64 \times 3 \times 7 \times 7$ filter and stride 2. Other layers conv$i$@3x3 are for $1 \times 7 \cdot 2^{5-i} \times 7 \cdot 2^{5-i} \times 2^{4+i}$ images, $2^{4+i} \times 2^{4+i} \times 3 \times 3$ filters, and stride 1. The above convolutions are repeated c times in layer conv$i$. We give the runtime for all c convolutions of a layer.

c Layer	LowGear [31]	Matmul [14][a]	Our Simple Packing	Our General. Huang et al. Packing
	2 Party LAN Setting			
1 conv1@7x7	21499[b]	7014	382[b]	384[b]
3 conv2@3x3	63246[c]	13420	1913	1914
4 conv3@3x3	83255[c]	9016	2409	2615
6 conv4@3x3	119903[c]	15459	4184	3962
3 conv5@3x3	56576[c]	15459	3631[b]	4008[b]
Total	**344478**	**60368**	**12519**	**12882**
	2 Party WAN Setting			
1 conv1@7x7	53264[b]	7408	653[b]	649[b]
3 conv2@3x3	154090[c]	14175	3303	3287
4 conv3@3x3	202938[c]	9523	4095	4432
6 conv4@3x3	291982[c]	16327	7051	6661
3 conv5@3x3	139118[c]	16331	6112[b]	6732[b]
Total	**841392**	**63764**	**21214**	**21761**

[a] column extrapolated from the runtime results in [14] using Tables 4 and 5
[b] extrapolated from results with halved output depth
[c] extrapolated from results with output depth $d' = 2$ and $c = 1$

[5] Chen et al. seem to only provide an implementation for homomorphic matrix operations – not a full protocol: https://github.com/snwagh/ponytail-public as of 2022-11-03. Therefore, we compare our results to the numbers given in their paper [14]. We emulate their network settings exactly – with the exception of the LAN setting where we use a reduced bandwidth of 1 Gbit/s instead of 5 Gbit/s to model a more realistic setting. Additionally, we use a slightly faster CPU (2.85 GHz instead of 2.7 GHz).

Table 3: Runtime Results for dconv2d **Operations in the Offline Phase (in Seconds).** Our protocols here are LowGear-based (cf. Section 6.2). Runtime is given for $1 \times h \times h \times 512$ images and $512 \times 3 \times 3$ filters.

h	LowGear [31]	Matmul [14][a]	Our Simple Packing	Our Gen. Huang et al. Packing	Our Depthw. Packing
	2 Party LAN Setting				
7	37	5153	321	322	55
25	504	25767	323	322	165
50[b]	2137	103068	352	352	343
120[b]	12279	582335	804	809	1086
240[b]	49398	2319034	2657	2667	3704
	2 Party WAN Setting				
7	90	5444	637	636	121
25	1244	27223	637	637	327
50[b]	5155	108892	692	688	664
120[b]	29764	615241	1443	1438	1907
240[b]	119842	2450076	4520	4516	6227

[a] column extrapolated from the runtime results in [14]
[b] row extrapolated from results with depth $d = 32$ (except for matmul runtime)

Round Complexity in the Offline Phase. Also note that the (theoretical) round complexity of the protocols is almost the same. Not considering the setup (key and MAC generation, etc.), the triple generation requires 4 rounds for [14], 6 rounds for LowGear-style protocols (ours and [31]), and 8 rounds for HighGear-style protocols (ours and [3, 31]).

Runtime in the Offline Phase for Depthwise Convolutions. We also benchmarked depthwise convolutions. The results are depicted in Table 3. For dconv2d, filter sizes of 3×3 are standard [2, 15, 24, 48]. Therefore, we benchmark these for different image sizes. We fix the depth to 512 due to the separable nature of dconv2d, i.e., each entry along the depth dimension is independent and thus the runtime scales linearly with the depth. Runtime for other values of d can simply be extrapolated from our results.

As can be seen in Table 3, the matrix-based approach of [14] is unsuitable for depthwise convolutions and performs worse than the standard LowGear protocol. This is because [14] would compute matrix multiplications with the same size as for a conv2d computation (with input and output depth set to 512 in this example), incurring the overhead of the non-optimal emulation of convolutions with matrix multiplications and the overhead of the mismatch between the 128×128 matrices computed by [14] and the matrices needed to compute convolutions. Note that this still performs better than computing a single matrix multiplication for each of the output channels.

In contrast, we can use our depthwise packing of Section 4.3 which performs well for images of size 50 and below, or compute conv2d operations to emulate dconv2d (with the simple packing or the generalized Huang et al. packing) which performs well for larger images (larger than size 50). The conv2d packings compute only one output channel for polynomial multiplication and are therefore slower if we could instead compute multiple channels with the depthwise packing. If the image size grows, the depthwise packing would also only compute only one channel per convolution and then our implementation of the conv2d packings utilize the optimization in Section 6.5.2 better to compute a few (partial) convolutions per output channel.

Overall, the right choice of one of our packing schemes can outperform LowGear for all but the smallest image sizes (LAN: up to $18.59\times$ faster with $h = 240$; WAN: up to $26.53\times$ faster with $h = 240$) and all of them outperform [14]. We also tested Bian et al.'s packing scheme (see Appendix G). First tests show considerably worse performance compared to LowGear ($\approx 100\times$ slower). The main reason for this inefficiency is the computational overhead of the modified BGV scheme that we employ for this packing (cf. Appendix A.2) and the increase in communication from the new type of ciphertexts.

Runtime in the Online Phase. In the online phase, we compare our approach (using convolution triples) to the standard SPDZ protocol (the distinction between LowGear and HighGear is only meaningful for the offline phase) and the use of matrix triples to emulate convolutions (as done in [14]). Note that for matrix triples, we assume that matrix triples of any shape are already precomputed. This is the optimal setting for the matrix-based approach and strictly better than [14] which only produces matrices that are a multiple of 128×128 in size. For the same layers as in Table 2, our approach with convolution triples clearly outperforms the SPDZ online phase (LAN: $16.39\times$ faster; WAN: $27.21\times$ faster) and also the approach based on matrix triples (LAN: 8 % faster; WAN: 12 % faster). The detailed results can be found in Appendix G.

For depthwise convolutions, our advantage of specialized convolution triples is even more pronounced (in certain cases) compared to SPDZ (LAN: $19.41\times$ faster on average for $h \in \{7, 25, 50, 120, 240\}$ and $41.84\times$ faster for $h = 7$; WAN: $20.14\times$ faster on average and $42.58\times$ faster for $h = 7$) and also compared to matrix triples (LAN: $13.51\times$ faster on average and $40.15\times$ for $h = 7$; WAN: $15.70\times$ faster on average and $41.84\times$ for $h = 7$). Hence, we observe a considerable speed-up for small images (due to the better communication complexity) that gets smaller as the image size (and computational complexity) increases. However, even for large images of size 240, our advantage is $3.87\times$ (LAN) to $5.33\times$ (WAN) compared to matrix triples.

Storage Cost for Convolutions. To run the above-mentioned convolutions in the online phase, SPDZ requires storage for 188.899 GB of Beaver triples. Chen et al. would have to store 2.653 GB of 128×128 matrix triples. Our convolution triples require 572 MB.

In summary, our evaluation shows that our implementation significantly outperforms current actively secure state-of-the-art protocols for convolution and convolution-based ML tasks.

ACKNOWLEDGMENTS

Marc Rivinius, Pascal Reisert, and Ralf Küsters were supported by the CRYPTECS project. The CRYPTECS project has received funding from the German Federal Ministry of Education and Research under Grant Agreement No. 16KIS1441 and from the French

National Research Agency under Grant Agreement No. ANR-20-CYAL-0006. Sebastian Hasler was supported by Advantest as part of the Graduate School "Intelligent Methods for Test and Reliability" (GS-IMTR) at the University of Stuttgart. The authors also acknowledge support by the state of Baden-Württemberg through bwHPC.

We thank our anonymous reviewers and our shepherd for their invaluable feedback. We also thank Andrés Bruhn and Azin Jahedi from the Institute for Visualization and Interactive Systems at the University of Stuttgart for providing the computational resources and assistance with running our experiments.

REFERENCES

[1] Ehud Aharoni, Allon Adir, Moran Baruch, Nir Drucker, Gilad Ezov, Ariel Farkash, Lev Greenberg, Ramy Masalha, Guy Moshkowich, Dov Murik, Hayim Shaul, and Omri Soceanu. 2023. HeLayers: A Tile Tensors Framework for Large Neural Networks on Encrypted Data. *Proc. Priv. Enhancing Technol.* 2023, 1 (2023), 325–342.

[2] Alaaeldin Ali, Hugo Touvron, Mathilde Caron, Piotr Bojanowski, Matthijs Douze, Armand Joulin, Ivan Laptev, Natalia Neverova, Gabriel Synnaeve, Jakob Verbeek, and Hervé Jégou. 2021. XCiT: Cross-Covariance Image Transformers. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*. 20014–20027.

[3] Carsten Baum, Daniele Cozzo, and Nigel P. Smart. 2019. Using TopGear in Overdrive: A More Efficient ZKPoK for SPDZ. In *Selected Areas in Cryptography - SAC 2019 - 26th International Conference, Waterloo, ON, Canada, August 12-16, 2019, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 11959)*. Springer, 274–302.

[4] Donald Beaver. 1991. Efficient Multiparty Protocols Using Circuit Randomization. In *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings (Lecture Notes in Computer Science, Vol. 576)*. Springer, 420–432.

[5] Ayoub Benaissa, Bilal Retiat, Bogdan Cebere, and Alaa Eddine Belfedhal. 2021. TenSEAL: A Library for Encrypted Tensor Operations Using Homomorphic Encryption. *CoRR* abs/2104.03152 (2021). arXiv:2104.03152

[6] Song Bian, Dur-e-Shahwar Kundi, Kazuma Hirozawa, Weiqiang Liu, and Takashi Sato. 2021. APAS: Application-Specific Accelerators for RLWE-Based Homomorphic Linear Transformations. *IEEE Trans. Inf. Forensics Secur.* 16 (2021), 4663–4678.

[7] Zvika Brakerski. 2012. Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP. In *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7417)*. Springer, 868–886.

[8] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2012. (Leveled) fully homomorphic encryption without bootstrapping. In *Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8-10, 2012*. ACM, 309–325.

[9] Alon Brutzkus, Ran Gilad-Bachrach, and Oren Elisha. 2019. Low Latency Privacy Preserving Inference. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA (Proceedings of Machine Learning Research, Vol. 97)*. PMLR, 812–821.

[10] José Cabrero-Holgueras and Sergio Pastrana. 2021. SoK: Privacy-Preserving Computation Techniques for Deep Learning. *Proceedings on Privacy Enhancing Technologies* 2021 (10 2021), 139–162.

[11] Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. 2020. End-to-End Object Detection with Transformers. In *Computer Vision - ECCV 2020 - 16th European Conference, Glasgow, UK, August 23-28, 2020, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12346)*. Springer, 213–229.

[12] Octavian Catrina and Sebastiaan de Hoogh. 2010. Improved Primitives for Secure Multiparty Integer Computation. In *Security and Cryptography for Networks, 7th International Conference, SCN 2010, Amalfi, Italy, September 13-15, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6280)*. Springer, 182–199.

[13] Hao Chen, Wei Dai, Miran Kim, and Yongsoo Song. 2021. Efficient Homomorphic Conversion Between (Ring) LWE Ciphertexts. In *Applied Cryptography and Network Security - 19th International Conference, ACNS 2021, Kamakura, Japan, June 21-24, 2021, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12726)*. Springer, 460–479.

[14] Hao Chen, Miran Kim, Ilya P. Razenshteyn, Dragos Rotaru, Yongsoo Song, and Sameer Wagh. 2020. Maliciously Secure Matrix Multiplication with Applications to Private Deep Learning. In *Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part III (Lecture Notes in Computer Science, Vol. 12493)*. Springer, 31–59.

[15] François Chollet. 2017. Xception: Deep Learning with Depthwise Separable Convolutions. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*. IEEE Computer Society, 1800–1807.

[16] Anders P. K. Dalskov, Daniel Escudero, and Marcel Keller. 2020. Secure Evaluation of Quantized Neural Networks. *Proc. Priv. Enhancing Technol.* 2020, 4 (2020), 355–375.

[17] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. 2013. Practical Covertly Secure MPC for Dishonest Majority - Or: Breaking the SPDZ Limits. In *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8134)*. Springer, 1–18.

[18] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. 2012. Multiparty Computation from Somewhat Homomorphic Encryption. In *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7417)*. Springer, 643–662.

[19] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin E. Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. 2019. CHET: an optimizing compiler for fully-homomorphic neural-network inferencing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. ACM, 142–156.

[20] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. 2021. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

[21] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. 2020. Improved Primitives for MPC over Mixed Arithmetic-Binary Circuits. In *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12171)*. Springer, 823–852.

[22] Junfeng Fan and Frederik Vercauteren. 2012. *Somewhat Practical Fully Homomorphic Encryption.* Technical Report 2012/144. Cryptology ePrint Archive.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 770–778.

[24] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR* abs/1704.04861 (2017). arXiv:1704.04861

[25] Zhicong Huang, Wen-jie Lu, Cheng Hong, and Jiansheng Ding. 2022. Cheetah: Lean and Fast Secure Two-Party Deep Neural Network Inference. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*. USENIX Association, 809–826.

[26] Xiaoqian Jiang, Miran Kim, Kristin E. Lauter, and Yongsoo Song. 2018. Secure Outsourced Matrix Computation and Application to Neural Networks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. ACM, 1209–1222.

[27] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha P. Chandrakasan. 2018. GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. USENIX Association, 1651–1669.

[28] Andrej Karpathy and Li Fei-Fei. 2015. Deep visual-semantic alignments for generating image descriptions. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*. IEEE Computer Society, 3128–3137.

[29] Marcel Keller. 2020. MP-SPDZ: A Versatile Framework for Multi-Party Computation. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*. ACM, 1575–1590.

[30] Marcel Keller, Emmanuela Orsini, and Peter Scholl. 2016. MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. ACM, 830–842.

[31] Marcel Keller, Valerio Pastro, and Dragos Rotaru. 2018. Overdrive: Making SPDZ Great Again. In *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part III (Lecture Notes in Computer Science, Vol. 10822)*. Springer, 158–189.

[32] Marcel Keller and Ke Sun. 2022. Secure Quantized Training for Deep Learning. In *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA (Proceedings of Machine Learning Research, Vol. 162)*. PMLR, 10912–10938.

[33] Miran Kim, Xiaoqian Jiang, Kristin E. Lauter, Elkhan Ismayilzada, and Shayan Shams. 2021. HEAR: Human Action Recognition via Neural Networks on Homomorphically Encrypted Data. *CoRR* abs/2104.09164 (2021). arXiv:2104.09164

[34] Brian Knott, Shobha Venkataraman, Awni Y. Hannun, Shubho Sengupta, Mark Ibrahim, and Laurens van der Maaten. 2021. CrypTen: Secure Multi-Party Computation Meets Machine Learning. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*. 4961–4973.

[35] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*. 1106–1114.

[36] Yann LeCun, Bernhard E. Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne E. Hubbard, and Lawrence D. Jackel. 1989. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Comput.* 1, 4 (1989), 541–551.

[37] Eunsang Lee, Joon-Woo Lee, Junghyun Lee, Young-Sik Kim, Yongjune Kim, Jong-Seon No, and Woosuk Choi. 2022. Low-Complexity Deep Convolutional Neural Networks on Fully Homomorphic Encryption Using Multiplexed Parallel Convolutions. In *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA (Proceedings of Machine Learning Research, Vol. 162)*. PMLR, 12403–12422.

[38] Ming Liang and Xiaolin Hu. 2015. Recurrent convolutional neural network for object recognition. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*. IEEE Computer Society, 3367–3375.

[39] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. 2017. Oblivious Neural Network Predictions via MiniONN Transformations. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. ACM, 619–631.

[40] Tianyi Liu, Xiang Xie, and Yupeng Zhang. 2021. zkCNN: Zero Knowledge Proofs for Convolutional Neural Network Predictions and Accuracy. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. ACM, 2968–2985.

[41] Qian Lou and Lei Jiang. 2021. HEMET: A Homomorphic-Encryption-Friendly Privacy-Preserving Mobile Neural Network Architecture. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event (Proceedings of Machine Learning Research, Vol. 139)*. PMLR, 7102–7110.

[42] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. 2020. Delphi: A Cryptographic Inference Service for Neural Networks. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. USENIX Association, 2505–2522.

[43] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 19–38.

[44] Maxime Oquab, Léon Bottou, Ivan Laptev, and Josef Sivic. 2014. Learning and Transferring Mid-level Image Representations Using Convolutional Neural Networks. In *2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2014, Columbus, OH, USA, June 23-28, 2014*. IEEE Computer Society, 1717–1724.

[45] Emmanuela Orsini. 2020. Efficient, Actively Secure MPC with a Dishonest Majority: A Survey. In *Arithmetic of Finite Fields - 8th International Workshop, WAIFI 2020, Rennes, France, July 6-8, 2020, Revised Selected and Invited Papers (Lecture Notes in Computer Science, Vol. 12542)*. Springer, 42–71.

[46] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. 2020. CrypTFlow2: Practical 2-Party Secure Inference. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*. ACM, 325–342.

[47] Marc Rivinius, Pascal Reisert, Sebastian Hasler, and Ralf Küsters. 2023. Convolutions in Overdrive: Maliciously Secure Convolutions for MPC (Implementation). https://github.com/sec-stuttgart/MP-SPDZ-convolution-triples.

[48] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. Computer Vision Foundation / IEEE Computer Society, 4510–4520.

[49] Sijun Tan, Brian Knott, Yuan Tian, and David J. Wu. 2021. CryptGPU: Fast Privacy-Preserving Machine Learning on the GPU. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 1021–1038.

[50] Zachary Teed and Jia Deng. 2020. RAFT: Recurrent All-Pairs Field Transforms for Optical Flow. In *Computer Vision - ECCV 2020 - 16th European Conference, Glasgow, UK, August 23-28, 2020, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12347)*. Springer, 402–419.

[51] Yifan Tian, Laurent Njilla, Jiawei Yuan, and Shucheng Yu. 2021. Low-Latency Privacy-Preserving Outsourcing of Deep Neural Network Inference. *IEEE Internet Things J.* 8, 5 (2021), 3300–3309.

[52] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. 5998–6008.

[53] Sergey Zagoruyko and Nikos Komodakis. 2015. Learning to compare image patches via convolutional neural networks. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*. IEEE Computer Society, 4353–4361.

[54] Qiao Zhang, Chunsheng Xin, and Hongyi Wu. 2021. GALA: Greedy ComputAtion for Linear Algebra in Privacy-Preserved Neural Networks. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society.

A PRELIMINARIES (CONTINUED)

Here, we give additional details to the preliminaries outlined in Section 2. This includes, for example, common MPC subprotocols and details on BGV, as well as the modification necessary to use BGV with Bian et al.'s packing [6].

