Actively Secure Polynomial Evaluation from Shared Polynomial Encodings

Pascal Reisert[1], Marc Rivinius[1], Toomas Krips[2], Sebastian Hasler[1], and Ralf Küsters[1]

[1] Institute of Information Security, University of Stuttgart, Germany
{pascal.reisert,marc.rivinius,sebastian.hasler,ralf.kuesters}@sec.uni-stuttgart.de
[2] University of Tartu, Estonia
toomas.krips@ut.ee

Abstract. Many of the currently best actively secure Multi-Party Computation (MPC) protocols like SPDZ (Damgård et al., CRYPTO 2012) and improvements thereof use correlated randomness to speed up the time-critical online phase. Although many of these protocols still rely on classical Beaver triples, recent results show that more complex correlations like matrix or convolution triples lead to more efficient evaluations of the corresponding operations, i.e. matrix multiplications or tensor convolutions. In this paper, we address the evaluation of multivariate polynomials with a new form of randomness: polytuples. We use the polytuples to construct a new family of randomized encodings which then allow us to evaluate the given multivariate polynomial. Our approach can be fine-tuned in various ways to the constraints of applications at hand, in terms of round complexity, bandwidth, and tuple size. We show that for many real-world setups, a polytuples-based online phase outperforms state-of-the-art protocols based on Beaver triples.

Keywords: Multi-party computation · randomized encodings · SPDZ

This report is a major extension of our previous eprint [Rei+22].

1 Introduction

Multi-Party Computation (MPC) enables multiple parties to perform computations on private inputs without revealing any information about the inputs apart from what can be deduced from the result. State-of-the-art actively secure MPC protocols, like SPDZ [DKL+13, DPSZ12] and related protocols [BCS20, KOS16, KPR18], follow a two-phase approach, where correlated randomness is precomputed in an offline phase, and later consumed in an online phase to efficiently evaluate a function on private inputs. In this setup, a less efficient offline phase is normally considered acceptable since the offline phase can start well before the input data becomes available. Efficiency in two-phase protocols (and generally in MPC protocols) depends on the number of communication rounds needed and the bandwidth, i.e. the amount of data that has to be transmitted between the parties. Local computations, which can be performed without interaction, are usually considered less problematic as long as hardware requirements, e.g. memory requirements, remain manageable.

In MPC protocols based on additive secret sharing like SPDZ, addition and multiplication with public values are local operations and therefore fast, while the multiplication of secret values requires interaction and correlated randomness. The most common and widely used form of correlated randomness is classical Beaver triples [Bea92]. The standard approach is to represent a function, e.g. a matrix multiplication, as a series of additions and multiplications and then to use a Beaver triple for each multiplication and to add locally. However, this approach is often not the most efficient choice and for several common operations like matrix multiplication [MZ17,Rei+23]

or tensor convolutions [CKR+20, RRHK23] there are by now more efficient actively secure MPC solutions that rely on different forms of correlated randomness like matrix or convolution triples.

Many of these operations like simple field multiplication (Beaver triples), matrix multiplication (matrix triples) and tensor convolution (convolution triples) have in common that they are at most quadratic in the secret inputs. Using this property the protocols achieve a low online communication complexity. Additionally, the quadratic nature can be used in the offline phase, e.g. by using the linear homomorphic structure of lattice-based encryption schemes like BGV [BGV12] to generate the correlated randomness efficiently.

For higher-order operations, like the evaluation of (high-degree) multivariate polynomials, the situation is more difficult, and comparable constructions do not exist. We want to address this problem and present a new actively secure MPC protocol and a suitable new form of correlated randomness called *polytuples*, which speeds up the online evaluation of multivariate polynomials compared to the Beaver triples based approach and still has a reasonably fast offline phase.