A.1 Homomorphic Encryption and BGV (Continued)

Here, we present a more detailed description of the BGV encryption scheme [8]. Some aspects are discussed only on a conceptual level as the details are less relevant for this work. An interested reader can find all details in [8, 17].

First, we present the necessary distributions that values are sampled from (cf., e.g., [17, 31]).

- HW_h^N: Outputs a vector of length N with elements chosen from $\{-1, 0, 1\}$. Exactly $h \leq N$ elements are chosen to be non-zero (uniformly random from $\{-1, 1\}$). The others are zero. h is chosen based on the target security level, e.g., $h = 64 + \mathrm{sec}$ for statistical security parameter sec in [31].
- ZO^N: Outputs a vector of length N with elements chosen from $\{-1, 0, 1\}$. Elements are zero with probability $1/2$ and -1 or 1 with probability $1/4$ each.
- DG_σ^N: Outputs a vector of length N where each element is sampled from a discrete Gaussian distribution with variance σ^2. $\sigma = 3.2$ is a common choice [17, 31].
- $\mathrm{RC}_\sigma^N = \mathrm{ZO}^N \times \mathrm{DG}_\sigma^N \times \mathrm{DG}_\sigma^N$: Outputs encryption randomness for BGV, i.e., three vectors sampled from the distributions (described above).
- UB_B^N: Outputs a vector of length N where each element is sampled uniformly at random from $[-B..B)$.

In the context of BGV, the resulting N-vectors are interpreted as $\mathcal{R}_q$ elements. For this, the output vector is used as coefficient vector in the polynomial ring (and reduced modulo q).

Let $\mathsf{sk} = \mathsf{s}$ be the BGV private key. s is sampled from HW_h^N. Then, $\mathsf{pk} = (\mathsf{a}, \mathsf{b})$ is the corresponding public key for a uniformly random $\mathsf{a} \in \mathcal{R}_q$ and $\mathsf{b} = \mathsf{a} \cdot \mathsf{s} + p \cdot \mathsf{e}$ where e is sampled from DG_σ^N. Encryption is performed with randomness $\mathbf{r} = (\mathsf{u}, \mathsf{v}, \mathsf{w})$ sampled with RC_σ^N, i.e., the encryption of $\mathsf{x} \in \mathcal{R}_p$ is

$$\mathrm{enc}_{\mathsf{pk}}(\mathsf{x}, \mathbf{r}) = (\mathsf{b} \cdot \mathsf{u} + p \cdot \mathsf{v} + \mathsf{x}, \mathsf{a} \cdot \mathsf{u} + p \cdot \mathsf{w}). \tag{15}$$

The corresponding decryption is

$$\mathrm{dec}_{\mathsf{sk}}(\langle \mathsf{x} \rangle) = (\langle \mathsf{x} \rangle[0] - \langle \mathsf{x} \rangle[1] \cdot \mathsf{s}) \bmod p. \tag{16}$$

We also make use of the following ciphertext operations. Let $\mathsf{x}, \mathsf{y} \in \mathcal{R}_p$ and $\mathbf{r}, \mathbf{r'} \in \mathcal{R}_q^3$ be valid encryption randomness, the BGV scheme

(for suitable p, q) has the following homomorphic properties:

$$\mathrm{dec}_{\mathrm{sk}}(\mathrm{enc}_{\mathrm{pk}}(x, \mathbf{r}) + \mathrm{enc}_{\mathrm{pk}}(y, \mathbf{r'})) = x + y \tag{17}$$
$$\mathrm{dec}_{\mathrm{sk}}(\mathrm{enc}_{\mathrm{pk}}(x, \mathbf{r}) + y) = x + y \tag{18}$$
$$\mathrm{dec}_{\mathrm{sk}}(\mathrm{enc}_{\mathrm{pk}}(x, \mathbf{r}) \cdot \mathrm{enc}_{\mathrm{pk}}(y, \mathbf{r'})) = x \cdot y \tag{19}$$
$$\mathrm{dec}_{\mathrm{sk}}(x \cdot \mathrm{enc}_{\mathrm{pk}}(y, \mathbf{r'})) = x \cdot y. \tag{20}$$

Note that the $x + y$ and $x \cdot y$ are additions and multiplications of polynomials, where coefficients are additionally taken modulo p. We abuse the notation to also write $+$ and $\cdot$ for operations on ciphertexts but these can be more complex (especially ciphertext-ciphertext multiplication; see below).

Also note that there is an isomorphism $\mathrm{CRT} : \mathcal{R}_p \to \bigtimes_{k=0}^{N-1} \mathbb{F}_p$ (based on the Chinese remainder theorem) for the plaintext space. In particular,

$$\mathrm{CRT}(x \cdot y) = \mathrm{CRT}(x) \odot \mathrm{CRT}(y), \tag{21}$$

where $\odot$ is the component-wise multiplication. We remark that using (21), a single ciphertext-ciphertext multiplication represents N underlying field multiplications in $\mathbb{F}_p$. This is used in most SPDZ-like protocols since [18].

Some MPC protocols use *modulus switching* and *key switching* in the ciphertext-ciphertext multiplication (e.g., [17, 31]), i.e., the multiplication of two ciphertexts in $C = \mathcal{R}_q^2$ yields a ciphertext in $C' = \mathcal{R}_{q'}^2$ that can be decrypted just as before. Note that (16) takes $\langle x \rangle \in C$ as input and all operations before the reduction modulo p are modulo q. For ciphertexts after modulus and key switching, $\mathrm{dec}'_{\mathrm{sk}} : C' \to \mathcal{R}_p$ should be used where the operations are the same as in (16) but modulo q' before the modulo p reduction. For simplicity, we simply write $\mathrm{dec}_{\mathrm{sk}}$ also for this decryption operation. Details on the ciphertext-ciphertext multiplication, as well as modulus and key switching can be found in [8, 17]. Ciphertext-ciphertext addition is done component-wise and plaintext-ciphertext multiplication simply multiplies the plaintext with each ciphertext component. Ciphertext-plaintext addition can be done by adding the plaintext to the first component of the ciphertext. (Equivalently, one could generate a ciphertext for the plaintext by encrypting it with zero-randomness and then use the ciphertext-ciphertext addition.)

A.1.1 BGV Noise Drowning. We are interested in an encryption enc' with additional noise (*drowning noise*) that is large enough to statistically hide the decryption noise of plaintext of the form $x \cdot \langle y \rangle$, i.e., the following.

Theorem A.1. *The encryption with drowning noise* $\mathrm{enc}'_{\mathrm{pk}}(z)$ *statistically hides the noise of* $x \cdot \mathrm{enc}_{\mathrm{pk}}(y)$ *for arbitrary* $x, y, z \in \mathcal{R}_p$.

This is used in LowGear [31] to build a secure triple generation from only linear homomorphic encryption. The original approach of LowGear simply choses encryption randomness (and z) exponentially larger than for normal encryption. We give the newer version, e.g., implemented in [29], with

$$\mathrm{enc}'_{\mathrm{pk}}(z, \mathbf{r'}) = \mathrm{enc}_{\mathrm{pk}}(z, \mathbf{r'}) \tag{22}$$

where $\mathbf{r'} = (u', v', w')$ is not sampled from RC_σ^N but from $\mathrm{ZO}^N \times \mathrm{UB}_B^N \times \mathrm{DG}_\sigma^N$ with

$$B \geq 2^{\mathrm{sec}} \cdot \left(\left\| \left\lceil \frac{\mathrm{partdec}_{\mathrm{sk}}(x \cdot \langle y \rangle)}{p} \right\rfloor \right\|_\infty + \left\| e \cdot u' \right\|_\infty + \left\| w' \cdot s \right\|_\infty \right),$$

where partdec is the partial decryption, i.e., decryption without reduction modulo p. This is also the noise than can be observed after decryption.

Proof of Theorem A.1. Let $\langle x \cdot y \rangle = x \cdot \mathrm{enc}_{\mathrm{pk}}(y, \mathbf{r})$ with $\mathbf{r} = (u, v, w)$. We get (similarly to the proof of Theorem A.3)

$$\begin{aligned}
\mathrm{partdec}_{\mathrm{sk}}(x \cdot \langle y \rangle) &= \mathrm{partdec}_{\mathrm{sk}}(x \cdot \mathrm{enc}_{\mathrm{pk}}(y, \mathbf{r})) \\
&= \mathrm{partdec}_{\mathrm{sk}}(x \cdot b \cdot u + p \cdot x \cdot v + x \cdot y, \\
&\qquad\qquad x \cdot a \cdot u + p \cdot x \cdot w) \\
&= x \cdot y + x \cdot b \cdot u + p \cdot x \cdot v \\
&\quad - x \cdot a \cdot u \cdot s - p \cdot x \cdot w \cdot s \\
&= x \cdot y + x \cdot a \cdot s \cdot u + p \cdot x \cdot e \cdot u + p \cdot x \cdot v \\
&\quad - x \cdot a \cdot u \cdot s - p \cdot x \cdot w \cdot s \\
&= x \cdot y + p \cdot x \cdot e \cdot u + p \cdot x \cdot v - p \cdot x \cdot w \cdot s
\end{aligned}$$

and analogously

$$\mathrm{partdec}_{\mathrm{sk}}(\mathrm{enc}'_{\mathrm{pk}}(z, \mathbf{r'})) = z + p \cdot e \cdot u' + p \cdot v' - p \cdot w' \cdot s.$$

By choice of B, $\mathrm{enc}'_{\mathrm{pk}}(z, \mathbf{r'})$ statistically hides (the noise of) $x \cdot \langle y \rangle$ as $\mathrm{partdec}_{\mathrm{sk}}(x \cdot \langle y \rangle + \mathrm{enc}'_{\mathrm{pk}}(z, \mathbf{r'}))$ and $\mathrm{partdec}_{\mathrm{sk}}(\mathrm{enc}'_{\mathrm{pk}}(z, \mathbf{r'}))$ are statistically indistinguishable. □

A.2 Applying Bian et al.'s Modifications to Linear Homomorphic BGV

Bian et al. [6] modified (private-key) BFV to homomorphically apply an arbitrary linear operation to encrypted data vectors. Here, we present the corresponding modification to (public-key) BGV. The generation and keys remain the same as in Appendix A.1. However, we only use the vector notation of polynomial multiplication and explicit negacyclic convolution instead of polynomial multiplication here, i.e., $\mathrm{sk} = \boldsymbol{s}$ and $\mathrm{pk} = (\boldsymbol{a}, \boldsymbol{b})$ with $\boldsymbol{b} = \boldsymbol{a} \bar{*} \boldsymbol{s} + p \cdot \boldsymbol{e}$ where $\boldsymbol{a}$ is sampled uniformly random from $\mathbb{Z}_q^N$ and $\boldsymbol{e}$ is sampled with DG_σ^N.

A tool that Bian et al. use (and that is usually not used for BGV) is the representation of polynomial multiplications (or negacyclic convolutions) with (nega)circulant matrices $\overline{\mathrm{circ}} : \mathbb{Z}^N \to \mathbb{Z}^{N \times N}, \boldsymbol{v} \mapsto \overline{\mathrm{circ}}(\boldsymbol{v})$ where

$$\overline{\mathrm{circ}}(\boldsymbol{v})[i, j] := (-1)^{i - j \operatorname{div} N} \cdot \boldsymbol{v}[i - j \bmod N], \tag{23}$$
$$\boldsymbol{c} = \boldsymbol{a} \bar{*} \boldsymbol{b} = \boldsymbol{b} \bar{*} \boldsymbol{a} = \overline{\mathrm{circ}}(\boldsymbol{a}) \cdot \boldsymbol{b} = \overline{\mathrm{circ}}(\boldsymbol{b}) \cdot \boldsymbol{a}. \tag{24}$$

This means, we can write the typical polynomial multiplications in terms of matrix-vector multiplications.

With this, encryption is also similar to (15) but the second ciphertext component is expanded:

$$\mathrm{expandenc}_{\mathrm{pk}}(\boldsymbol{x}, \boldsymbol{r}) := (\boldsymbol{b} \bar{*} \boldsymbol{u} + p \cdot \boldsymbol{v} + \boldsymbol{x}, \overline{\mathrm{circ}}(\boldsymbol{a} \bar{*} \boldsymbol{u} + p \cdot \boldsymbol{w})), \tag{25}$$

where the encryption randomness $\boldsymbol{r} = (\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w})$ is again sampled from RC_σ^N. We use $\langle\!\langle \cdot \rangle\!\rangle_{\mathrm{pk}}$ analogously to $\langle \cdot \rangle_{\mathrm{pk}}$ as notation for expanded ciphertexts under public key pk. Decryption is similar to

(16) but the second part of the ciphertext is now multiplied with $\boldsymbol{s}$ as a matrix-vector product (instead of a polynomial multiplication):

$$\text{expanddec}_{\text{sk}}(\langle\!\langle \boldsymbol{x} \rangle\!\rangle) \coloneqq (\langle\!\langle \boldsymbol{x} \rangle\!\rangle[0] - \langle\!\langle \boldsymbol{x} \rangle\!\rangle[1] \cdot \boldsymbol{s}) \bmod p. \tag{26}$$

As before, all operation in (25) and (26) are modulo q (except the reduction modulo p in (26)).

Theorem A.2. *The modified BGV scheme (pictured above) is a correct (public-key) encryption scheme and CPA secure.*

Proof. This can be seen by following the proof of [6] but with the modified BGV scheme instead of BFV. We summarize the core observations.

Correctness: The scheme still decrypts correctly as we simply split up the negacyclic convolution of (16) (that appears in form of a polynomial multiplication) in two. This can be done using (24).

CPA Security: Consider the standard BGV encryption of (15). Compared to (25), $\overline{\text{circ}}$ is applied to the second component of a standard BGV ciphertext. This can be efficiently done (and undone) without any secret information. Thus, the CPA security of the modified scheme can be trivially reduced to the CPA security of the standard BGV scheme. □

Remark A.1. Indeed, we use the standard BGV encryption, ZKPs, etc. in our protocols to expand BGV ciphertexts on demand with $\text{expand}(\langle \boldsymbol{x} \rangle) \coloneqq (\langle \boldsymbol{x} \rangle[0], \overline{\text{circ}}(\langle \boldsymbol{x} \rangle[1])) = \langle\!\langle \boldsymbol{x} \rangle\!\rangle$.

In addition to simple linear operations (ciphertext-ciphertext addition, ciphertext-plaintext addition, and plaintext-ciphertext multiplication), which can be performed as for standard BGV, the modified scheme allows for applying linear operations on the encrypted plaintext vector.

Theorem A.3. *Let $\boldsymbol{M} \in \mathbb{Z}_p^{N \times N}$ be the matrix for an arbitrary linear transformation and $\boldsymbol{x} \in \mathbb{Z}_p^N$. Then,*

$$\text{expanddec}_{\text{sk}}(\boldsymbol{M} \cdot \text{expandenc}_{\text{pk}}(\boldsymbol{x}, \boldsymbol{r})) = \boldsymbol{M} \cdot \boldsymbol{x} \tag{27}$$

for valid encryption randomness $\boldsymbol{r}$ and

$$\boldsymbol{M} \cdot \langle\!\langle \boldsymbol{x} \rangle\!\rangle \coloneqq (\boldsymbol{M} \cdot \langle\!\langle \boldsymbol{x} \rangle\!\rangle[0], \boldsymbol{M} \cdot \langle\!\langle \boldsymbol{x} \rangle\!\rangle[1]). \tag{28}$$

Proof. Similar to Theorem A.2, this can be done by following the steps of the proofs in [6]. Let $\boldsymbol{r} = (\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w})$. Then,

$$\begin{aligned}
&\text{expandpartdec}_{\text{sk}}(\boldsymbol{M} \cdot \text{enc}_{\text{pk}}(\boldsymbol{x}, \boldsymbol{r})) \\
&= \boldsymbol{M} \cdot (\boldsymbol{b} \circledast \boldsymbol{u} + p \cdot \boldsymbol{v} + \boldsymbol{x}) - \boldsymbol{M} \cdot \overline{\text{circ}}(\boldsymbol{a} \circledast \boldsymbol{u} + p \cdot \boldsymbol{w}) \cdot \boldsymbol{s} && (29)\\
&= \boldsymbol{M}\boldsymbol{x} + \boldsymbol{M}(\boldsymbol{b} \circledast \boldsymbol{u}) - \boldsymbol{M}\,\overline{\text{circ}}(\boldsymbol{a} \circledast \boldsymbol{u})\boldsymbol{s} + p\boldsymbol{M}\boldsymbol{v} - p\boldsymbol{M}\,\overline{\text{circ}}(\boldsymbol{w})\boldsymbol{s} && (30)\\
&= \boldsymbol{M}\boldsymbol{x} + \boldsymbol{M}(\boldsymbol{a} \circledast \boldsymbol{s} \circledast \boldsymbol{u}) - \boldsymbol{M}\,\overline{\text{circ}}(\boldsymbol{a} \circledast \boldsymbol{u})\boldsymbol{s} \\
&\quad + p\boldsymbol{M}(\boldsymbol{e} \circledast \boldsymbol{u}) + p\boldsymbol{M}\boldsymbol{v} - p\boldsymbol{M}\,\overline{\text{circ}}(\boldsymbol{w})\boldsymbol{s} && (31)\\
&= \boldsymbol{M}\boldsymbol{x} + p\boldsymbol{M}(\boldsymbol{e} \circledast \boldsymbol{u}) + p\boldsymbol{M}\boldsymbol{v} - p\boldsymbol{M}(\boldsymbol{w} \circledast \boldsymbol{s}) && (32)
\end{aligned}$$

where expandpartdec is the decryption of (26) without the reduction modulo p. Equation (29) simply applies the definition of expandpartdec (cf. (26)) and (28), while (30) and (31) make use of the linearity of $\circledast$ (and $\overline{\text{circ}}$) where (31) also applies the definition of $\boldsymbol{b}$. Finally, (32) follows from (24): $\boldsymbol{M}(\boldsymbol{a} \circledast \boldsymbol{s} \circledast \boldsymbol{u}) = \boldsymbol{M}(\boldsymbol{a} \circledast \boldsymbol{u} \circledast \boldsymbol{s}) = \boldsymbol{M}\,\overline{\text{circ}}(\boldsymbol{a} \circledast \boldsymbol{u})\boldsymbol{s}$.

The partial decryption result of (32) modulo p is thus the required product $\boldsymbol{M} \cdot \boldsymbol{x}$ if the decryption noise (all of (32)) is not too large for decryption. Similarly to LowGear, the decryption parameters (mainly q and N) have to be chosen in such a way. □

Prove$(I, \mathbf{x}_i, \mathbf{r}_i)$**:** With parameters $I \subseteq [..N), U, V \in \mathbb{Z}, \text{pk}_j$ for $j \in [..n)$, do the following.

1. Initialization Phase:
1.1. Compute $\langle \mathbf{x}_i \rangle_{\text{pk}_i} \coloneqq \text{enc}_{\text{pk}_i}(\mathbf{x}_i, \mathbf{r}_i)$.
2. Commitment Phase:
2.1. Sample $\mathbf{y}_i \in \mathcal{R}_q^V$ and $\mathbf{s}_i \in (\mathcal{R}_q^V)^3$ with

$$\begin{aligned}
\|\mathbf{y}_i[k]\|_\infty &\le 128 \cdot N \cdot \text{sec}_{\text{ZK}}^2 \cdot \frac{p}{2} \\
\|\mathbf{s}_i[k][l]\|_\infty &\le 128 \cdot N \cdot \text{sec}_{\text{ZK}}^2 \cdot 3 \cdot \rho \\
\mathbf{y}_i[k][l'] &= 0
\end{aligned}$$

for $(k, l, l') \in V \times 3 \times I$.
2.2. Compute $\langle \mathbf{y}_i \rangle_{\text{pk}_i} \coloneqq \text{enc}_{\text{pk}_i}(\mathbf{y}_i, \mathbf{s}_i)$.
3. Challenge Phase:
3.1. Compute $\text{hash}(\langle \mathbf{y}_i \rangle_{\text{pk}_i}, \langle \mathbf{x}_i \rangle_{\text{pk}_i}) =: \boldsymbol{e}_j \in \{0,1\}^{\text{sec}_{\text{ZK}}}$ for each party P_j.
3.2. Compute $\boldsymbol{W}_{\boldsymbol{e}_j}$ with $\boldsymbol{W}_{\boldsymbol{e}_j}[k, l] \coloneqq \boldsymbol{e}_j[k-l]$ if $0 \le k-l < \text{sec}_{\text{ZK}}$ and zero otherwise for each party P_j.
4. Response Phase:
4.1. Compute $\mathbf{z}_i \coloneqq \mathbf{y}_i + \boldsymbol{W}_{\boldsymbol{e}_i} \cdot \mathbf{x}_i$ and $\mathbf{t}_i \coloneqq \boldsymbol{W}_{\boldsymbol{e}_i} \cdot \mathbf{r}_i$.
4.2. If for any $(k, l) \in V \times 3$

$$\begin{aligned}
\|\mathbf{z}_i[k]\|_\infty &> 128 \cdot N \cdot \text{sec}_{\text{ZK}}^2 \cdot \frac{p}{2} - \text{sec}_{\text{ZK}} \cdot \frac{p}{2} \\
\|\mathbf{t}_i[k][l]\|_\infty &> 128 \cdot N \cdot \text{sec}_{\text{ZK}}^2 \cdot 3 \cdot \rho - \text{sec}_{\text{ZK}} \cdot \rho,
\end{aligned}$$

abort and restart the protocol.
4.3. Broadcast $(\langle \mathbf{x}_i \rangle_{\text{pk}_i}, \langle \mathbf{y}_i \rangle_{\text{pk}_i}, \mathbf{z}_i, \mathbf{t}_i)$.
5. Verification Phase:
5.1. Compute $\langle \mathbf{d}_j \rangle_{\text{pk}_j} \coloneqq \text{enc}_{\text{pk}_j}(\mathbf{z}_j, \mathbf{t}_j)$ for $j \in [..n)$.
5.2. Check whether $\langle \mathbf{d}_j \rangle_{\text{pk}_j} = \langle \mathbf{y}_j \rangle_{\text{pk}_j} + \boldsymbol{W}_{\boldsymbol{e}_j} \cdot \langle \mathbf{x}_j \rangle_{\text{pk}_j}$ and whether

$$\begin{aligned}
\|\mathbf{z}_j[k]\|_\infty &\le 128 \cdot N \cdot \text{sec}_{\text{ZK}}^2 \cdot \frac{p}{2} \\
\|\mathbf{t}_j[k][l]\|_\infty &\le 128 \cdot N \cdot \text{sec}_{\text{ZK}}^2 \cdot 3 \cdot \rho \cdot \rho_l \\
\mathbf{z}_j[k][l'] &= 0
\end{aligned}$$

for $(j, k, l, l') \in n \times V \times 3 \times I$.
5.3. If all previous checks passed, accept the ciphertexts $\langle \mathbf{x}_j \rangle_{\text{pk}_j}$, otherwise reject them.

Figure 8: SPDZ-Style ZKPoK Subprotocol at Party P_i

A.3 Zero-Knowledge Proofs

In the following, we present the zero-knowledge proofs of knowledge (ZKPoKs) used in our protocols. First, we present a non-interactive proof based on SPDZ [18] (which utilizes the Fiat-Shamir heuristic). Then, we give an interactive TopGear-style (multiparty) ZKPoK [3]. The first is used in LowGear-style protocols and the second one in HighGear-style protocols.

A.3.1 SPDZ-Style ZKPs. Our SPDZ-style ZKP can be found in Fig. 8. We slightly change the ZKP compared to SPDZ by requiring the plaintexts to be zero at fixed positions. In [18], this is only done with $I = \emptyset$ and $I = [1..N)$. However, one can easily prove this general version secure in the same way as the original ZKP of [18].

Prove$(I, \mathbf{x}_i, \mathbf{r}_i)$**:** With parameters $I \subseteq [..N), U, V \in \mathbb{Z}, \mathrm{pk}_j$ for $j \in [..n)$, do the following.

1. Initialization Phase:
1.1. Ensure that $\mathrm{pk}_j = \mathrm{pk}$ for all $j \in [..n)$.
1.2. Broadcast $\langle \mathbf{x}_i \rangle \coloneqq \mathrm{enc}_{\mathrm{pk}}(\mathbf{x}_i, \mathbf{r}_i)$.
2. Commitment Phase:
2.1. Sample $\mathbf{y}_i \in \mathcal{R}_q^V$ and $\mathbf{s}_i \in (\mathcal{R}_q^V)^3$ with

$$\|\mathbf{y}_i[k]\|_\infty \leq 2^{\mathrm{sec}_{\mathrm{ZK}}-1} \cdot p$$
$$\|\mathbf{s}_i[k][l]\|_\infty \leq 2^{\mathrm{sec}_{\mathrm{ZK}}} \cdot \rho_l$$
$$\mathbf{y}_i[k][l'] = 0$$

for $(k, l, l') \in V \times 3 \times I$.
2.2. Broadcast $\langle \mathbf{y}_i \rangle \coloneqq \mathrm{enc}_{\mathrm{pk}}(\mathbf{y}_i, \mathbf{s}_i)$.
3. Challenge Phase:
3.1. If $I = \emptyset$, sample a uniformly random $V \times U$ matrix $\boldsymbol{W}$ with entries in $\{X^k\}_{k \in [..m)} \cup \{0\}$ using $\mathcal{F}_{\mathrm{rand}}$. Otherwise, sample the entries of $\boldsymbol{W}$ from $\{0, 1\}$.
4. Response Phase:
4.1. Compute $\mathbf{z}_i \coloneqq \mathbf{y}_i + \boldsymbol{W} \cdot \mathbf{x}_i$ and $\mathbf{t}_i \coloneqq \boldsymbol{W} \cdot \mathbf{r}_i$.
4.2. Broadcast $(\mathbf{z}_i, \mathbf{t}_i)$.
5. Verification Phase:
5.1. Compute $\langle \mathbf{d}_j \rangle \coloneqq \mathrm{enc}_{\mathrm{pk}}(\mathbf{z}_j, \mathbf{t}_j)$ for $j \in [..n)$.
5.2. Compute $\langle \mathbf{x} \rangle \coloneqq \sum_{j=0}^{n-1} \langle \mathbf{x}_j \rangle$, $\langle \mathbf{y} \rangle \coloneqq \sum_{j=0}^{n-1} \langle \mathbf{y}_j \rangle$, $\langle \mathbf{d} \rangle \coloneqq \sum_{j=0}^{n-1} \langle \mathbf{d}_j \rangle$, $\mathbf{z} \coloneqq \sum_{j=0}^{n-1} \mathbf{z}_j$, and $\mathbf{t} \coloneqq \sum_{j=0}^{n-1} \mathbf{t}_j$.
5.3. Check whether $\langle \mathbf{d} \rangle = \langle \mathbf{y} \rangle + \boldsymbol{W} \cdot \langle \mathbf{x} \rangle$ and whether

$$\|\mathbf{z}[k]\|_\infty \leq n \cdot 2^{\mathrm{sec}_{\mathrm{ZK}}} \cdot p$$
$$\|\mathbf{t}[k][l]\|_\infty \leq 2 \cdot n \cdot 2^{\mathrm{sec}_{\mathrm{ZK}}} \cdot \rho_l$$
$$\mathbf{z}_j[k][l'] = 0$$

for $(k, l, l') \in V \times 3 \times I$.
5.4. If all previous checks passed, accept the ciphertexts $\langle \mathbf{x}_j \rangle$ and $\langle \mathbf{x} \rangle$, otherwise reject them.

Figure 9: TopGear-Style ZKPoK Subprotocol at Party P_i

In the protocol, we use a general security parameter $\mathrm{sec}_{\mathrm{ZK}}$ and $\rho \approx 2 \cdot \sigma \cdot \sqrt{N}$ as in [18].

A.3.2 TopGear-Style ZKPs. In Fig. 9, we present the TopGear ZKPoK protocol [3]. This is a n-party ZKPoK and proves that summing up all parties' ciphertexts yields a valid ciphertext. Baum et al. [3] also give only version of this proof for $I = \emptyset$ or $I = [1..N)$. As with the above changes to the original SPDZ ZKPoK, one can easily extend this to arbitrary values of I and prove it secure.

In the protocol, we use a security parameter $\mathrm{sec}_{\mathrm{ZK}}$ for the statistical distance of the real ZKP execution from a simulation and $\rho_0 = 1, \rho_1 = \rho_2 = 20$ just like Baum et al. For $I = \emptyset$, one requires $V \geq (\mathrm{sec}_{\mathrm{soundness}} + 2)/\log(m + 1)$ for security, where $\mathrm{sec}_{\mathrm{soundness}}$ is the security parameter for the proof soundness. For other values of I, $V \geq \mathrm{sec}_{\mathrm{soundness}} + 1$ is required.

A.4 MPC and SPDZ

Here, we want to point out the remaining subprotocols used in our SPDZ-like protocols. This includes the MAC check (Fig. 10), ZKP subprotocols (Fig. 11), initialization or setup phases (Figs. 12 and 13), and distributed decryption (Fig. 14). Additionally, our protocols use several standard functionalities. We do not picture them here but describe their function shortly. $\mathcal{F}_{\mathrm{rand}}$ is used to agree on random values. These values are then available at every party and uniformly random from the required set (usually $\mathbb{F}_p$ elements or challenges for TopGear ZKPs; cf. Appendix A.3.2). $\mathcal{F}_{\mathrm{commit}}$ models a synchronization step where all parties first send a value to

CheckMAC$(x_1, \ldots, x_t)$**:** For opened values $x_k \in \mathbb{F}_p$, where the party also has access to $[\![x_k]\!]_i$, $k \in [..t)$, do the following. (Adapted from [17].)

1. Let $\boldsymbol{x} \coloneqq (x_1, \ldots, x_t)$ and let $[\boldsymbol{\gamma}]_i \coloneqq [\alpha \cdot \boldsymbol{x}]_i$ be the vector of P_i's MAC shares.
2. Sample a uniformly random $\boldsymbol{r}$ using $\mathcal{F}_{\mathrm{rand}}$.
3. Compute the dot product $x \coloneqq \mathrm{dot}(\boldsymbol{r}, \boldsymbol{x})$.
4. Compute $[\gamma]_i \coloneqq \mathrm{dot}(\boldsymbol{r}, [\boldsymbol{\gamma}]_i)$ and $[\sigma]_i \coloneqq [\gamma]_i - x \cdot [\alpha]_i$.
5. Broadcast (using commitments via $\mathcal{F}_{\mathrm{commit}}$) $[\sigma]_i$.
6. Reconstruct $\sigma \coloneqq \sum_{j=0}^{n-1} [\sigma]_j$ and abort if $\sigma \neq 0$.

Figure 10: MAC Check Subprotocol at Party P_i

ZKP(I)**:** Given an index set $I \subseteq [..N)$, compute ciphertexts for plaintexts $\boldsymbol{x}_i$ with $\boldsymbol{x}_i[l] = 0$ for $l \in I$ for each party and prove in ZK that a suitable witness exists. Output the own plaintext and the ciphertext for each party.

1. Keep track of a list L_I with elements in $\mathcal{R}_p \times C^n$.
2. If L_I is empty, do the following.
2.1. Let U be the number of ciphertexts to amortize over.
2.2. Sample random $\mathbf{x}_i \in \mathcal{R}_p^U$ with $\mathbf{x}_i[k][l] = 0$ for $(k, l) \in U \times I$. Sample encryption randomness $\mathbf{r}$ for $\mathbf{x}_i$.
2.3. If the protocol is LowGear-style, perform a SPDZ-style ZKP (cf. Appendix A.3.1). If the protocol is HighGear-style, perform a TopGear-style ZKP (cf. Appendix A.3.2). Obtain $\langle \mathbf{x}_j \rangle_{\mathrm{pk}_j}$ for each party P_j.
2.4. If the protocol is HighGear-style and $I = \emptyset$, define $\mathbf{x}_i \coloneqq 2 \cdot \mathbf{x}_i$ and $\langle \mathbf{x}_j \rangle_{\mathrm{pk}_j} \coloneqq 2 \cdot \langle \mathbf{x}_j \rangle_{\mathrm{pk}_j}$ for each party P_j.
2.5. Append $(\mathbf{x}_i[k], \langle \mathbf{x}_0[k] \rangle_{\mathrm{pk}_0}, \ldots, \langle \mathbf{x}_{n-1}[k] \rangle_{\mathrm{pk}_{n-1}})$ for $k \in [..U)$ to L_I.
3. Pop and return the last element of L_I.

Figure 11: ZKP Utility for the Offline Phases at Party P_i

Init: Compute $[\alpha]_i$ and $\langle [\alpha]_j \rangle_{\mathrm{pk}_j}$ for $j \in [..n)$. (Adapted from [31].)

1. Run $\mathcal{F}_{\mathrm{setup}}$ to establish public-key private-key pairs $(\mathrm{pk}_j, \mathrm{sk}_j)$ for each party P_j, where the public keys are known afterwards at every party.
2. Sample the MAC key share $[\alpha]_i \xleftarrow{\$} \mathbb{F}_p$.
3. Sample $\mathbf{r}_i$ and perform a SPDZ-style ZKP with $I = [1..N)$.

Figure 12: Initialization Step of the Linear Homomorphic Offline Phase Used in $\Pi_{\text{offline-LHE}}$ (cf. Fig. 7) at Party P_i

Init: Compute $[\alpha]_i$ and $\langle\alpha\rangle$. (Adapted from [3].)

1. Run $\mathcal{F}_{\text{setup}}$ to establish a shared public-key pk and private-key shares $[\text{sk}]_j$ at each party P_j.
2. Sample the MAC key share $[\alpha]_i \xleftarrow{\$} \mathbb{F}_p$.
3. Sample $\mathbf{r}_i$ and perform a TopGear-style ZKP with $I = [1..N)$ where the initialization step of the proof additionally uses commitments (via $\mathcal{F}_{\text{commit}}$) to broadcast the ciphertexts The remaining steps can be iterated instead of choosing a large value of V to achieve the same security with smaller (memory) overhead [3].

Figure 13: Initialization Step of the Somewhat Homomorphic Offline Phase Used in $\Pi_{\text{offline-SHE}}$ (cf. Fig. 19) at Party P_i

DistDec$(\langle \mathsf{x}\rangle)$**:** Perform distributed decryption to obtain x. (Adapted from [18].)

1. Let B be a bound on the noise of $\langle \mathsf{x}\rangle$ and let $\langle \mathsf{x}\rangle =: (\mathsf{c}_0, \mathsf{c}_1)$.
2. Sample $\mathsf{m}_i \xleftarrow{\$} [0..B \cdot 2^{\text{sec}}/(n \cdot p))^N$.
3. Broadcast $\mathsf{x}'_i := \delta_i \cdot \mathsf{c}_0 - \mathsf{c}_1 \cdot [\text{sk}]_i + p \cdot \mathsf{m}_i$
4. Output $\mathsf{x} := \sum_{j=0}^{n-1} \mathsf{x}'_j \bmod p$

ShareDec$(\langle \mathsf{x}\rangle)$**:** Perform distributed decryption to obtain $[\mathsf{x}]_i$. (Adapted from [31].)

1. Let B be a bound on the noise of $\langle \mathsf{x}\rangle$ and let $\langle \mathsf{x}\rangle =: (\mathsf{c}_0, \mathsf{c}_1)$.
2. Sample $\mathsf{m}_i \xleftarrow{\$} [0..B \cdot 2^{\text{sec}})^N$.
3. Broadcast $\mathsf{x}'_i := \delta_i \cdot \mathsf{c}_0 - \mathsf{c}_1 \cdot [\text{sk}]_i - \mathsf{m}_i$.
4. Output $[\mathsf{x}]_i := \delta_i \cdot (\sum_{j=0}^{N-1} \mathsf{x}'_j) + \mathsf{m}_i \bmod p$.

Figure 14: Distributed Decryption for the Somewhat Homomorphic Offline Phase Used in $\Pi_{\text{offline-SHE}}$ (cf. Fig. 19) at Party P_i

the functionality and then receive every other party's value after all messages of the first round arrived. Finally, $\mathcal{F}_{\text{setup}}$ models key generation for BGV. Depending on the protocol style (LowGear or HighGear), these are either keys for every parts or a single public key and a secret-shared private key.

B SCALAR MULTIPLICATION PACKING

To show the generality of the framework presented in Section 3.1, we show that the standard way to generate scalar multiplication tuples can be captured by our definitions as well. This also allows us to generate scalar multiplication triples with Figs. 7 and 19 without defining specific interfaces for this case (as the instantiation of the protocols with the below instantiation of the packing framework collapses to the standard construction).

Theorem B.1. *Let $\odot$ be the component-wise multiplication of two vectors. Let $D = D' = [..N)$. Let $\boldsymbol{a}$ and $\boldsymbol{f}$ be vectors of length N. Let $\boldsymbol{b} = \text{packi}(\odot, D, D', \boldsymbol{a})$ and $\boldsymbol{g} = \text{packf}(\odot, D, D', \boldsymbol{f})$. Then*

$$(\boldsymbol{a} \odot \boldsymbol{f}) = \text{unpackr}(\odot, D, D', \boldsymbol{g} \circledast \boldsymbol{b}) \tag{33}$$

for $\text{packi}(\odot, D, D', \boldsymbol{a})[i] = \text{CRT}^{-1}(\boldsymbol{a})[i]$, $\text{packf}(\odot, \cdot) = \text{packi}(\odot, \cdot)$, and $\text{unpackr}(\odot, D, D', \boldsymbol{c})[i] = \text{CRT}(\boldsymbol{c})[i]$.

Proof. This follows directly from (21). □

Remark B.1. Note that the above defines the (un)packing directly and not via mapping mapi, mapf, mapr. For compatibility with the protocols that make use of the mapping, e.g., Fig. 19, we define them as the identity.