We want to briefly describe the high-level idea of our approach. A SPDZ-like online phase has the following characteristics: at the beginning n parties $P_1, \ldots, P_n$ possess (among others) additive shares of the input variables $x_0, \ldots, x_{m-1}$, they perform local computations and communicate until each party P_i has a share $[y]_i$ of the result $y = f(x_0, \ldots, x_{m-1})$ (cf. Section 3.2 for the definition of additive shares $[\,\cdot\,]$). To open the result, the parties exchange the $[y]_i$ and locally reconstruct the result $\mathsf{Rec}([y]_1, \ldots, [y]_n) := \sum_{i=1}^{n} [y]_i = y$. [3]

This scheme is, however, by no means the only possible construction. In fact, it is enough for the parties to construct any *randomized encoding* [IK00, AIK06] of f. A randomized encoding is a set of terms $y_0, \ldots, y_{k-1}$ that depend on the inputs (and some randomness) and a reconstruction algorithm Rec such that $\mathsf{Rec}(y_0, \ldots, y_{k-1}) = f(x_0, \ldots, x_{m-1})$. Additionally, $y_0, \ldots, y_{k-1}$ and Rec are chosen in a way to not leak more information than the actual output $f(x_0, \ldots, x_{m-1})$ (cf. the formal Definition 1). Note that randomized encodings contain the classical SPDZ-setup as the special case $y_i = [y]_i$ for $0 \leq i < k = n$ where the parties do almost all of the computation in the interactive phase and only a simple sum in the final reconstruction phase. In particular, the evaluation of a degree d multivariate polynomial then requires around $\log_2(d)$ rounds of communication, which might be too much, especially in networks with high latency. It is then advantageous to reduce the round complexity by shifting more of the overall computation into a then more complex reconstruction Rec, since this reconstruction is done locally by each party and therefore nevertheless cheap. Naturally, certain limitations apply to this shift of computation. For example, the size k of the encoding should still remain within practical range for two reasons: (i) For a very large k (e.g. exponential) the local evaluation of Rec might still slow down the overall multi-party computation and (ii) all the encodings $y_0, \ldots, y_{k-1}$ have to be created either by the offline phase or through communication with the other parties and hence a large k increases the bandwidth or the offline runtime.

One of the main contributions of this paper is the construction of a new family of efficient randomized encodings of arbitrary multivariate polynomials f which satisfies these constraints and allows an efficient MPC protocol with only one round of communication. To this end, we follow an iterative approach, where we first construct an encoding $y_0, \ldots, y_{k-1}$ such that each y_l is of degree at most $d_1 < \deg(f)$ in the inputs. We next construct a randomized encoding $(y_{ll'})$ for each y_l of even smaller degree $d_2 < d_1$. The idea is that if the parties have a degree d_2 randomized encoding of each y_l, then they can locally reconstruct all y_l and if they have all y_l

[3] In order to get actively secure protocols, the opening protocols additionally include a MAC check (see our full version Protocol 5 or [DPSZ12]).

Table 1: Comparison for the computation of $[x_1^{d/m} \cdots x_{m-1}^{d/m}]$ of degree d with $d/m \in \mathbb{N}$, for Beaver triples, binomial tuples, and polytuples.

Approach	Rounds	Bandwidth	Tuple Size
Beaver Triples	$\lceil \log d \rceil$	$2(m-1)\lceil \log \frac{d}{m} \rceil$	$3(m-1)\lceil \log d/m \rceil$
e.g. for $d = m = 16$	4	30	45
Binomial Tuples[4] [CWB18]	1	m	$(\frac{d}{m}+1)^m - 1$
e.g. for $d = m = 16$	1	16	65535
Example Intermediate Polytuple	1	$\mathcal{O}(m \log(m))$	$\mathcal{O}(d(\log m)^2)$
e.g. for $d = m = 16$	1	41	149

then they can reconstruct the result $f(x_0, \ldots, x_{m-1})$. Hence the collection of all $y_{ll'}$ is a degree d_2 randomized encoding of f itself.

While the composition (cf. Lemma 2) of randomized encodings is a well-known result [AIK06], we add a twist. Namely, we construct the encodings $y_{ll'}$ in a way that they can be used in the reconstruction of *multiple* y_l, e.g. $y_{ll'}$ occurs in the randomized encoding of y_l and y_ℓ for some l, ℓ. We prove that the multiple use of such an encoding does not affect the security of the resulting overall randomized encoding of f of degree d_2. Thus we need less encodings (of degree d_2) to construct all y_l and hence $f(x_0, \ldots, x_{m-1})$.