Remark B.2. As individual shares for scalar elements are more versatile than N-vectors of shares, one can add as a last step of **Triples** in Figs. 7 and 19 to split the resulting triple $(\llbracket \boldsymbol{a} \rrbracket_i, \llbracket \boldsymbol{f} \rrbracket_i, \llbracket \boldsymbol{c} \rrbracket_i)$ with $\boldsymbol{c} = \boldsymbol{a} \odot \boldsymbol{f}$ in N scalar triples $(\llbracket a_k \rrbracket_i, \llbracket f_k \rrbracket_i, \llbracket c_k \rrbracket_i) = (\llbracket \boldsymbol{a} \rrbracket_i[k], \llbracket \boldsymbol{f} \rrbracket_i[k], \llbracket \boldsymbol{c} \rrbracket_i[k])$ for $k \in [..N)$.

C CONVOLUTION PACKING (CONTINUED)

After finishing Section 3.2.1 by giving the proof for Theorem 3.3, we also give another recent convolution packing method: Bian et al.'s packing method [6] that performs multiple independent convolutions by performing a single matrix-vector multiplication.

C.1 Multidimensional Convolution Packing (Continued)

Here, we give the proof for Theorem 3.3.

Theorem 3.3. *Let $\boldsymbol{a}$ be a $D = h \times w$ (2d) image and $\boldsymbol{f}$ a $D' = h' \times w'$ (2d) filter. Choose D'' according to the padding mode and let $(h'', w'') = \text{up}(D'')$. For N with $h'' \cdot w'' \le N$ define $\phi : h'' \times w'' \to [..N)$, $(y, x) \mapsto y \cdot w'' + x$. Then the packing of $\boldsymbol{a}$ and $\boldsymbol{f}$ as (1d) N-vectors $\boldsymbol{b} = \text{packi}(*, D, D', \boldsymbol{a})$ and $\boldsymbol{g} = \text{packf}(*, D, D', \boldsymbol{f})$ satisfies*

$$(\boldsymbol{f} * \boldsymbol{a})[y, x] = \text{unpackr}(*, D, D', \boldsymbol{g} \circledast \boldsymbol{b})[y, x] \tag{10}$$

for $(y, x) \in D''$, where the packing method (packi, packf, unpackr) *is induced by* $\text{mapi}(\boldsymbol{x}) = \text{mapf}(\boldsymbol{x}) = \text{mapr}(\boldsymbol{x}) = \phi(\boldsymbol{x})$.

Proof. We start this proof generically for $h'' \ge h$, $h'' \ge h'$, $w'' \ge w$, $w'' \ge w'$. We can rewrite (10) as[6]

$$\begin{aligned}
(\boldsymbol{g} \circledast \boldsymbol{b})[\phi(y, x)] &= \sum_{k=0}^{N-1} \boldsymbol{g}[k] \cdot \boldsymbol{b}[\phi(y, x) - k] \\
&= \sum_{y'=0}^{h''-1} \sum_{x'=0}^{w''-1} \boldsymbol{g}[\phi(y', x')] \cdot \boldsymbol{b}[\phi(y, x) - \phi(y', x')] \\
&= \sum_{y'=0}^{h'-1} \sum_{x'=0}^{w'-1} \boldsymbol{f}[y', x'] \cdot \boldsymbol{b}[\phi(y, x) - \phi(y', x')] && (34) \\
&= \sum_{y'=0}^{h'-1} \sum_{x'=0}^{w'-1} \boldsymbol{f}[y', x'] \cdot \boldsymbol{b}[\phi(y - y', x - x')] && (35) \\
&= \sum_{y'=0}^{h'-1} \sum_{x'=0}^{w'-1} \boldsymbol{f}[y', x'] \cdot \boldsymbol{a}[y - y', x - x'] && (36) \\
&= (\boldsymbol{f} * \boldsymbol{a})[y, x],
\end{aligned}$$

[6] Note that, compared to (1), we drop the -1 factor and modulo N calculation of the index of $\boldsymbol{b}$. Both are considered explicitly later (and we see that in all cases where we would have to multiply by -1 and take the index modulo N, $\boldsymbol{b}[\,\cdot\,] = 0$).

where (34) follows from the definition of $\boldsymbol{g}$ and ϕ. Equation (35) follows from the fact that $\phi(y,x)-\phi(y',x')=\phi(y-y',x-x')$:

$$\begin{aligned}\phi(y,x)-\phi(y',x') &= y\cdot w''+x-(y'\cdot w''-x')\\ &= (y-y')\cdot w''+(x-x'),\end{aligned}\tag{37}$$

which is obviously bilinear. Finally, for (36), we utilize the fact that $\boldsymbol{b}[\phi(y-y',x-x')]=\boldsymbol{a}[y-y',x-x']$ if $(y-y',x-x')\in h\times w=D$ and zero otherwise. To see this, we consider the different padding modes separately.

Full padding: Let $h''=h+h'-1, w''=w+w'-1$. We know that $-h'+1\le y-y'\le h+h'-1$ and $-w'+1\le x-x'\le w+w'-1$. For $y-y'\ge h$ or $x-x'\ge w$, we get $\boldsymbol{b}[\phi(y-y',x-x')]=0$ by construction of $\boldsymbol{b}$. For $x-x'<0$ and $\varphi:=\phi(y-y',x-x')\ge 0$, we get $\varphi=\phi(y-y'-1,x-x' \bmod w'')$ and $w\le x-x' \bmod w''<w''$, i.e., $\boldsymbol{b}[\varphi]=0$. Finally, for $y-y'<0$, we get $(-h'+1)\cdot w''-w'+1\le\varphi<0$. In this case, we have to do the index calculation modulo N. However, the last $(h'-1)\cdot w''+w'-1$ entries of $\boldsymbol{b}$ (or more) are zero: For $\phi(h-1,w-1)<k<h''\cdot w''\le N$, $\boldsymbol{b}[k]=0$. Thus, overall $\boldsymbol{b}[\varphi]=0$ if $(y-y',x-x')\notin D$.

Same padding: Let $h''=h+\lfloor h'/2\rfloor, w''=w+\lfloor w'/2\rfloor$. We know that $\lfloor h'/2\rfloor-h'+1\le y-y'\le h+\lfloor h'/2\rfloor-1$ and $\lfloor w'/2\rfloor-w'+1\le x-x'\le w+\lfloor w'/2\rfloor-1$. For $y-y'\ge h$ or $x-x'\ge w$, we get the same results as above for full padding. For $x-x'<0$ and $\varphi:=\phi(y-y',x-x')\ge 0$, we get $w''\ge x-x' \bmod w''\ge w+2\lfloor w'/2\rfloor-w'+1\ge w$ and therefore the same result as above. For the final case, $y-y'<0$, we get $0>\varphi\ge(\lfloor h'/2\rfloor-h'+1)\cdot w''+\lfloor w'/2\rfloor-w'+1\ge-\lfloor h'/2\rfloor\cdot w''-\lfloor w'/2\rfloor$ and again at least this many elements are zero.

Valid padding: Let $h''=h, w''=w$. By construction, we always have $(y-y',x-x')\in D$. □

Remark C.1. For cross-correlations, we clearly have

$$\begin{aligned}&(\boldsymbol{f}\star\boldsymbol{a})[y-h'+1,x-w'+1]\\ &=\sum_{y'=0}^{h'-1}\sum_{x'=0}^{w'-1}\boldsymbol{f}[y',x']\cdot\boldsymbol{a}[y+y'-h'+1,x+x'-w'+1]\\ &=\sum_{y'=0}^{h'-1}\sum_{x'=0}^{w'-1}\boldsymbol{f}[h'-1-y',w'-1-x']\cdot\boldsymbol{a}[y-y',x-x']\\ &=(\boldsymbol{f}'*\boldsymbol{a})[y,x]\end{aligned}\tag{38}$$

with $\boldsymbol{f}'[y,x]:=\boldsymbol{f}[h'-1-y,w'-1-x]$.[7] Also note that a cross-correlation with full padding would have a domain of $[-h'+1..h)\times[-w'+1..w)$, i.e., the domain is shifted by $(-h'+1,-w'+1)$ compared to a full convolution.[8] Therefore, simply reversing $\boldsymbol{f}$ (i.e., convolving with $\boldsymbol{f}'$) is enough to get the cross-correlation result

$$(\boldsymbol{f}\star\boldsymbol{a})=\mathrm{unpackr}(\star,D,D',\boldsymbol{g}\mathbin{\bar{*}}\boldsymbol{b}).\tag{39}$$

In other words, to get an analogue of Theorem 3.3 for cross-correlations (with (39) instead of (10)), one would use $\mathrm{mapf}(y,x)=\phi(h'-1-y,w'-1-x)$, while $\mathrm{mapi}(\star,\cdot)=\mathrm{mapi}(*,\cdot)$ and $\mathrm{mapr}(\star,\cdot)=\mathrm{mapr}(*,\cdot)$.

[7] Equation (38) follows by reversing the sums over x' and y'.
[8] The same is true for same and valid padding.

C.2 Bian et al.'s Parallel Convolution Packing

In [6], Bian et al. propose a technique to perform multiple independent convolutions in parallel. In contrast to most other approaches discussed in this work, their approach does not encode multiple convolutions into a single polynomial multiplication. Instead, they make use of specially constructed matrices. More specifically, they aim to compute $\boldsymbol{f}_l*\boldsymbol{a}_l$ for (1d) images $\boldsymbol{a}_1,\dots,\boldsymbol{a}_d\in R^D$ and (1d) filters $\boldsymbol{f}_1,\dots,\boldsymbol{f}_d\in R^{D'}$.[9] Our bilinear operation $\mathrm{op}: R^{D\times d}\times R^{D'\times d}\to R^{D''\times d}$ is in this case just $((\boldsymbol{a}_1,\dots,\boldsymbol{a}_d),(\boldsymbol{f}_1,\dots,\boldsymbol{f}_d))\mapsto(\boldsymbol{f}_1*\boldsymbol{a}_1,\dots,\boldsymbol{f}_d*\boldsymbol{a}_d)$. For the packing, one would define packi as the concatenation of the $\boldsymbol{a}_l$s into a single vector $\boldsymbol{a}$ and the output of packf as $\boldsymbol{F}=\mathrm{diag}(\overline{\mathrm{circ}}(\boldsymbol{f}_1),\dots,\overline{\mathrm{circ}}(\boldsymbol{f}_d))$, i.e., a block-diagonal matrix with matrices that correspond to convolutions with $\boldsymbol{f}_l$ in the l-th block. $\mathrm{op}_{\mathcal{R}}: R^{N\times N}\times R^N\to R^N$ (or $\mathcal{R}^N\times\mathcal{R}\to\mathcal{R}$) is then the matrix multiplication $(\boldsymbol{F},\boldsymbol{a})\mapsto\boldsymbol{F}\cdot\boldsymbol{a}$, which yields $\boldsymbol{F}\cdot\boldsymbol{a}=(\boldsymbol{f}_1\mathbin{\bar{*}}\boldsymbol{a}_1,\dots,\boldsymbol{f}_d\mathbin{\bar{*}}\boldsymbol{f}_d)^{\mathsf{T}}$. This way, one can obtain a vector that encodes the concatenation of d parallel/independent convolutions with a single matrix-vector product. Please note that our general framework also supports this matrix variation. To evaluate this securely, they present a variant of a homomorphic encryption scheme that supports such a matrix-vector multiplication of a plaintext matrix with an encrypted vector. We further extend this, such that the use in LowGear-style protocols is secure (cf. Appendix F.3.2).

Remark C.2. Instead of of $\overline{\mathrm{circ}}(\boldsymbol{f}_l)$, one could place matrices that directly encode the convolution with $\boldsymbol{f}_l$ on the diagonal of $\boldsymbol{F}$. The same can be done for cross-correlations.

Remark C.3. This construction can be extended and used in a straightforward way to convolve multiple 2d images with the same filters, as is needed for conv2d (and also dconv2d).

D NEW PACKING METHODS (CONTINUED)

Here, we present the correctness proof for our new packing methods of Section 4.

D.1 Simple Convolution Packing (Continued)

First, we start with the proof of Theorem 4.1. Recall $(\boldsymbol{b}_k)_{k\in d}=\mathrm{packi}(\mathrm{op},D,D',\boldsymbol{a})\in\mathcal{R}^d$, and $(\boldsymbol{g}_k)_{k\in d}=\mathrm{packf}(\mathrm{op},D,D',\boldsymbol{f})\in\mathcal{R}^d$.

Theorem 4.1. *Let $\boldsymbol{a}$ be a (4d) $D=b\times h\times w\times d$ tensor and let $\boldsymbol{f}$ be a (4d) $D'=d'\times d\times h'\times w'$ tensor. Choose D'' according to the padding mode and let $(h'',w'')=\mathrm{up}(D'')$. Let $\phi(i,j,y,x)=((i\cdot d'+j)\cdot h''+y)\cdot w''+x$ be the canonical indexing into a (flattened 4d) $b\times d'\times h''\times w''$ tensor. Let*

$$\begin{aligned}\mathrm{mapi}(i,y,x,j)&=(\phi(i,0,y,x),j)\in N\times d\\ \mathrm{mapf}(j',j,y,x)&=(\phi(0,j',h'-1-y,w'-1-x),j)\in N\times d\\ \mathrm{mapr}(i,y,x,j')&=\phi(i,j',y,x).\end{aligned}$$

For the induced packing (packi, packf, unpackr) *and $\boldsymbol{b}_k,\boldsymbol{g}_k$ as above,*

$$\mathrm{conv2d}(\boldsymbol{a},\boldsymbol{f})=\mathrm{unpackr}(\mathrm{conv2d},D,D',\sum_{k=0}^{d-1}\boldsymbol{g}_k\mathbin{\bar{*}}\boldsymbol{b}_k).\tag{11}$$

Proof. First note that $\boldsymbol{b}_k\in\mathcal{R}$ is induced by $\mathrm{mapi}(\cdot,\cdot,\cdot,k)$ and $\boldsymbol{g}_k\in\mathcal{R}$ is induced by $\mathrm{mapf}(\cdot,k,\cdot,\cdot)$. Now let $\varphi=\phi(i,j',y,x)$, $\varphi'=$

[9] 2d images and filters could be first encoded as 1d vectors similar to Theorem 3.3.

$\phi(i', j, y', x'), \varphi'' = \phi(i, j' - j, y - y', x - x'), \bar{y}' = h' - 1 - y', \bar{x}' = w' - 1 - x'$. As in the proof of Theorem 3.3, the -1 factors and modulo computations for accessing $\boldsymbol{b}_k$ are dropped for readability.

$$
\begin{aligned}
&(\sum_{k=0}^{d-1} \boldsymbol{g}_k \mathbin{\bar{*}} \boldsymbol{b}_k)[\varphi] = \sum_{k=0}^{d-1}\sum_{k'=0}^{N-1} \boldsymbol{g}_k[k'] \cdot \boldsymbol{b}_k[\varphi - k'] \\
&= \sum_{k=0}^{d-1}\sum_{i'=0}^{b-1}\sum_{j=0}^{d'-1}\sum_{y'=0}^{h''-1}\sum_{x'=0}^{w''-1} \boldsymbol{g}_k[\varphi'] \cdot \boldsymbol{b}_k[\varphi - \varphi'] && (40)\\
&= \sum_{k=0}^{d-1}\sum_{i'=0}^{0}\sum_{j=0}^{d'-1}\sum_{y'=0}^{h'-1}\sum_{x'=0}^{w'-1} \boldsymbol{f}[j, k, \bar{y}', \bar{x}'] \cdot \boldsymbol{b}_k[\varphi - \varphi'] && (41)\\
&= \sum_{k=0}^{d-1}\sum_{j=0}^{d'-1}\sum_{y'=0}^{h'-1}\sum_{x'=0}^{w'-1} \boldsymbol{f}[j, k, \bar{y}', \bar{x}'] \cdot \boldsymbol{b}_k[\varphi''] && (42)\\
&= \sum_{k=0}^{d-1}\sum_{y'=0}^{h'-1}\sum_{x'=0}^{w'-1} \boldsymbol{f}[j', k, \bar{y}', \bar{x}'] \cdot \boldsymbol{b}_k[\phi(i, 0, y - y', x - x')] && (43)\\
&= \sum_{k=0}^{d-1} \boldsymbol{f}[j', k, \cdot, \cdot] \star \boldsymbol{a}[i, \cdot, \cdot, k] && (44)\\
&= \mathrm{conv2d}(\boldsymbol{a}, \boldsymbol{f})[i, y, x, j',],
\end{aligned}
$$

where (40) and (41) follow from the definition of ϕ and $\boldsymbol{g}_k$, i.e., $\boldsymbol{g}_k[\phi(i', \cdot, \cdot, \cdot)] = 0$ for $i' \neq 0$. Next, (42) follows from the linearity of ϕ (similarly to (37)), (43) follows from the definition of $\boldsymbol{b}_k$, i.e., $\boldsymbol{b}_k[\phi(\,\cdot\,, j' - j, \cdot, \cdot)] = 0$ for $j' \neq j$, and (44) follows from the final steps – (35) and (36) – of Theorem 3.3 for the variant mentioned in Remark C.1. □

D.2 Generalization of Huang et al.'s Convolution Packing (Continued)

Here, we give the proof for Theorem 4.2.

Theorem 4.2. *Let $\boldsymbol{a}$ be a (4d) $D = b \times h \times w \times d$ tensor and let $\boldsymbol{f}$ be a (4d) $D' = d' \times d \times h' \times w'$ tensor. Choose D'' according to the padding mode and let $(h'', w'') = \mathrm{up}(D'')$. Let $\phi(i, j, k, y, x) = (((i \cdot d' + j) \cdot d + k) \cdot h'' + y) \cdot w'' + x$ be the canonical indexing into a (flattened 5d) $b \times d' \times d \times h'' \times w''$ tensor. Let $\boldsymbol{b} = \mathrm{packi}(\mathrm{conv2d}, D, D', \boldsymbol{a})$ and $\boldsymbol{g} = \mathrm{packf}(\mathrm{conv2d}, D, D', \boldsymbol{f})$. Then,*

$$\mathrm{conv2d}(\boldsymbol{a}, \boldsymbol{f}) = \mathrm{unpackr}(\mathrm{conv2d}, D, D', \boldsymbol{g} \mathbin{\bar{*}} \boldsymbol{b}) \tag{12}$$

for the packing method (packi, packf, unpackr) *induced by* $\mathrm{mapi}(i, y, x, j) = \phi(i, 0, j, y, x)$, $\mathrm{mapf}(j', j, y, x) = \phi(0, j', d - 1 - j, h' - 1 - y, w' - 1 - x)$, *and* $\mathrm{mapr}(i, y, x, j') = \phi(i, j', d - 1, y, x)$.

Proof. Let $\varphi = \phi(i, j', d - 1, y, x), \varphi' = \phi(i', j, k, y', x'), \varphi'' = \phi(i, j' - j, k, y - y', x - x'), \bar{y}' = h' - 1 - y', \bar{x}' = w' - 1 - x'$. As in the proof of Theorem 3.3, the -1 factors and modulo computations for accessing $\boldsymbol{b}$ are dropped for readability.

$$
\begin{aligned}
&(\boldsymbol{g} \mathbin{\bar{*}} \boldsymbol{b})[\varphi] = \sum_{k'=0}^{N-1} \boldsymbol{g}[k'] \cdot \boldsymbol{b}[\varphi - k'] \\
&= \sum_{i'=0}^{b-1}\sum_{j=0}^{d'-1}\sum_{k=0}^{d-1}\sum_{y'=0}^{h''-1}\sum_{x'=0}^{w''-1} \boldsymbol{g}[\varphi'] \cdot \boldsymbol{b}[\varphi - \varphi'] && (45)\\
&= \sum_{i'=0}^{0}\sum_{j=0}^{d'-1}\sum_{k=0}^{d-1}\sum_{y'=0}^{h'-1}\sum_{x'=0}^{w'-1} \boldsymbol{f}[j, d - 1 - k, \bar{y}', \bar{x}'] \cdot \boldsymbol{b}[\varphi - \varphi'] && (46)\\
&= \sum_{j=0}^{d'-1}\sum_{k=0}^{d-1}\sum_{y'=0}^{h'-1}\sum_{x'=0}^{w'-1} \boldsymbol{f}[j, k, \bar{y}', \bar{x}'] \cdot \boldsymbol{b}[\varphi''] && (47)\\
&= \sum_{k=0}^{d-1}\sum_{y'=0}^{h'-1}\sum_{x'=0}^{w'-1} \boldsymbol{f}[j', k, \bar{y}', \bar{x}'] \cdot \boldsymbol{b}[\phi(i, 0, k, y - y', x - x')] && (48)\\
&= \sum_{k=0}^{d-1} \boldsymbol{f}[j', k, \cdot, \cdot] \star \boldsymbol{a}[i, \cdot, \cdot, k] && (49)\\
&= \mathrm{conv2d}(\boldsymbol{a}, \boldsymbol{f})[i, y, x, j'],
\end{aligned}
$$

where (45) and (46) follow from the definition of ϕ and $\boldsymbol{g}$ ($\boldsymbol{g}[\varphi] = 0$ for $i' \neq 0$), while (47) uses the linearity of ϕ (analogously to (37)) and reverses the sum over k (the d-dimension). Then, (48) follows from the definition of $\boldsymbol{b}$ ($\boldsymbol{b}[\varphi''] = 0$ for $j \neq j''$). Finally, (49) follows analogously to the last steps in the proof of Theorem 3.3, i.e., (35) and (36). □

D.3 Depthwise Convolution Packing (Continued)

Here, we give the proof for Theorem 4.3.

Theorem 4.3. *Let $\boldsymbol{a}$ be a (4d) $D = b \times h \times w \times d$ tensor and let $\boldsymbol{f}$ be a (3d) $D' = d \times h' \times w'$ tensor. Choose D'' according to the padding mode and let $(h'', w'') = \mathrm{up}(D'')$. As in Theorem 4.2, let $\phi(i, j, k, y, x) = (((i \cdot d + j) \cdot d + k) \cdot h'' + y) \cdot w'' + x$ be the canonical indexing into a (flattened 5d) $b \times d \times d \times h'' \times w''$ tensor.*[10] *Let $\boldsymbol{b} = \mathrm{packi}(\mathrm{dconv2d}, D, D', \boldsymbol{a})$ and $\boldsymbol{g} = \mathrm{packf}(\mathrm{dconv2d}, D, D', \boldsymbol{f})$. Then,*

$$\mathrm{dconv2d}(\boldsymbol{a}, \boldsymbol{f}) = \mathrm{unpackr}(\mathrm{dconv2d}, D, D', \boldsymbol{g} \mathbin{\bar{*}} \boldsymbol{b}) \tag{13}$$

for the packing method (packi, packf, unpackr) *induced by* $\mathrm{mapi}(i, y, x, j) = \phi(i, 0, j, y, x)$, $\mathrm{mapf}(j, y, x) = \phi(0, j, 0, h' - 1 - y, w' - 1 - x)$, *and* $\mathrm{mapr}(i, y, x, j) = \phi(i, j, j, y, x)$.

Proof. Let $\varphi = \phi(i, j, j, y, x), \varphi' = \phi(i', j', k', y', x'), \bar{y}' = h' - 1 - y', \bar{x}' = w' - 1 - x'$. As in the proof of Theorem 3.3, the -1 factors and modulo computations for accessing $\boldsymbol{b}$ are dropped for

[10] Compared to Section 4.2, we have $d' = d$.

readability.

$$(\boldsymbol{g} \circledast \boldsymbol{b})[\varphi] = \sum_{k=0}^{N-1} \boldsymbol{g}[k] \cdot \boldsymbol{b}[\varphi - k]$$

$$= \sum_{i'=0}^{b-1} \sum_{j'=0}^{d-1} \sum_{k'=0}^{d-1} \sum_{y'=0}^{h''-1} \sum_{x'=0}^{w''-1} \boldsymbol{g}[\varphi'] \cdot \boldsymbol{b}[\varphi - \varphi'] \tag{50}$$

$$= \sum_{i'=0}^{0} \sum_{j'=0}^{d-1} \sum_{k'=0}^{0} \sum_{y'=0}^{h'-1} \sum_{x'=0}^{w'-1} \boldsymbol{f}[j', \bar{y}', \bar{x}'] \cdot \boldsymbol{b}[\varphi - \varphi'] \tag{51}$$

$$= \sum_{j'=0}^{d-1} \sum_{y'=0}^{h'-1} \sum_{x'=0}^{w'-1} \boldsymbol{f}[j', \bar{y}', \bar{x}'] \cdot \boldsymbol{b}[\phi(i, j - j', j, y - y', x - x')] \tag{52}$$

$$= \sum_{y'=0}^{h'-1} \sum_{x'=0}^{w'-1} \boldsymbol{f}[j, \bar{y}', \bar{x}'] \cdot \boldsymbol{b}[\phi(i, 0, j, y - y', x - x')] \tag{53}$$

$$= \boldsymbol{f}[j, \cdot, \cdot] \star \boldsymbol{a}[i, \cdot, \cdot, j] \tag{54}$$

$$= \mathrm{dconv2d}(\boldsymbol{a}, \boldsymbol{f})[i, y, x, j], \tag{55}$$

where the steps are analogous to the ones in Appendix D.2: Equations (50), (51) and (53) follow – like (45), (46), and (48) – from the definition of $\phi, \boldsymbol{g}, \boldsymbol{b}$, while (52) – just like (47) – uses the linearity of ϕ. Finally, (54) follows – analogously to (49) – from Theorem 3.3. □

E SECURITY OF THE ONLINE PHASE

Before we can prove Theorem 5.1, we give the full online protocol in Fig. 16 and the corresponding functionality in Fig. 15. Note that Π_{online} also uses $\mathcal{F}_{\text{offline}}$ given in Fig. 17 (cf. Appendix F), as well as $\mathcal{F}_{\text{rand}}$ that we describe in Appendix A.4. The security of our offline phase therefore directly follows from the established security guarantees of the underlying constructions.

Theorem 5.1. *The online protocol Π_{online} securely implements the ideal functionality $\mathcal{F}_{\text{online}}$ in the $(\mathcal{F}_{\text{offline}}, \mathcal{F}_{\text{rand}})$-hybrid model.*

Proof. Compared to SPDZ [17, 18], the only difference in our protocol is the use of specialized triples for convolutions (and matrix multiplications). This, however, is just a generalization of the standard Beaver triples for scaler multiplication and is secure for any bilinear operation [14]. □

F SECURITY OF THE OFFLINE PHASE

Figure 17 pictures the offline functionality that we want to implement in classical SPDZ-like protocols. Functionalities and subprotocols are discussed in Appendix A.4.

F.1 Linear Homomorphic Offline Phase

For the LowGear protocol [31], the security proof does not prove the security of an online phase that performs an arithmetic circuit computation and an offline phase that only produces correlated randomness. Instead a somewhat combined functionality $\mathcal{F}_{\text{auth-MPC}}$ is constructed and the proof shows that LowGear securely implements this. We omit an explicit depiction of this functionality here, but the general design follows [31].

$\mathcal{F}_{\text{auth-MPC}}$ behaves as follows. Firstly, a functionality $\mathcal{F}_{\text{auth-linear}}$ can be constructed from $\mathcal{F}_{\text{online}}$ that simply omits the non-linear operations (**Multiply**, **Convolve**, etc.). This corresponds to $\mathcal{F}_{[\![\cdot]\!]}$ in [31]. Secondly, $\mathcal{F}_{\text{auth-MPC}}$ is $\mathcal{F}_{\text{auth-linear}}$ where these omitted operations are contained but changed in the following way. Instead of taking two already assigned and one unassigned identifier, the operation takes three unassigned identifiers. It then samples random triples for the operation (e.g., a random image and a random filter of the correct shape for a convolution and then computes the convolution result) and stores it under the three identifiers. $\mathcal{F}_{\text{auth-MPC}}$ corresponds to $\mathcal{F}_{\text{Triple}}$ in [31].

Also, $\Pi_{\text{offline-LHE}}$ could use the $\mathcal{F}_{\text{auth-linear}}$ internally for linear operations in **Sacrifice** and to authenticate values. Please note, that

Functionality $\mathcal{F}_{\text{online}}$

Init: On input $(\texttt{init}, p)$ from all parties.

1. Setup a storage for a write-only mapping of identifiers to values (or tensors) in $\mathbb{F}_p$.

Input: On input $(\texttt{input}, P_i, \mathrm{ID}(\boldsymbol{x}), \boldsymbol{x})$ from P_i and $(\texttt{input}, P_i, \mathrm{ID}(\boldsymbol{x}))$ from all other parties where $\mathrm{ID}(\boldsymbol{x})$ has not been assigned a value before.

1. Store $(\mathrm{ID}(\boldsymbol{x}), \boldsymbol{x})$.

Add: On input $(\texttt{add}, \mathrm{ID}(\boldsymbol{z}), \mathrm{ID}(\boldsymbol{x}), \mathrm{ID}(\boldsymbol{y}))$ from all parties where $\mathrm{ID}(\boldsymbol{z})$ has not been assigned a value before and $\mathrm{ID}(\boldsymbol{x})$ and $\mathrm{ID}(\boldsymbol{y})$ have been assigned.

1. Retrieve $\boldsymbol{x}$ and $\boldsymbol{y}$ via their identifiers.
2. Store $(\mathrm{ID}(\boldsymbol{z}), \boldsymbol{x} + \boldsymbol{y})$.

Multiply: On input $(\texttt{mul}, \mathrm{ID}(\boldsymbol{z}), \mathrm{ID}(\boldsymbol{x}), \mathrm{ID}(\boldsymbol{y}))$ from all parties where $\mathrm{ID}(\boldsymbol{z})$ has not been assigned a value before and $\mathrm{ID}(\boldsymbol{x})$ and $\mathrm{ID}(\boldsymbol{y})$ have been assigned.

1. Retrieve $\boldsymbol{x}$ and $\boldsymbol{y}$ via their identifiers.
2. Store $(\mathrm{ID}(\boldsymbol{z}), \boldsymbol{x} \odot \boldsymbol{y})$.

Convolve: On input $(\texttt{conv}, \mathrm{ID}(\boldsymbol{z}), \mathrm{ID}(\boldsymbol{x}), \mathrm{ID}(\boldsymbol{y}), \textit{params})$ from all parties where $\mathrm{ID}(\boldsymbol{z})$ has not been assigned a value before and $\mathrm{ID}(\boldsymbol{x})$ and $\mathrm{ID}(\boldsymbol{y})$ have been assigned. *params* are valid convolution parameters (padding, stride, etc.).

1. Retrieve $\boldsymbol{x}$ and $\boldsymbol{y}$ via their identifiers.
2. Store $(\mathrm{ID}(\boldsymbol{z}), \mathrm{conv2d}(\boldsymbol{x}, \boldsymbol{y}))$ where conv2d respects *params*.

DepthwiseConvolve: On input $(\texttt{dconv}, \mathrm{ID}(\boldsymbol{z}), \mathrm{ID}(\boldsymbol{x}), \mathrm{ID}(\boldsymbol{y}), \textit{params})$ from all parties where $\mathrm{ID}(\boldsymbol{z})$ has not been assigned a value before and $\mathrm{ID}(\boldsymbol{x})$ and $\mathrm{ID}(\boldsymbol{y})$ have been assigned. *params* are valid convolution parameters (padding, stride, etc.).

Proceed as in **Convolve** but with dconv2d instead of conv2d.

MatrixMultiply: On input $(\texttt{matmul}, \mathrm{ID}(\boldsymbol{Z}), \mathrm{ID}(\boldsymbol{X}), \mathrm{ID}(\boldsymbol{Y}))$ from all parties where $\mathrm{ID}(\boldsymbol{Z})$ has not been assigned a value before and $\mathrm{ID}(\boldsymbol{X})$ and $\mathrm{ID}(\boldsymbol{Y})$ have been assigned.

1. Retrieve $\boldsymbol{X}$ and $\boldsymbol{Y}$ via their identifiers.
2. Store $(\mathrm{ID}(\boldsymbol{Z}), \boldsymbol{X} \cdot \boldsymbol{Y})$.

Output: On input $(\texttt{output}, \mathrm{ID}(\boldsymbol{x}))$ from all parties where $\mathrm{ID}(\boldsymbol{x})$ has been assigned.

1. Retrieve $\boldsymbol{x}$ and output it to the adversary.
2. If the adversary replies ok, also output this value to all parties. Otherwise, output $\perp$.

Figure 15: Functionality for the Online Phase

Protocol Π_{online}

Init: Setup the MPC computation.

1. Send $(\texttt{init}, p)$ to $\mathcal{F}_{\text{offline}}$. Receive $[\alpha]_i$.

Input: For $x_1, \cdots, x_l \in \mathbb{F}_p$ inputs from a party P_j:

1. Invoke Π_{online}**Input** from [18], Fig. 1, for the inputs $x_1, \cdots, x_l$.

Add$(\llbracket \boldsymbol{x} \rrbracket_i, \llbracket \boldsymbol{y} \rrbracket_i)$**:**

1. Compute $\llbracket \boldsymbol{z} \rrbracket_i \coloneqq \llbracket \boldsymbol{x} + \boldsymbol{y} \rrbracket_i$ locally.

Multiply$(\llbracket \boldsymbol{x} \rrbracket_i, \llbracket \boldsymbol{y} \rrbracket_i)$**:**

1. Retrieve a Beaver triple $(\llbracket \boldsymbol{a} \rrbracket_i, \llbracket \boldsymbol{b} \rrbracket_i, \llbracket \boldsymbol{c} \rrbracket_i)$ from $\mathcal{F}_{\text{offline}}$.
2. Open $\llbracket \boldsymbol{u} \rrbracket_i \coloneqq \llbracket \boldsymbol{x} - \boldsymbol{a} \rrbracket_i$ and $\llbracket \boldsymbol{v} \rrbracket_i \coloneqq \llbracket \boldsymbol{y} - \boldsymbol{b} \rrbracket_i$.
3. Compute $\llbracket \boldsymbol{z} \rrbracket_i \coloneqq \llbracket \boldsymbol{x} \odot \boldsymbol{y} \rrbracket_i$ as in (6).

Convolve$(\llbracket \boldsymbol{x} \rrbracket_i, \llbracket \boldsymbol{y} \rrbracket_i)$**:**

1. Retrieve a convolution triple $(\llbracket \boldsymbol{a} \rrbracket_i, \llbracket \boldsymbol{f} \rrbracket_i, \llbracket \boldsymbol{c} \rrbracket_i)$ from $\mathcal{F}_{\text{offline}}$.
2. Open $\llbracket \boldsymbol{u} \rrbracket_i \coloneqq \llbracket \boldsymbol{x} - \boldsymbol{a} \rrbracket_i$ and $\llbracket \boldsymbol{v} \rrbracket_i \coloneqq \llbracket \boldsymbol{y} - \boldsymbol{f} \rrbracket_i$.
3. Compute $\llbracket \boldsymbol{z} \rrbracket_i \coloneqq \llbracket \boldsymbol{x} \odot \boldsymbol{y} \rrbracket_i$ as in (14).

DepthwiseConvolve$(\llbracket \boldsymbol{x} \rrbracket_i, \llbracket \boldsymbol{y} \rrbracket_i)$**:**

Proceed as in **Convolve** but with dconv2d instead of conv2d.

MatrixMultiply$(\llbracket X \rrbracket_i, \llbracket Y \rrbracket_i)$**:**

Proceed as in **Multiply** but with matrix multiplication as operation.

Output$(\llbracket \boldsymbol{x} \rrbracket_i)$**:**

1. All parties invoke $\Pi_{\llbracket \cdot \rrbracket}$**Check** from [31] on all opened values. If the check succeceeds the parties open $(\llbracket \boldsymbol{x} \rrbracket_i)$ and reconstruct the output $\boldsymbol{x}$; else abort. The parties use $\Pi_{\llbracket \cdot \rrbracket}$**Check.** to check $\boldsymbol{x}$ and accept the output if the check succeceeds, else they abort.

Figure 16: Protocol for the Online Phase at Party P_i

Functionality $\mathcal{F}_{\text{offline}}$

Init: On input $(\texttt{init}, p)$ from all parties.

1. For all parties P_i sample $[\alpha]_i \in \mathbb{F}_p$ and send it to P_i.

Macro GenerateMAC$(\boldsymbol{x}, \boldsymbol{\delta})$: This subroutine generates a MAC for $\boldsymbol{x}$ with adversarial offset $\boldsymbol{\delta}$.

1. Receive $[\boldsymbol{\gamma}]_i$ for corrupted parties P_i from the adversary.
2. Set $\boldsymbol{\gamma} \coloneqq \alpha \cdot \boldsymbol{x} + \boldsymbol{\delta}$.
3. Sample $[\boldsymbol{\gamma}]_i$ for honest P_i such that $\boldsymbol{\gamma} = \sum_{i=0}^{n-1} [\boldsymbol{\gamma}]_i$.
4. Send the honest parties' shares to their designated owner.

Triples: On input $(\texttt{triples}, \text{op}, D, D', D'')$ from all parties.

1. Receive $[\boldsymbol{a}]_i, \boldsymbol{\delta_a} \in \mathbb{F}_p^D$, $[\boldsymbol{b}]_i, \boldsymbol{\delta_b} \in \mathbb{F}_p^{D'}$, and $[\boldsymbol{c}]_i, \boldsymbol{\delta}, \boldsymbol{\delta_c} \in \mathbb{F}_p^{D''}$ for corrupted P_i from the adversary.
2. Sample $[\boldsymbol{a}]_i$ and $[\boldsymbol{b}]_i$ for honest P_i uniformly at random. Let $\boldsymbol{a} \coloneqq \sum_{i=0}^{n-1} [\boldsymbol{a}]_i$ and similarly for $\boldsymbol{b}$.
3. Let $\boldsymbol{c} \coloneqq \text{op}(\boldsymbol{a}, \boldsymbol{b}) + \boldsymbol{\delta}$.
4. Sample $[\boldsymbol{c}]_i$ for honest P_i uniformly at random such that $\boldsymbol{c} = \sum_{i=0}^{n-1} [\boldsymbol{c}]_i$.
5. Run GenerateMAC$(\boldsymbol{a}, \boldsymbol{\delta_a})$, GenerateMAC$(\boldsymbol{b}, \boldsymbol{\delta_b})$, and GenerateMAC$(\boldsymbol{c}, \boldsymbol{\delta_c})$.