In the next iteration step, we replace the degree d_2 encoding y_l of f by an encoding $y_{ll'l''}$ of even smaller degree $d_3 < d_2$. Again we can find $y_{ll'l''}$ which can be used in the reconstruction of multiple $y_{ll'}$ and we only need to construct a small number of these $y_{ll'}$ by the previous step. Hence iterating further the advantage of our multipurpose encodings becomes more significant and allows us to e.g. construct a degree 3 encoding of $f(x_0, \ldots, x_{m-1}) = x_0 \cdots x_{m-1}$ with output size in $\mathcal{O}(m \log(m))$. Previous results like [CFIK03] reached $\mathcal{O}(m^2)$.

In order to use the new randomized encodings to locally reconstruct the results, the parties first need to construct the components y_l in an interactive protocol. We therefore build a new MPC online protocol based on a new form of correlated randomness, i.e. our *polytuples*. Polytuples are specially crafted to allow the computation of the shares $[y_l]$ in only one round of online communication. These shares are then (partly) opened and each party can locally reconstruct the output $f(x_0, \ldots, x_{m-1})$ (or a share thereof).

Our new family of randomized encodings contains a large number of randomized encodings for each single polynomial f. While all these randomized encodings use the aforementioned optimization with multipurpose encodings, they differ in the number of iteration steps and the degree of the final overall encoding. Moreover, we can use encodings of different degrees for different components, e.g. a degree 4 encoding for y_1 and a degree 3 encoding for y_2.

The choice of a randomized encoding for a given polynomial f and the resulting number and shape of the encodings y_l and of the polytuples, strongly influence various aspects of the online and offline phase for the parties. For example, a low number of iteration steps and/or overall encodings of high degree reduce the output size k. Since all encodings have to be opened this decreases the bandwidth. The tradeoff is a larger tuple size and hence a more complex offline phase (see Section 4 for the explicit formulas for tuple size and bandwidth).

Table 1 shows one specific kind of randomized encoding and polytuple. This tuple lies between the linear size for Beaver multiplication and the exponential size of the more straightforward one-

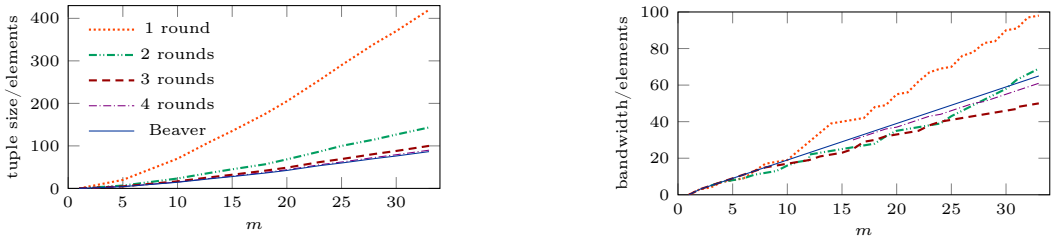


Fig. 1: Multi-round example to evaluate a product of m factors with polytuples with optimal tuple size.

round approach from [CWB18,Cou19].[4] It has minimal round complexity and a higher bandwidth cost than the other approaches. Almost all other trade-offs are however possible. The exact relation will be explained in Section 4.

Moreover, our protocol is also composable, i.e. we can write a multivariate polynomial $f(x_0, \ldots, x_{m-1}) = g(g_1(x_0, \ldots x_{m-1}), \ldots, g_j(x_0, \ldots x_{m-1}))$ for multivariate polynomials g, g_l $(1 \leq l \leq j)$ and then compute $[g_l(x_0, \ldots x_{m-1})]$ in the first round with our one-round protocol applied to all g_l and then compute $f(x_0, \ldots, x_{m-1})$ in the second round with the protocol applied to g for inputs $[g_l(x_0, \ldots x_{m-1})]$. This feature adds additional flexibility since it allows us to trade round complexity and bandwidth/tuple size; Figure 1 illustrates that adding just one round can already make a big difference.

Altogether, we can fine-tune our randomized encodings and polytuples for optimal performance in the concrete setting where the protocol is deployed w.r.t. bandwidth, tuple size, and/or round complexity. For example, if network latency is (moderately) high, we should try to minimize round complexity. Similarly, bandwidth/data rate restrictions imply that one should use polytuples with lower bandwidth. If the runtime of the offline phase, local memory or local computation time are important, striving for small tuple sizes is recommended. Our first experiments show that strategic deployment of polytuples can significantly speedup the performance of the online phase.