Figure 17: Functionality for the Offline Phase

Reshare$(\langle \mathsf{x} \rangle)$**:** Compute $[\mathsf{x}]_i$. Also compute a new ciphertext $\langle \mathsf{y} \rangle$ with $\mathsf{y} = \mathsf{x}$.

1. Run **ZKP**$(\emptyset)$ to obtain $(\mathsf{m}_i, \langle \mathsf{m}_0 \rangle, \ldots, \langle \mathsf{m}_{n-1} \rangle)$. Define $\langle \mathsf{m} \rangle \coloneqq \sum_{j=0}^{n-1} \langle \mathsf{m}_j \rangle$ and $[\mathsf{m}]_i \coloneqq \mathsf{m}_i$.
2. Obtain d by decrypting $\langle \mathsf{x} - \mathsf{m} \rangle$ using **DistDec**.
3. Return $([\mathsf{x}]_i \coloneqq [\mathsf{m} + \mathsf{d}]_i \coloneqq [\mathsf{m}]_i + \mathsf{d} \cdot [1]_i, \langle \mathsf{y} \rangle \coloneqq \langle \mathsf{m} + \mathsf{d} \rangle)$.

Figure 18: Utilities for the Somewhat Homomorphic Offline Phase Used in $\Pi_{\text{offline-SHE}}$ (cf. Fig. 19) at Party P_i

to get a standalone protocol description, we did not use $\mathcal{F}_{\text{auth-linear}}$ in Fig. 7.

Theorem 6.1. *The offline protocol $\Pi_{\text{offline-LHE}}$ securely implements the ideal functionality $\mathcal{F}_{\text{auth-MPC}}$ in the $(\mathcal{F}_{\text{auth-linear}}, \mathcal{F}_{\text{commit}}, \mathcal{F}_{\text{rand}}, \mathcal{F}_{\text{setup}})$-hybrid model with rewinding if the used BGV cryptosystem achieves enhanced CPA-security [31].*

Proof. Compared to LowGear [31], our protocol exhibits the following changes: (i) Parties possess a single public-key private-key pair instead of key pairs $(\text{pk}_{i,j}, \text{sk}_{i,j})$ between each pair of parties (P_i, P_j). (ii) The key pairs are generated by a setup functionality $\mathcal{F}_{\text{setup}}$. (iii) Different encodings/packings and adapted ZKPs are used. The first two points are done to simplify the exposition of the protocol (a simulator can still decrypt messages encrypted under the public key of corrupted parties as the key generation is under control of the simulator in the simulation; the adversary can encrypt messages that – without access to the private key – only the intended recipient can decrypt). One could also modify our protocol and use the original LowGear design instead of changing (i) and (ii).

For point (iii), notice that the ZKPs only differ in that they additionally prove that encrypted messages are correctly packed. Hence this does not influence the security of the protocol. However, notice that the masks in **Multiply** are chosen to drown any information about the multiplication result and additionally hide any structure that could result from multiplying packed values, i.e., the outputs of the **Multiply** step are indistinguishable from what would be received in LowGear. Therefore, we can simply perform all steps of the simulation of LowGear's security proof to also prove our protocol secure. □

F.2 Somewhat Homomorphic Offline Phase

Before we continue with the proof of Theorem 6.2, we present the SHE-based offline protocol and necessary subprotocols in Figs. 13, 14, 18 and 19.

Theorem 6.2. *The offline protocol $\Pi_{\text{offline-SHE}}$ securely implements the ideal functionality $\mathcal{F}_{\text{offline}}$ in the $(\mathcal{F}_{\text{commit}}, \mathcal{F}_{\text{rand}}, \mathcal{F}_{\text{setup}})$-hybrid model if the used BGV cryptosystem achieves CPA-security and has an algorithm for meaningless public key generation [18].*

Proof. The offline protocol $\Pi_{\text{offline-SHE}}$ is structured like the offline phase of HighGear [31] or TopGear [3]. The use of different encodings/packings and the adapted ZKPs are the only difference compared to these protocols. As already mentioned above, proving the necessary properties for the ZKPs can be done by simply

Protocol $\Pi_{\text{offline-SHE}}$

Triples(op, D, D')**:** Generate a triple for the bilinear map op.

1. Run **ZKP**$([..N) \setminus \text{image}(\text{mapi}(\text{op}, D, D', \cdot\ )))$ to obtain $(\text{b}_i, \langle \text{b}_0 \rangle, \dots, \langle \text{b}_{n-1} \rangle)$. Run **ZKP**$([..N) \setminus \text{image}(\text{mapf}(\text{op}, D, D', \cdot\ )))$ twice to obtain $(\text{g}_i, \langle \text{g}_0 \rangle, \dots, \langle \text{g}_{n-1} \rangle)$ and $(\text{g}'_i, \langle \text{g}'_0 \rangle, \dots, \langle \text{g}'_{n-1} \rangle)$. Define $\langle \text{b} \rangle \coloneqq \sum_{j=0}^{n-1} \langle \text{b}_j \rangle$, $\langle \text{g} \rangle \coloneqq \sum_{j=0}^{n-1} \langle \text{g}_j \rangle$, $\langle \text{g}' \rangle \coloneqq \sum_{j=0}^{n-1} \langle \text{g}'_j \rangle$, $[\boldsymbol{a}]_i \coloneqq \text{unpacki}(\text{op}, D, D', \text{b}_i)$, $[\boldsymbol{f}]_i \coloneqq \text{unpackf}(\text{op}, D, D', \text{g}_i)$, $[\boldsymbol{f'}]_i \coloneqq \text{unpackf}(\text{op}, D, D', \text{g}'_i)$.
2. Compute $\langle \text{b} \cdot \text{g} \rangle$ and $\langle \text{b} \cdot \text{g}' \rangle$ with somewhat homomorphic encryption.
3. Run **Reshare** with $\langle \text{b} \cdot \text{g} \rangle$ and $\langle \text{b} \cdot \text{g}' \rangle$ to obtain $[\text{b} \cdot \text{g}]_i$, $\langle \text{d} \rangle$, $[\text{b} \cdot \text{g}']_i$, and $\langle \text{d}' \rangle$. Define $[\boldsymbol{c}]_i \coloneqq \text{unpackr}(\text{op}, D, D', [\text{b} \cdot \text{g}]_i)$ and $[\boldsymbol{c'}]_i \coloneqq \text{unpackr}(\text{op}, D, D', [\text{b} \cdot \text{g}']_i)$.
4. Compute $\langle \alpha \cdot \text{b} \rangle$, $\langle \alpha \cdot \text{g} \rangle$, $\langle \alpha \cdot \text{g}' \rangle$, $\langle \alpha \cdot \text{d} \rangle$, and $\langle \alpha \cdot \text{d}' \rangle$.
5. Run **ShareDec** with $\langle \alpha \cdot \text{b} \rangle$, $\langle \alpha \cdot \text{g} \rangle$, $\langle \alpha \cdot \text{g}' \rangle$, $\langle \alpha \cdot \text{d} \rangle$, and $\langle \alpha \cdot \text{d}' \rangle$ to obtain $[\alpha \cdot \text{b}]_i$, $[\alpha \cdot \text{g}]_i$, $[\alpha \cdot \text{g}']_i$, $[\alpha \cdot \text{d}]_i$, and $[\alpha \cdot \text{d}']_i$. Unpack these shares to get $[\alpha \boldsymbol{a}]_i$, $[\alpha \boldsymbol{f}]_i$, $[\alpha \boldsymbol{f'}]_i$, $[\alpha \boldsymbol{c}]_i$, and $[\alpha \boldsymbol{c'}]_i$, respectively.
6. Return **Sacrifice**$(\text{op}, [\![\boldsymbol{a}]\!]_i, [\![\boldsymbol{f}]\!]_i, [\![\boldsymbol{f'}]\!]_i, [\![\boldsymbol{c}]\!]_i, [\![\boldsymbol{c'}]\!]_i)$.

Figure 19: Protocol for the Somewhat Homomorphic Offline Phase at Party P_i

following the proof in [3]. Similarly, the full security proof follows the blueprint of a SHE-based SPDZ-like offline phase [3, 18, 31], where the simulator simply has to be adapted to apply the packing method. Notice that the shares in the output of **Reshare** and **ShareDec** appear uniformly random in our protocol (independently of the packing method used), as well as in SPDZ, as the masks are uniformly random. □

F.3 Linear Homomorphic Offline Phase Utilizing Bian et al.'s Parallel Convolution Packing

In this section, we investigate the packing of [6] (cf. Appendix C.2) as part of an offline phase. To use this convolution packing, Bian et al. modified the (private-key version of) the BFV encryption scheme [7, 22] in [6] to support homomorphic matrix-vector multiplication. A similar modification for (public-key) BGV is possible in a straightforward way. We call the encryption algorithm of the modified BGV instance expandenc and we have expandenc$(\cdot) = \text{expand}(\text{enc}(\cdot))$ (cf. Appendix A.2 for details). The new encryption scheme can then be used to perform matrix-vector multiplication with encrypted vectors and plaintext matrices – instead of polynomial multiplications (or negacyclic convolutions). The respective packing method allows us to encode multiple convolutions in a single matrix multiplication (cf. Appendix C.2) in an actively secure way. Before we describe our offline protocol in Appendix F.3.2, we investigate the combination of the packing from [6] and our new encryption scheme w.r.t. active security.

F.3.1 Active Security with the Modified BGV Scheme. Recall that LowGear-style protocols use a pairwise subprotocol that multiplies a ciphertext and a plaintext and drowns the result with an encrypted mask (see Step 1.2. of **Multiply** in Fig. 6). The straightforward extension that simply uses the new encryption scheme expandenc$(\cdot) = \text{expand}(\text{enc}(\cdot))$ comes with a security issue: due to the underlying packing the matrices and vectors come with a certain structure. This structure changes under the plaintext-ciphertext matrix multiplication. Hence the product and the mask no longer have the same structure. In particular, the mask no longer drowns all information in the product and information on the plaintext matrix (the structure or the values) are leaked. Obviously not masking the product at all, as in [6], is also not viable as it directly leaks information about the plaintext matrix.

Instead, we propose a (secure) alternative encryption expandenc′ for the mask to be used in a LowGear-style protocol. We remark that our construction might be of independent interest for other protocols. Formally, we get the following two security guarantees:

Theorem F.1. *Let* enc′ *be the encryption with drowning noise from LowGear (cf. [31] and Appendix A.1.1). The encryption with drowning noise*

$$\text{expandenc}'_{\text{pk}}(\boldsymbol{z}, \boldsymbol{r}) \coloneqq \sum_{k=0}^{N-1} \Delta_k \cdot \text{expand}(\text{enc}'_{\text{pk}}(\boldsymbol{z}, \boldsymbol{r}[k])) \tag{56}$$

statistically hides the noise of $\boldsymbol{M} \cdot \text{expandenc}_{\text{pk}}(\boldsymbol{y})$ *for arbitrary* $\boldsymbol{M} \in \mathbb{Z}_p^{N \times N}$, $\boldsymbol{y} \in \mathbb{Z}_p^N$, $\boldsymbol{r}[k]$ *is encryption randomness for* enc′, $\boldsymbol{z} \in \mathbb{Z}_p^N$ *is sampled uniformly at random, and* $\Delta_k[i, j] = \delta_{k-i} \cdot \delta_j$.

Proof. Let $\boldsymbol{r}[k] = (\boldsymbol{u}[k], \boldsymbol{v}[k], \boldsymbol{w}[k])$. Then, we can rewrite the decryption noise of (56) via the step of the proof of Theorem A.3 as

$$\begin{aligned}
&\text{expandpartdec}_{\text{sk}}(\text{expandenc}'_{\text{pk}}(\boldsymbol{z}, \boldsymbol{r})) \\
&= \sum_{k=0}^{N-1} \Delta_k \boldsymbol{z} + p\Delta_k(\boldsymbol{e} \bar{*} \boldsymbol{u}[k]) + p\Delta_k \boldsymbol{v}[k] - p\Delta_k(\boldsymbol{w}[k] \bar{*} \boldsymbol{s}) \\
&= \boldsymbol{z} + \sum_{k=0}^{N-1} p\Delta_k(\boldsymbol{e} \bar{*} \boldsymbol{u}[k]) + p\Delta_k \boldsymbol{v}[k] - p\Delta_k(\boldsymbol{w}[k] \bar{*} \boldsymbol{s}).
\end{aligned} \tag{57}$$

Analyzing the bounds of this, we get

$$\begin{aligned}
&\|\text{expandpartdec}_{\text{sk}}(\text{expandenc}'_{\text{pk}}(\boldsymbol{z}, \boldsymbol{r}))\|_\infty \\
&= \|\text{partdec}_{\text{sk}}(\text{enc}'_{\text{pk}}(\boldsymbol{z}, \boldsymbol{r}[k]))\|_\infty
\end{aligned}$$

for any $k \in [..N)$. Finally, note that for arbitrary $\boldsymbol{y} \in \mathbb{Z}_p^N$, $\boldsymbol{M} \in \mathbb{Z}_p^{N \times N}$, $\text{x}, \text{y} \in \mathcal{R}_p$

$$\begin{aligned}
&\|\text{expandpartdec}_{\text{sk}}(\boldsymbol{M} \cdot \text{expandenc}_{\text{pk}}(\boldsymbol{y}))\|_\infty \\
&= \|\text{partdec}_{\text{sk}}(\text{x} \cdot \text{enc}_{\text{pk}}(\text{y}))\|_\infty
\end{aligned}$$

as we upper-bound both the result of a multiplication of a value $\boldsymbol{y}$/y with $\boldsymbol{M}$ (i.e., $\|\boldsymbol{M} \cdot \boldsymbol{y}\|_\infty$) and with x (i.e., $\|\text{x} \cdot \text{y}\|_\infty$ where the multiplication is a polynomial multiplication) by $p \cdot N \cdot \|\boldsymbol{y}\|_\infty$ and $p \cdot N \cdot \|\text{y}\|_\infty$, respectively. □

Theorem F.2. *The encryption with drowning noise* expandenc′ *for the modified BGV scheme computationally hides* $\boldsymbol{M} \cdot \text{expandenc}_{\text{pk}}(\boldsymbol{y})$ *(for* expandenc′, $\boldsymbol{M}$, $\boldsymbol{y}$, *etc. as in Theorem F.1).*

Proof. Let $(\boldsymbol{c}_0, \boldsymbol{C}_1) := \langle\!\langle \boldsymbol{z} \rangle\!\rangle_{\mathrm{pk}} = \mathrm{expandenc}'_{\mathrm{pk}}(\boldsymbol{z}, \boldsymbol{r})$ and $\boldsymbol{r}[k] = (\boldsymbol{u}[k], \boldsymbol{v}[k], \boldsymbol{w}[k])$. For $\boldsymbol{c}_0$, we have

$$\begin{aligned}\boldsymbol{c}_0 &= \sum_{k=0}^{N-1} \Delta_k \cdot (\boldsymbol{b} \circledast \boldsymbol{u}[k] + p \cdot \boldsymbol{v}[k] + \boldsymbol{z}) \\ &= \boldsymbol{z} + \sum_{k=0}^{N-1} \Delta_k \cdot (\boldsymbol{b} \circledast \boldsymbol{u}[k] + p \cdot \boldsymbol{v}[k]),\end{aligned}$$

i.e., $\boldsymbol{z}$ is masked with (parts of) RLWE samples. For this, note that $\boldsymbol{b} \circledast \boldsymbol{u}[k] + p \cdot \boldsymbol{v}[k]$ is indistinguishable from uniformly random (if $\boldsymbol{b}$ is uniformly random or indistinguishable from it – as it is for every party expect the one holding sk). The multiplication with Δ_k simply selects the k-th element of the k-th RLWE sample. With the sum over all k, we get that the k-th element of $\boldsymbol{z}$ is masked with the k-th element of the k-th RLWE sample which is indistinguishable from random. Therefore, $\boldsymbol{c}_0$ is indistinguishable from uniformly random (based on the hardness of the RLWE problem).

For $\boldsymbol{C}_1$, we notice that $\boldsymbol{C}_1 = \sum_{k=0}^{N-1} \Delta_k \cdot \overline{\mathrm{circ}}(\boldsymbol{a} \circledast \boldsymbol{u}[k] + p \cdot \boldsymbol{w}[k])$ and thus

$$\boldsymbol{C}_1[k, j] = \overline{\mathrm{circ}}(\boldsymbol{a} \circledast \boldsymbol{u}[k] + p \cdot \boldsymbol{w}[k])[k, j],$$

i.e., column 0 is simply the RLWE sample $\boldsymbol{a} \circledast \boldsymbol{u}[0] + p \cdot \boldsymbol{w}[0]$ and each other column k is a negacyclicly rotated RLWE sample $\boldsymbol{a} \circledast \boldsymbol{u}[k] + p \cdot \boldsymbol{w}[k]$. The first RLWE sample and all the rotated samples are indistinguishable from uniformly random (based on the the hardness of the RLWE problem – if $\boldsymbol{a}$ is sampled uniformly at random) and thus, the whole matrix $\boldsymbol{C}_1$ is indistinguishable from a uniformly random matrix. □

Remark F.1. Note that we only use the k-th coefficient of $\boldsymbol{v}[k]$ in the above proof. Therefore, we could give an alternative formulation

$$\mathrm{expandenc}'_{\mathrm{pk}}(\boldsymbol{z}, \boldsymbol{r}) := \sum_{k=0}^{N-1} \Delta_k \cdot \mathrm{expand}(\mathrm{enc}'_{\mathrm{pk}}(\boldsymbol{z}, (\boldsymbol{u}[k], \boldsymbol{v}, \boldsymbol{w}[k])))$$

instead of (56), where $\boldsymbol{r} = (\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w})$ with $(\boldsymbol{u}[k], \boldsymbol{v}, \boldsymbol{w}[k])$ being (sampled like) valid encryption randomness for enc′. This saves sampling some randomness for drowning but has the same computational complexity (and security). For simplicity, we only use (56) in the rest of this work.

With this (secure) drowning encryption, we can construct a LowGear-style protocol (similar to the LHE protocol described in Section 6.2) but based on matrix-vector products instead of polynomial multiplication, as well as the modified BGV scheme. This is outlined next.

F.3.2 Offline Protocol Utilizing the Modified BGV Scheme. In Figure 20, you can find our linear homomorphic offline phase that utilizes secure matrix-vector products instead of secure polynomial multiplication. The protocol adds variants of **Multiply** (cf. Fig. 6) and **Triples** (cf. Fig. 7) using these matrix operations: **MMultiply** and **MTriples**.

This offline phase mostly mirrors the linear homomorphic offline phase of Section 6.2 but with different encodings and homomorphic matrix-vector multiplications instead of polynomial multiplications of ciphertexts. Note that we still use the standard BGV scheme for ZKPs and authentication since the previously described modifications to BGV are not needed for these subprotocols and would only lead to an additional overhead from the use of expanded ciphertexts. Indeed, we can simply perform the standard ZKPs and expand the ciphertexts later to send less data and reuse existing implementations. Also, the multiplication with encrypted shares of α does not require the properties of the modified BGV scheme and can thus fall back to the same techniques as in Section 6.2.

Protocol $\Pi_{\text{modified-offline-LHE}}$

Note that we use the modified BGV scheme (cf. Appendix A.2) and normal BGV encryption below.

MMultiply$(\boldsymbol{A}_i, \boldsymbol{b}_i, \langle \boldsymbol{b}_0 \rangle_{\mathrm{pk}_0}, \ldots, \langle \boldsymbol{b}_{n-1} \rangle_{\mathrm{pk}_{n-1}})$**:** Compute $[\boldsymbol{c}]_i$ such that $\boldsymbol{c} = (\sum_{j=0}^{n-1} \boldsymbol{A}_j) \cdot (\sum_{j=0}^{n-1} \boldsymbol{b}_j)$.

1. For $j \in [..n) \setminus \{i\}$ do the following (in parallel).
   1.1. Sample an uniformly random $\boldsymbol{m}_{i,j} \in \mathbb{Z}_p^N$.
   1.2. Compute $\langle\!\langle \boldsymbol{c}_{i,j} \rangle\!\rangle_{\mathrm{pk}_j} = \boldsymbol{A}_i \cdot \mathrm{expand}(\langle \boldsymbol{b}_j \rangle_{\mathrm{pk}_j}) - \mathrm{expandenc}'_{\mathrm{pk}_j}(\boldsymbol{m}_{i,j})$ where expandenc′ is encryption with large *drowning* noise and adaptions for the modified BGV scheme (larger than normal encryption randomness; cf. Appendices A.1.1 and F.3.1).
   1.3. Send $\langle\!\langle \boldsymbol{c}_{i,j} \rangle\!\rangle_{\mathrm{pk}_j}$ to P_j and receive $\langle\!\langle \boldsymbol{c}_{j,i} \rangle\!\rangle_{\mathrm{pk}_i}$ in return.
   1.4. Decrypt $\langle\!\langle \boldsymbol{c}_{j,i} \rangle\!\rangle_{\mathrm{pk}_i}$ to $\boldsymbol{c}_{j,i}$ with $\mathrm{expanddec}_{\mathrm{sk}_i}$.
2. Compute $[\boldsymbol{c}]_i = \boldsymbol{A}_i \cdot \boldsymbol{b}_i + \sum_{j \neq i} (\boldsymbol{c}_{j,i} + \boldsymbol{m}_{i,j})$.

MTriples(op, D, D')**:** Generate a triple for the bilinear map op.

1. Run **ZKP**$([..N) \setminus \mathrm{image}(\mathrm{mapi}(\mathrm{op}, D, D', \cdot)))$ to obtain $(\boldsymbol{b}_i, \langle \boldsymbol{b}_0 \rangle_{\mathrm{pk}_0}, \ldots, \langle \boldsymbol{b}_{n-1} \rangle_{\mathrm{pk}_{n-1}})$. Sample uniformly random $[\boldsymbol{f}]_i, [\boldsymbol{f}']_i \in \mathbb{Z}_p^{D'}$. Define $[\boldsymbol{a}]_i := \mathrm{unpacki}(\mathrm{op}, D, D', \boldsymbol{b}_i)$, $\boldsymbol{G}_i := \mathrm{packf}(\mathrm{op}, D, D', [\boldsymbol{f}]_i)$, $\boldsymbol{G}'_i := \mathrm{packf}(\mathrm{op}, D, D', [\boldsymbol{f}']_i)$, $\boldsymbol{g}_i = \mathrm{packf}(\mathrm{op}, D, D', [\boldsymbol{f}]_i)$, $\boldsymbol{g}'_i = \mathrm{packf}(\mathrm{op}, D, D', [\boldsymbol{f}']_i)$.
2. Run **MMultiply** with $\boldsymbol{G}_i$, $\boldsymbol{b}_i$, and the ciphertexts for $\boldsymbol{b}_j$ to obtain $[\boldsymbol{c}]_i$. Analogously, obtain $[\boldsymbol{c}']_i$ for $\boldsymbol{G}'_i$, $\boldsymbol{b}_i$, and the ciphertexts.
3. Run **Multiply** with $\boldsymbol{b}_i$, $[\alpha]_i$, and the ciphertexts of the MAC key shares to obtain $[\alpha \cdot \boldsymbol{b}]_i$. Analogously, obtain $[\alpha \cdot \boldsymbol{g}]_i$, $[\alpha \cdot \boldsymbol{g}']_i$, $[\alpha \cdot \boldsymbol{c}]_i$, and $[\alpha \cdot \boldsymbol{c}']_i$. Unpack these shares to get $[\alpha \boldsymbol{a}]_i$, $[\alpha \boldsymbol{f}]_i$, $[\alpha \boldsymbol{f}']_i$, $[\alpha \boldsymbol{c}]_i$, and $[\alpha \boldsymbol{c}']_i$, respectively.
4. Return **Sacrifice**$(\mathrm{op}, [\![\boldsymbol{a}]\!]_i, [\![\boldsymbol{f}]\!]_i, [\![\boldsymbol{f}']\!]_i, [\![\boldsymbol{c}]\!]_i, [\![\boldsymbol{c}']\!]_i)$.

Figure 20: Extension to the Protocol for the Linear Homomorphic Offline Phase (cf. Fig. 7) at Party P_i

Similar to Theorem 6.1, the following theorem captures the security of our modified LHE-based theorem. The required functionalities are the same as for Theorem 6.1.

Theorem F.3. *The offline protocol* $\Pi_{\text{modified-offline-LHE}}$ *securely implements the ideal functionality* $\mathcal{F}_{\text{auth-MPC}}$ *in the* $(\mathcal{F}_{\text{auth-linear}}, \mathcal{F}_{\text{commit}}, \mathcal{F}_{\text{rand}}, \mathcal{F}_{\text{setup}})$*-hybrid model with rewinding if the used BGV cryptosystem achieves enhanced CPA-security [31].*

Proof. The only difference between **MTriples** in Fig. 20 and **Triples** (in $\Pi_{\text{offline-LHE}}$; Fig. 7) is the use of **MMultiply** instead of **Multiply**. These protocols only differ in the use of classical BGV or the modified BGV scheme. Our results in Appendix A.2 and Theorems F.1 and F.2 show however that both schemes come

Table 4: Runtime Results for Matrix Multiplications of Square 128×128 Matrices in the Online Phase (in Seconds). These results are used as an approximation for the online phase of [14]. The results are given for $c \cdot c'$ multiplications, where c rounds of c' (parallel) multiplications are computed. The layers and settings correspond to Tables 2 and 3.

		2 Party Setting	
c Layer	c'	LAN	WAN
1 conv1@7x7	196	61.38	69.13
3 conv2@3x3	125	117.76	130.93
4 conv3@3x3	64	80.95	91.13
6 conv4@3x3	72	136.69	153.31
3 conv5@3x3	144	135.88	150.28
1 dconv2d ($h \in \{7, 9, 11\}$)	144	45.29	50.09
1 dconv2d ($h \in \{13, 15\}$)	288	90.24	99.51
1 dconv2d ($h \in \{17, 19\}$)	432	134.98	147.74
1 dconv2d ($h = 21$)	576	179.87	195.85
1 dconv2d ($h \in \{23, 25\}$)	720	224.96	244.93
1 dconv2d ($h = 50$)	2880	899.82[a]	979.72[a]
1 dconv2d ($h = 120$)	16272	5084.01[a]	5535.40[a]
1 dconv2d ($h = 240$)	64800	20246.06[a]	22043.63[a]

[a] extrapolated from results with $c' = 720$

with the same security guarantees. Hence the security of our protocol $\Pi_{\text{modified-offline-LHE}}$ follows exactly as in the proof of Theorem 6.1. □

G IMPLEMENTATION AND EVALUATION (CONTINUED)

Here, we give supplementary information for our evaluation (cf. Section 7). Firstly, note that the optimized distributed decryption to get shares directly (**ShareDec** in Fig. 18) of [31] is not implemented in MP-SPDZ [29] as of the time of our implementation.[11] Secondly, we use a statistical security parameter of sec = 40 and a prime of length $\log p = 128$. This implies that our protocols (in the LowGear-variant) have the same BGV parameters as standard LowGear ($N = 8192$ and ciphertext modulus of the same size as LowGear). For HighGear, a ciphertext modulus that is 9 bit larger than standard HighGear (and $N = 16384$ as for HighGear) is necessary as we want to compute (up to) 512 ciphertext additions.

Note that some results for our protocols and LowGear/HighGear are extrapolated from our experiments as MP-SPDZ does not support (very) large tensors. Another reason why we extrapolate results is to finish the experiments in a reasonable time frame. Therefore, we extrapolate the findings from our experiments for some runs of LowGear/HighGear and also for our protocols for large depthwise convolutions. To obtain separate timings for the offline and online phase of [14], we used their total (online and offline) results and subtracted timings obtained from experiments of our own for the online phase with a suitable number of matrix multiplications (cf. Table 4). As the difference in CPU performance of our machines and theirs is not large (ours are around 6 % faster) and the offline phase is considerably slower than the online phase, this is a reasonable approximation. However, the tables for the overall (online and offline) performance are available as well. Next, we present more details that complement Section 7.

[11] We based our implementation on commit `505d4838c18394e8bb87bc5bae5a8b9cc00d65ad` of https://github.com/data61/MP-SPDZ.

Table 5: Convolutions in ResNet50. The convolutions in ResNet50 [23] can be classified by shape for each layer conv$i$, e.g., the first column below corresponds to conv1@7x7 in Table 2 and the third column to conv2@3x3. For each conv2d operation of $1 \times h \times h \times d$ images and $d' \times d \times h' \times h'$ filters (with stride s), a corresponding matrix multiplication of a $k \times l$ matrix and a $l \times d'$ matrix can be constructed. In [14], these are further decomposed into c' matrix multiplications of 128×128 matrices. Each type of convolution is used c times in ResNet50, leading to a total of 3190 matrix multiplications in [14] to compute all convolutions. Convolutions that are relevant for Table 2 and other tables are marked in light gray.

		Convolution Dimensions					Matrix Dimensions		Matrix Count	
i	c	h	d	h'	d'	s	k	l	c'	$c \cdot c'$
1	1	224	3	7	64	2	12544	147	196	196
2	1	56	64	1	64	1	3136	64	25	25
2	3	56	64	3	64	1	3136	576	125	375
2	4	56	64	1	256	1	3136	64	50	200
2	2	56	256	1	64	1	3136	256	50	100
3	1	56	256	1	128	2	784	256	14	14
3	4	28	128	3	128	1	784	1152	64	252
3	4	28	128	1	512	1	784	128	28	112
3	1	56	256	1	512	2	784	256	56	56
3	3	28	512	1	128	1	784	512	28	84
4	1	28	512	1	256	2	196	512	16	16
4	6	14	256	3	256	1	196	2304	72	432
4	6	14	256	1	1024	1	196	256	32	192
4	1	28	512	1	1024	2	196	512	64	64
4	5	14	1024	1	256	1	196	1024	32	160
5	1	14	1024	1	512	2	49	1024	32	32
5	3	7	512	3	512	1	49	4608	144	432
5	3	7	512	1	2048	1	49	512	64	192
5	1	14	1024	1	2048	2	49	1024	128	128
5	2	7	2048	1	512	1	49	2048	64	128
									Total	**3190**[a]

[a] [14] gives 3298 as the total number of matrix multiplications

Table 5 shows the parameters for all convolutions in ResNet50,[12] as well as the corresponding matrix multiplication that emulates the convolution. We also show the number of matrix multiplication one would use with [14] for each convolution. This corresponds to the number multiplication for square 128×128 matrices that are required to emulate the multiplication of a $k \times l$ and a $l \times d'$ matrix.

LowGear-Style Protocols. To complement the result for the runtime in the offline phase (Table 2 in Section 7), we give the overall runtime

[12] Analyzed based on the model from https://github.com/onnx/models/blob/main/vision/classification/resnet/model/resnet50-v1-7.onnx as of 2022-11-03.

Table 6: Overall (Online and Offline) Runtime Results for conv2d **Operations (in Seconds).** Our protocols here are LowGear-based (cf. Section 6.2). Runtime is given for ResNet50 convolution layers as in Table 2.

c Layer	LowGear [31]	Matmul [14]	Our Simple Packing	Our General. Huang et al. Packing
	2 Party LAN Setting			
1 conv1@7x7	21765[b]	7076	400[b]	401[b]
3 conv2@3x3	64021[c]	13538	1961	1962
4 conv3@3x3	84277[c]	9097	2471	2678
6 conv4@3x3	121409[c]	15595	4274	4053
3 conv5@3x3	57265[c]	15595	3676[b]	4053[b]
Total	**348736**	**60901**	**12783**	**13146**
	2 Party WAN Setting			
1 conv1@7x7	53733[b]	7477	672[b]	668[b]
3 conv2@3x3	155454[c]	14306	3353	3337
4 conv3@3x3	204759[c]	9614	4160	4496
6 conv4@3x3	294773[c]	16481	7146	6756
3 conv5@3x3	139118[c]	16481	6167[b]	6788[b]
Total	**847838**	**64359**	**21497**	**22045**

[a] column extrapolated from the runtime results in [14] using Table 5
[b] extrapolated from results with halved output depth
[c] extrapolated from results with output depth $d' = 2$ and $c = 1$

Table 7: Communication Costs for conv2d **Operations in the Offline Phase (in MB).** Our protocols here are LowGear-based (cf. Section 6.2). Costs are given for ResNet50 convolution layers as in Table 2.

c Layer	LowGear [31]	Matmul [14]	Our Simple Packing	Our General. Huang et al. Packing
	2 Party Setting			
1 conv1@7x7	137659[a]	2442	3967[a]	3967[a]
3 conv2@3x3	402325[b]	4673	20372	20157
4 conv3@3x3	528985[b]	3140	25759	26938
6 conv4@3x3	759956[b]	5383	45678	41470
3 conv5@3x3	357631[b]	5383	42896[a]	42104[a]
Total	**2186556**	**21020**	**138672**	**134635**

[a] extrapolated from results with halved output depth
[b] extrapolated from results with output depth $d' = 2$ and $c = 1$

for our LowGear-style protocols compared to the related work in Table 6. Additionally, the computation cost can be seen in Table 7 for the offline phase and in Table 8 for the overall (online and offline) cost.

HighGear-Style Protocols. To evaluate our protocols for a larger number of parties, we implemented the HighGear variants of our protocols. The results are given for $n = 4$ parties. We do not compare our HighGear variants to [14] as their only provide results for

Table 8: Overall (Online and Offline) Communication Costs for conv2d **Operations (in MB).** Our protocols here are LowGear-based (cf. Section 6.2). Costs are given for ResNet50 convolution layers as in Table 2.

c Layer	LowGear [31]	Matmul [14]	Our Simple Packing	Our General. Huang et al. Packing
	2 Party Setting			
1 conv1@7x7	141377[a]	2545	3972[a]	3972[a]
3 conv2@3x3	413161[b]	4869	20383	20168
4 conv3@3x3	543086[b]	3272	25775	26954
6 conv4@3x3	780088[b]	5609	45740	41531
3 conv5@3x3	366716[b]	5609	43012[a]	42219[a]
Total	2244429	21904	138881	134844

[a] extrapolated from results with halved output depth
[b] extrapolated from results with output depth $d' = 2$ and $c = 1$

Table 9: Runtime Results for conv2d **Operations in the Offline Phase (in Seconds).** Our protocols here are HighGear-based (cf. Section 6.3). Runtime is given for ResNet50 convolution layers as in Table 2.

c Layer	HighGear [3, 31]	Our Simple Packing	Our General. Huang et al. Packing
	4 Party WAN Setting		
1 conv1@7x7	127999[a]	3305[a]	3314[a]
3 conv2@3x3	373928[b]	23068	23396
4 conv3@3x3	489932[b]	29543	32172
6 conv4@3x3	697499[b]	49684	52305
3 conv5@3x3	323489[b]	44320[a]	53457[a]
Total	**2012848**	**149920**	**164644**

[a] extrapolated from results with halved output depth
[b] extrapolated from results with output depth $d' = 2$ and $c = 1$

Table 10: Overall (Online and Offline) Runtime Results for conv2d **Operations (in Seconds).** Our protocols here are HighGear-based (cf. Section 6.3). Runtime is given for ResNet50 convolution layers as in Table 2.

c Layer	HighGear [3, 31]	Our Simple Packing	Our General. Huang et al. Packing
	4 Party WAN Setting		
1 conv1@7x7	129019[a]	3327[a]	3335[a]
3 conv2@3x3	376894[b]	23121	23449
4 conv3@3x3	493809[b]	29611	32239
6 conv4@3x3	702893[b]	49788	52410
3 conv5@3x3	325924[b]	44400[a]	53537[a]
Total	**2028539**	**150246**	**164969**

[a] extrapolated from results with halved output depth
[b] extrapolated from results with output depth $d' = 2$ and $c = 1$

Table 11: Communication Costs for conv2d **Operations in the Offline Phase (in MB).** Our protocols here are HighGear-based (cf. Section 6.3). Costs are given for ResNet50 convolution layers as in Table 2.

c Layer	HighGear [3, 31]	Our Simple Packing	Our General. Huang et al. Packing
	4 Party Setting		
1 conv1@7x7	476682[a]	16857[a]	16857[a]
3 conv2@3x3	1392396[b]	139934	140880
4 conv3@3x3	1820852[b]	179672	192660
6 conv4@3x3	2580318[b]	303406	308276
3 conv5@3x3	1186961[b]	272383[a]	315551[a]
Total	**7457208**	**912251**	**974224**

[a] extrapolated from results with halved output depth
[b] extrapolated from results with output depth $d' = 2$ and $c = 1$

Table 12: Overall (Online and Offline) Communication Costs for conv2d **Operations (in MB).** Our protocols here are HighGear-based (cf. Section 6.3). Costs are given for ResNet50 convolution layers as in Table 2.

c Layer	HighGear [3, 31]	Our Simple Packing	Our General. Huang et al. Packing
	4 Party Setting		
1 conv1@7x7	482260[a]	16865[a]	16865[a]
3 conv2@3x3	1408649[b]	139951	140897
4 conv3@3x3	1842004[b]	179696	192684
6 conv4@3x3	2610517[b]	303498	308368
3 conv5@3x3	1200588[b]	272556[a]	315724[a]
Total	**7544019**	**912565**	**974538**

[a] extrapolated from results with halved output depth
[b] extrapolated from results with output depth $d' = 2$ and $c = 1$

$n = 2$ parties. Table 9 shows the benchmark results for the packing schemes and SPDZ with HighGear-based protocols (similar to Table 2 for LowGear). Again, we can see that the convolution packing methods outperform the classical SPDZ approach. The corresponding overall runtime can be found in Table 10. We also give the communication costs in Tables 11 and 12.