Our Contributions. In summary, our contributions are as follows:

- We introduce a new family of randomized encodings for the evaluation of multivariate polynomials as well as suitable correlated randomness, i.e. *polytuples*, to integrate the randomized encodings into a dishonest majority actively secure MPC protocol. Our randomized encodings have the smallest known output size for arbitrary monomials. Our approach evaluates a multivariate polynomial in just one round of online communication plus one opening round.
- We compute the tuple size and bandwidth needed in the online phase for all new randomized encodings and corresponding polytuples. Our tuple size is significantly lower than for existing single-round approaches and also multi-round computations yield improvements (e.g. lower bandwidth and round complexity than Beaver multiplication).
- We evaluate the performance of our approach for sample applications (evaluation of polynomials, comparisons of secret-shared values, simple machine learning algorithms) in Section 5 which shows that polytuples speed-up these computations compared to Beaver multiplication.

[4] To the best of our knowledge no name has been fixed for the [CWB18] underlying correlated randomness—we therefore chose *binomial tuples* to refer to this type of randomness within our paper (cf. also Section 3.4 for a definition).

2 Related Work

We see our work as an improvement over the common online phase of SPDZ [DPSZ12] and related protocols [KOS16, KPR18, BCS20]. We therefore concentrate our discussion on recent progress applicable to SPDZ-like papers, rather than classical theoretical results like e.g. [CD01], or other MPC approaches like garbled circuits.

A first small optimization of the Beaver triple-based online phase in SPDZ already appeared in [DKL+13] where square pairs are used to improve the squaring of secret shared values. This idea has been picked up by Morten Dahl who describes in [Dah17] power tuples for the computation of a monomial x^d for a secret-shared value x, which are binomial tuples (cf. Section 3.4) for a single variable. Dahl [Dah17] also presents matrix triples and convolution triples which have also been discussed in [MZ17] in the passively secure domain, too. Matrix (and convolution) triples have since then seen further attention and are by now available as part of an actively secure protocol [CKR+20, RRHK23, HRRK24]. The multivariate version of binomial tuples appears in the passively secure protocol of [CWB18] with additional trust assumptions on the dealer, whereas the authenticated binomial tuples in this paper provide active security. Ohata and Nuida [ON20] as well as Couteau [Cou19] use a slight variation of a binomial tuple in the passively secure setup.

Another classical approach to the secure evaluation of a polynomial is included in [BB89] and again in [DFK+06]. The more recent extension presented in [LYKM22] uses multiplicative masking. Their combined passively-secure protocols need $4 + 1 + 2$ rounds of (online) communication (cf. [CdH10]). The general idea of using a multiplicative structure in the underlying primitives, e.g. a multiplicative secret sharing as in [BD19, GPS12], is quite tempting. However, these multiplicative sharings generally cannot compute additions in a cheap way, and conversion techniques back to an additive sharing as it is used in SPDZ-like protocols are costly. While these protocols have a constant round complexity and small tuple size, making them actively secure (if possible) usually comes with considerable overhead. Furthermore, there are many papers optimizing the use of maskings/tuples. For example, Boura et al. [BCG+18] reuse their masks for certain input variables for different multiplication gates. Moreover, function-dependent preprocessing can be used to decrease the required tuple size and bandwidth in the online phase [BENO19, PSSY21]. Also note that with a pseudo-random generator, as in [BCG+19], structured randomness can be produced without further communication. Special solutions also exist for more complex structured random data like the matrix triples mentioned before.

Randomized Encodings. Results on randomized encodings reach far back to the works of Ishai and Kushilevitz [IK00] who proved that every polynomial has a degree-3 randomized encoding. The complexity results of [IK00] have since been improved by [CFIK03] for general branching programs, e.g. for products of m variables they achieve randomized encodings of output size $\mathcal{O}(m^2)$. In comparison, our randomized encoding reduces the output complexity to $\mathcal{O}(m \log(m))$. Other papers like [Kol05] focus on the binary case (which is less related to our arithmetic setup) or relax the correctness or privacy requirements like [AIK06] to achieve better efficiency. We refer to [Ish13] for further classical references on randomized encodings. At the same time [IKM+13] presents new actively secure protocols with linear bandwidth and constant round complexity, but with exponential tuple size. Moreover, [Cou19] considers a multi-round approach which improves the bandwidth from linear in the classical Beaver triple-based approach to $\mathcal{O}(m/\sqrt{\log(m)})$. More recently, a new multi-party adapted version of randomized encodings (MPRE) evolved in [ABT18], where preprocessing and the first communication round are more flexible than in