Depthwise Convolutions. Tables 13 and 15 show additional results for our depthwise convolution experiments (Table 3). The first expands on Table 3 by giving the results for additional image sizes. One can clearly see that the (not depthwise) convolution packing methods (simple packing and generalization of Huang et al.'s packing) have essentially the same complexity for all small images as only one output channel can be computed at once. The depthwise packing can instead compute multiple results at once. The LowGear protocol is for very small image sizes most efficient (or similarly efficient to depthwise packing) as the packing method is not perfectly optimal w.r.t. the usage of ciphertexts slots. The computational

Table 13: Runtime Results for dconv2d **Operations in the Offline Phase (in Seconds).** Our protocols here are LowGear-based (cf. Section 6.2). Runtime is given for $1 \times h \times h \times 512$ images and $512 \times 3 \times 3$ filters. This table extends Table 3.

h	LowGear [31]	Matmul [14][a]	Our Simple Packing	Our Gen. Huang et al. Packing	Our Depthw. Packing
	2 Party LAN Setting				
7	37	5153	321	322	55
9	64	5153	321	323	74
11	96	5153	322	321	91
13	132	10307	321	321	92
15	177	10307	321	321	116
17	231	15460	321	322	117
19	288	15460	322	321	117
21	352	20614	322	321	165
23	424	25767	321	322	165
25	504	25767	323	322	165
50[b]	2137	103068	352	352	343
120[b]	12279	582335	804	809	1086
240[b]	49398	2319034	2657	2667	3704
	2 Party WAN Setting				
7	90	5444	637	636	121
9	157	5444	636	636	155
11	235	5444	635	635	188
13	324	10888	635	636	189
15	435	10888	636	638	234
17	568	16333	637	637	235
19	712	16333	637	635	235
21	867	21779	635	636	327
23	1045	27223	636	637	325
25	1244	27223	637	637	327
50[b]	5155	108892	692	688	664
120[b]	29764	615241	1443	1438	1907
240[b]	119842	2450076	4520	4516	6227

[a] column extrapolated from the runtime results in [14]
[b] row extrapolated from results with depth $d = 32$ (except for matmul runtime)

overhead from not using convolution triples is for these small images smaller than the overhead of the packing. However, this is no longer the case for images of size 11×11 (or even 9×9 in the WAN setting). The matrix-based approach [14] computes 128×128 matrix multiplications that are considerably oversized for the small images. Additionally, a depthwise convolution is here emulated by the same matrix multiplication as for a conv2d with $d = d'$. This needs less matrix multiplications than using one matrix multiplication per channel but still leaves much of the matrices unused. The corresponding overall runtime (online and offline) can be found in Table 14. The communication cost can be found in Tables 17 and 18.

Table 15 additionally shows benchmarks for our offline phase based on Bian et al.'s packing. The linear homomorphic offline phase for this packing seems very slow compared to the other packing methods. This is mostly due to the factor $N = 8192$ factor

Table 14: Overall (Online and Offline) Runtime Results for dconv2d **Operations (in Seconds).** Our protocols here are LowGear-based (cf. Section 6.2). Runtime is given for convolutions of $1 \times h \times h \times 512$ images with $512 \times 3 \times 3$ filters.

h	LowGear [31]	Matmul [14][a]	Our Simple Packing	Our Gen. Huang et al. Packing	Our Depthw. Packing
	2 Party LAN Setting				
7	48	5198	322	322	55
9	75	5198	321	323	74
11	108	5198	322	321	92
13	145	10397	321	322	92
15	191	10397	322	322	117
17	246	15595	322	322	117
19	305	15595	323	322	117
21	370	20794	323	322	166
23	444	25992	322	322	166
25	525	25992	323	323	166
50[b]	2175	103968	358	359	349
120[b]	12448	587419	820	825	1102
240[b]	50039	2339280	2709	2719	3757
	2 Party WAN Setting				
7	127	5494	638	636	122
9	195	5494	637	637	157
11	273	5494	636	636	189
13	363	10987	636	637	190
15	475	10987	637	640	235
17	611	16481	638	639	236
19	756	16481	638	637	236
21	913	21974	637	637	328
23	1093	27468	638	638	327
25	1295	27468	639	639	329
50[b]	5254	109872	709	706	682
120[b]	30075	620777	1475	1471	1939
240[b]	121021	2472120	4606	4603	6313

[a] column extrapolated from the runtime results in [14]
[b] row extrapolated from results with depth $d = 32$ (except for matmul runtime; see above)

overhead in computing the secure drowning (Appendix F.3.1) and in the size of expanded BGV ciphertexts that are sent in **MMultiply**. Even without the overhead of the secure drowning, one observes that this packing is slower than the other packing methods or even standard field multiplication with LowGear. The corresponding overall runtime (online and offline) can be found in Table 16. The communication cost can be found in Tables 19 and 20.

Online Phase. Tables 21 and 23 show our runtime results for the online phase. The online phase is benchmarked for $n = 2$ and $n = 4$, corresponding to the two settings (i.e., LowGear with two parties and HighGear with four parties) in the offline phase. The results for evaluating convolutions of ResNet50 show that convolution triples perform noticeably better than SPDZ for any type of convolution. However, a similar speed-up can be seen for matrix triples (however slightly less than for convolution triples). As discussed above, the main difference will be the total runtime (or the offline phase). There, our protocols have a clear advantage. The communication cost can be seen in Tables 22 and 24. There, we also see a clear advantage when using convolution triples compared to matrix triples.

Table 15: Runtime Results for dconv2d **Operations in the Offline Phase (in Seconds).** Our protocols here are LowGear-based (cf. Section 6.2 and Appendix F.3.2). Runtime is given for convolutions of $1 \times h \times h \times 512$ images with $512 \times 3 \times 3$ filters. This complements Table 3 by adding results for Bian et al.'s packing. The secure variant of the latter corresponds to the scheme described in Appendix F.3.1 and the insecure variant is the naive approach sketched at that very place.

h	LowGear [31]	Our Depthwise Packing	Ours With Insecure Bian et al. Packing	Ours With Secure Bian et al. Packing
	2 Party LAN Setting			
7	37	55	1084	5305
9	64	74	1625	7757
11	96	91	2288	10712
13	132	92	3064	14199
15	177	116	3960	18194
17	231	117	5042	23100
19	288	117	6185	28108
21	352	165	7851	36242
23	424	165	9220	42264
25	504	165	10957	50590

Table 16: Overall (Online and Offline) Runtime Results for dconv2d **Operations (in Seconds).** Our protocols here are LowGear-based (cf. Section 6.2 and Appendix F.3.2). Runtime is given for convolutions of $1 \times h \times h \times 512$ images with $512 \times 3 \times 3$ filters. Runtime is given for the experiments with Bian et al.'s packing as in Table 15.

h	LowGear [31]	Our Depthwise Packing	Ours With Insecure Bian et al. Packing	Ours With Secure Bian et al. Packing
	2 Party LAN Setting			
7	48	55	1084	5305
9	75	74	1625	7757
11	108	92	2288	10712
13	145	92	3065	14199
15	191	117	3961	18194
17	246	117	5043	23101
19	305	117	6186	28108
21	370	166	7851	36243
23	444	166	9220	42265
25	525	166	10958	50591

Table 17: Communication Costs for dconv2d **Operations in the Offline Phase (in MB).** Our protocols here are LowGear-based (cf. Section 6.2). Costs are given for convolutions of $1 \times h \times h \times 512$ images with $512 \times 3 \times 3$ filters.

h	LowGear [31]	Matmul [14]	Our Simple Packing	Our Gen. Huang et al. Packing	Our Depthw. Packing
	2 Party Setting				
7	233	1794	3502	3502	525
9	407	1794	3502	3502	762
11	611	1794	3502	3502	987
13	844	3588	3503	3503	988
15	1135	3588	3503	3503	1269
17	1484	5383	3504	3504	1270
19	1863	5383	3505	3505	1272
21	2270	7177	3505	3505	1831
23	2736	8971	3506	3506	1833
25	3260	8971	3507	3507	1834
50^a	13504	35885	3959	3959	3980
120^a	77764	202749	9729	9729	14782
240^a	312918	807408	34173	34173	50940

[a] row extrapolated from results with depth $d = 32$ (except for matmul costs)

Table 18: Overall (Online and Offline) Communication Costs for dconv2d **Operations (in MB).** Our protocols here are LowGear-based (cf. Section 6.2). Costs are given for convolutions of $1 \times h \times h \times 512$ images with $512 \times 3 \times 3$ filters.

h	LowGear [31]	Matmul [14]	Our Simple Packing	Our Gen. Huang et al. Packing	Our Depthw. Packing
	2 Party Setting				
7	239	1870	3502	3502	525
9	418	1870	3503	3503	763
11	627	1870	3503	3503	988
13	866	3739	3504	3504	989
15	1165	3739	3505	3505	1271
17	1524	5609	3506	3506	1273
19	1912	5609	3508	3508	1275
21	2331	7479	3509	3509	1835
23	2809	9349	3511	3511	1837
25	3347	9349	3512	3512	1839
50^a	13863	37395	3980	3980	4000
120^a	79864	211280	9847	9847	14900
240^a	321365	841382	34645	34645	51412

[a] row extrapolated from results with depth $d = 32$ (except for matmul costs)

Storage Cost. Finally, we give the storage cost for the different approaches in Tables 25 and 26. The behavior is similar to the communication cost in the offline phase but the advantage of convolution triples is not as pronounced. Note that for the conv1@7x7 convolution, our convolution triples are larger than the respective matrix triples. This is because (as mentioned in Section 6.5.4) we only compute the triple for a convolution with full padding and stride 1, whereas the convolution in question is with same padding an stride 2. One could simply discard also parts of the triple (as they are not used in the online phase) beforehand to avoid storing them in the first palace, which reduces the communication cost even more.

Table 19: Communication Costs for dconv2d **Operations in the Offline Phase (in MB).** Our protocols here are LowGear-based (cf. Section 6.2 and Appendix F.3.2). Costs are given for the experiments with Bian et al.'s packing as in Table 15.

h	LowGear [31]	Our Depthwise Packing	Ours With Insecure Bian et al. Packing	Ours With Secure Bian et al. Packing
	2 Party Setting			
7	233	525	35483	35483
9	407	762	51613	51613
11	611	987	70966	70966
13	844	988	93544	93544
15	1135	1269	119355	119355
17	1484	1270	151605	151605
19	1863	1272	183864	183864
21	2270	1831	238700	238700
23	2736	1833	277408	277408
25	3260	1834	332244	332244

Table 20: Overall (Online and Offline) Communication Costs for dconv2d **Operations (in MB).** Our protocols here are LowGear-based (cf. Section 6.2 and Appendix F.3.2). Costs are given for the experiments with Bian et al.'s packing as in Table 15.

h	LowGear [31]	Our Depthwise Packing	Ours With Insecure Bian et al. Packing	Ours With Secure Bian et al. Packing
	2 Party Setting			
7	239	525	35484	35484
9	418	763	51613	51613
11	627	988	70967	70967
13	866	989	93545	93545
15	1165	1271	119357	119357
17	1524	1273	151607	151607
19	1912	1275	183867	183867
21	2331	1835	238704	238704
23	2809	1837	277412	277412
25	3347	1839	332249	332249

Table 21: Runtime Results for conv2d **Operations in the Online Phase (in Seconds).** Runtime is given for ResNet50 convolution layers as in Table 2.

c Layer	SPDZ	Mat. Trip.	Conv. Trip.	SPDZ	Mat. Trip.	Conv. Trip.
	2 Party LAN Setting			4 Party LAN Setting		
1 conv1[a]	265.66	18.69	17.43	439.49	22.39	18.37
3 conv2	774.78	50.20	48.08	1270.92	54.93	49.62
4 conv3	1022.51	65.02	62.38	1662.57	69.03	64.48
6 conv4	1505.93	97.45	90.44	2317.69	103.34	94.51
3 conv5[a]	688.66	53.55	45.48	999.41	62.22	53.66
Total	**4257.55**	**284.91**	**263.81**	**6690.09**	**311.92**	**280.64**
	2 Party WAN Setting			4 Party WAN Setting		
1 conv1[a]	469.09	22.96	19.12	1020.07	34.39	21.31
3 conv2	1363.91	55.55	49.80	2966.02	70.33	52.78
4 conv3	1821.19	69.66	64.49	3876.37	80.71	67.40
6 conv4	2791.43	105.78	94.86	5393.12	119.70	104.35
3 conv5[a]	1266.39	64.05	55.19	2435.73	91.63	80.05
Total	**7712.00**	**318.00**	**283.47**	**15691.31**	**396.76**	**325.89**

[a] row extrapolated from results with halved output depth

Table 22: Communication Costs for conv2d **Operations in the Online Phase (in MB).** Costs are given for ResNet50 convolution layers as in Table 2.

c Layer	SPDZ	Matmul [14]	Matrix Triples	Conv. Triples
	2 Party Setting			
1 conv1@7x7[a]	3718.88	102.76	59.16	4.97
3 conv2@3x3	10835.50	196.61	88.47	11.40
4 conv3@3x3	14101.30	132.12	67.24	15.86
6 conv4@3x3	20132.70	226.49	99.98	61.44
3 conv5@3x3[a]	9084.90	226.49	134.92	115.66
Total	**57873.28**	**884.47**	**449.77**	**209.33**
	4 Party Setting			
1 conv1@7x7[a]	5578.32	154.14	88.74	7.45
3 conv2@3x3	16253.23	294.91	132.71	17.11
4 conv3@3x3	21152.02	198.18	100.86	23.79
6 conv4@3x3	30199.22	339.74	149.96	92.16
3 conv5@3x3[a]	13627.46	339.74	202.39	173.48
Total	**86810.24**	**1326.71**	**674.66**	**313.99**

[a] row extrapolated from results with halved output depth (expect for matmul costs)

Table 23: Runtime Results for dconv2d **Operations in the Online Phase (in Seconds).** Runtime is given for convolutions of $1 \times h \times h \times 512$ images with $512 \times 3 \times 3$ filters.

h	SPDZ	Matrix Triples	Conv. Triples	SPDZ	Matrix Triples	Conv. Triples
	2 Party LAN Setting			4 Party LAN Setting		
7	10.97	10.53	0.26	11.04	10.57	0.40
9	11.47	10.66	0.33	11.66	10.74	0.47
11	12.11	10.85	0.36	12.54	11.03	0.55
13	12.36	10.88	0.38	13.66	11.20	0.60
15	13.39	11.11	0.44	13.95	11.52	0.70
17	14.77	11.33	0.49	15.44	11.87	0.79
19	16.33	11.57	0.54	16.99	12.06	1.02
21	17.80	11.94	0.60	17.53	12.57	1.15
23	19.93	12.06	0.70	18.71	13.10	1.29
25	20.68	12.58	0.78	20.75	13.90	1.37
50[a]	37.27	24.55	6.20	47.76	28.98	9.39
120[a]	169.76	56.40	16.06	235.46	81.89	27.39
240[a]	640.97	202.27	52.31	1026.19	378.46	80.13
	2 Party WAN Setting			4 Party WAN Setting		
7	37.11	36.47	0.87	37.34	36.66	1.32
9	37.63	36.64	1.04	38.24	36.99	1.59
11	38.68	36.94	1.06	39.38	37.41	1.70
13	39.59	37.27	1.10	40.81	37.94	1.88
15	40.86	37.78	1.26	42.57	38.49	2.09
17	42.71	38.22	1.34	44.76	39.37	2.33
19	44.45	38.67	1.41	47.31	40.14	3.04
21	46.02	39.08	1.49	50.00	41.10	3.31
23	48.57	40.13	1.66	51.95	42.24	3.60
25	51.30	40.69	1.75	56.00	43.33	3.74
50[a]	99.12	65.25	17.45	120.98	81.43	29.24
120[a]	310.46	142.29	32.56	534.70	276.65	70.28
240[a]	1178.75	461.75	86.59	2428.32	1090.76	175.67

[a] row extrapolated from results with depth $d = 32$

Table 24: Communication Costs for dconv2d **Operations in the Online Phase (in MB).** Costs are given for convolutions of $1 \times h \times h \times 512$ images with $512 \times 3 \times 3$ filters.

h	SPDZ	Matmul [14]	Matrix Triples	Conv. Triples
	2 Party Setting			
7	5.91	75.50	3.69	0.48
9	10.24	75.50	6.05	0.74
11	15.75	75.50	8.99	1.07
13	22.43	150.99	12.53	1.46
15	30.29	150.99	16.66	1.92
17	39.34	226.49	21.38	2.44
19	49.56	226.49	26.69	3.03
21	60.97	301.99	32.59	3.69
23	73.55	377.49	39.08	4.41
25	87.31	377.49	46.15	5.19
50[a]	358.88	1509.95	184.40	20.56
120[a]	2099.86	8531.21	1061.76	118.04
240[a]	8446.40	33973.86	4246.83	471.93
	4 Party Setting			
7	8.87	113.25	5.53	0.71
9	15.36	113.25	9.07	1.11
11	23.62	113.25	13.49	1.60
13	33.65	226.49	18.80	2.19
15	45.44	226.49	24.99	2.88
17	59.01	339.74	32.07	3.66
19	74.34	339.74	40.04	4.55
21	91.45	452.98	48.88	5.53
23	110.32	566.23	58.61	6.61
25	130.97	566.23	69.23	7.79
50[a]	538.32	2264.92	276.60	30.84
120[a]	3149.80	12796.82	1592.66	177.06
240[a]	12669.68	50960.79	6370.28	707.91

[a] row extrapolated from results with depth $d = 32$ (except for matmul costs)

Table 25: Storage Costs for conv2d **Operations (in MB).** Costs are given for ResNet50 convolution layers as in Table 2.

c Layer	SPDZ	Matmul [14]	Matrix Triples	Conv. Triples
1 conv1@7x7	11329.34	308.28	85.00	113.46
3 conv2@3x3	33294.39	589.82	196.21	43.47
4 conv3@3x3	44392.51	396.36	147.32	46.47
6 conv4@3x3	66588.77	679.48	209.58	135.46
3 conv5@3x3	33294.39	679.48	250.58	232.88
Total	**188899.39**	**2653.42**	**888.70**	**571.74**

Table 26: Storage Costs for conv2d **Operations (in MB).** Costs are given for convolutions of $1 \times h \times h \times 512$ images with $512 \times 3 \times 3$ filters.

h	SPDZ	Matmul [14]	Matrix Triples	Conv. Triples
7	21.68	226.49	8.18	2.28
9	35.83	226.49	13.42	3.46
11	53.53	226.49	19.97	4.90
13	74.76	452.98	27.84	6.60
15	99.53	452.98	37.01	8.57
17	127.84	679.48	47.50	10.80
19	159.69	679.48	59.29	13.29
21	195.08	905.97	72.40	16.04
23	234.01	1132.46	86.82	19.05
25	276.48	1132.46	102.55	22.33
50	1105.92	4529.85	409.75	85.41
120	6370.10	25593.64	2359.44	479.94
240	25480.40	101921.59	9437.33	1903.38