our SPDZ-like setup—the latter is (almost completely) restricted to exchange masked inputs $x_j - a_j$ in the first communication round. The MPRE approach has led to new passively secure and actively secure MPC protocols [ABT18, ABT19, LLW20, LL22]. The currently best actively secure protocol [LL22] uses Oblivious Linear Evaluation (OLE) correlated randomness, needs two rounds of communication but bandwidth at least cubic in the number of parties n and in $\mathcal{O}(m^{1.5})$ in the online phase. In comparison, our protocols are linear in n and require only $\mathcal{O}(m \log(m))$ communication in the same number of rounds in the online phase.

3 Preliminaries

For our theoretical considerations in Sections 4.2 to 4.5 we are working on a commutative base ring R. For all other parts, we choose R a finite field as in [DPSZ12]. We call a computation local if the parties can perform it without interaction.

3.1 Performance Measures

When we analyze the theoretical performance of our protocols, bandwidth is measured in the number of ring elements sent. Analogously, the size of the structured randomness needed for one polynomial evaluation in the online phase, i.e. the tuple size, is the number of ring elements contained in the tuple. The round complexity of a protocol is the number of communication rounds. One communication round consists of all information that can be sent in parallel. In particular, if in a protocol party P_1 has to wait for a message from P_2 before P_1 can send her message, the protocol has round complexity 2. The opening phase in actively secure SPDZ-like protocols comes with an additional invocation of a MAC check subroutine (cf. Section 3.2 and Protocol 7)—to account for the different structures of an opening round we will count opening rounds separately, usually indicated by a "+1" in the round/bandwidth count. It is quite common to ignore the opening round completely for composable protocols since to compute the composition of two or more functions the parties need only one global opening round. E.g. if parties can compute a function f in $k_f + 1$ rounds and function g in $k_g + 1$ rounds, they can compute $g \circ f$ in $k_f + k_g + 1$ rounds. To simplify notation, we sometimes drop the "+1".

3.2 Secret-Sharing and SPDZ-MACs

As we focus on MPC in the dishonest majority setting, we use classical additive secret-sharing, denoted by $[\,\cdot\,]$. A secret x is shared among n parties such that $x = \sum_{i=1}^{n} [x]_i$ where $[x]_i$ is the share of party P_i. All shares are needed to reconstruct a secret and $n-1$ or less shares do not reveal any information. This secret sharing scheme is linear, i.e., we can set $[x+y]_i \coloneqq [x]_i + [y]_i$, $[cx]_i \coloneqq c \cdot [x]_i$, $[x+c]_i \coloneqq [x]_i + c \cdot \delta_{i1}$ for shared values x, y and a publicly known constant c, where δ_{ij} is the Kronecker delta. To open (or reconstruct) a secret-shared value, parties simply broadcast their shares and compute the sum of all shares. Our techniques are independent of the secret-sharing scheme.

In SPDZ and related protocols, shares are additionally authenticated to verify the outputs of the protocol using a MAC key [DKL+13, DPSZ12]. The MAC key $\alpha \in R$ is shared in the pre-processing phase. Secret shared values (including inputs and structured randomness like Beaver triples or polytuples) are authenticated in the offline phase—we use $[\![x]\!] \coloneqq ([x], [\alpha x])$ to denote authenticated shares of x and $[\![X]\!] = ([\![x_1]\!], \ldots, [\![x_k]\!])$ for a tuple $X = (x_1, \ldots, x_k)$. Linear operations on authenticated shares are a trivial extension of linear operations on shares with the

exception of $[\![x+c]\!]_i := ([x+c]_i, [\alpha x]_i + c \cdot [\alpha]_i)$. A MAC check enables parties to verify the integrity of previously opened shares (cf. Protocol 7 or [DKL$^+$13, DPSZ12]). The soundness of the MAC check is proportional to $\frac{1}{|R|}$ if R is a field, can be aggregated over many opened values, and does not reveal the MAC key [DKL$^+$13].

3.3 Randomized Encodings and Randomizing Polynomials

In our protocols we use randomized encodings [IK00] to reduce the communication rounds, bandwidth, and tuple size.

Definition 1. *Let $X, Y, \hat{Y}, A$ be finite sets and let $f : X \to Y$. A function $\hat{f} : X \times A \to \hat{Y}$ is called* randomized encoding *of f if the following holds:*

- **Correctness.** *There exists a reconstruction algorithm $\mathsf{Rec} : \hat{Y} \to Y$ such that $\mathsf{Rec} \circ \hat{f} = f \circ \mathrm{pr}_1$ where $\mathrm{pr}_1 : X \times A \to X, (x, a) \mapsto x$ is the projection.*
- **Privacy.** *There exists a simulator* Sim *such that $\mathrm{Sim}(f(x))$ and $\hat{f}(x, a)$ are identically distributed for all $x \in X$ if a is sampled uniformly from A.*

If $\hat{Y} = R^k$, we call the component functions of a randomized encoding, simply encodings *or* randomizing polynomials. *An encoding y_0 of $\hat{f} = (y_l)_{0 \le l < k}$ which is only added by the reconstruction algorithm, i.e. $\mathsf{Rec}(0, y_1, \ldots, y_{k-1}) + y_0 = \mathsf{Rec}(y_0, y_1, \ldots, y_{k-1})$, is called* additive.

In this paper, we usually have $X = R^m, Y = R, \hat{Y} = R^k$. The randomness space A is generally more complicated since it is a subvariety of some R^t defined by the structure of our randomness, e.g. for Beaver triples we would choose $A = \{(a, b, c) \in R^3 : ab = c\} \subset R^3$. We remark that for our MPC application, we also include components that are completely deterministic in the other components, e.g. $c = ab$ in the Beaver triple case, since we have to construct this randomness in the offline phase. For possible other applications of our randomized encodings, these deterministic components of A can be omitted.

Moreover, in our arithmetic setup we only need to consider randomized encodings where the entries y_l of $\hat{f}$ are *randomizing polynomials* in $m + t$ variables, i.e. $y_l : X \times A \to R, ((x_j)_{0 \le j < m}, (a_j)_{0 \le j < t}) \mapsto y_l(x_0, \ldots, x_{m-1}, a_0, \ldots, a_{t-1})$ is a polynomial in $x_0, \ldots, x_{m-1}, a_0, \ldots, a_{t-1}$. To simplify the notation we usually drop the explicit dependency of the y_l on x_j and a_j.

A randomized encoding $\hat{f}$ is said to be of *(total) degree-d*, if the entries y_l of $\hat{f}$ are of total degree at most d—both the x_j and the a_j count towards the total degree, e.g. $2x_0a_0^2$ has total degree 3. We write $\hat{f}$ is of x-degree d if it is of degree d in the variables x_j and of a-degree d if it is of degree d in the randomness a_j, i.e. $2x_0a_0^2$ is of x-degree 1 and a-degree 2.

The *output size* of a randomized encoding is the R-rank of $\hat{Y}$, i.e. in this paper the size k. In our protocols, the output size usually coincides (up to an addition by m) with the bandwidth of the corresponding MPC protocol. The *randomness size* t on the other hand corresponds to the tuple size of the employed polytuple.

For later use we recall some fundamental properties for the concatenation and composition of randomized encodings [AIK06]:

Lemma 1. *Let $\hat{f}_i(x, a_i)$ be randomized encodings for $f_i(x)$ with reconstruction algorithm Rec_i and $0 \le i < k$, then $\hat{f}(x, (a_i)_{0 \le i < k}) = (\hat{f}_i(x, a_i))_{0 \le i < k}$ is a randomized encoding of $f(x) = (f_i(x))_{0 \le i < k}$ with reconstruction $\mathsf{Rec} = (\mathsf{Rec}_i)_{0 \le i < k}$.*

Lemma 2. *Let* $(\hat{f}(x,a),\mathsf{Rec})$ *be a randomized encoding of* $f(x)$ *and* $(\hat{f}'((x,a),a'),\mathsf{Rec}')$ *a randomized encoding of* $\hat{f}(x,a)$ *(as a deterministic function of* (x,a)*). Then* $\tilde{f}(x,(a,a')) = \hat{f}'((x,a),a')$ *is a randomized encoding of* $f(x)$ *with reconstruction* $\mathsf{Rec}\circ\mathsf{Rec}'$*.*

3.4 Binomial Tuples

As mentioned before, our new MPC protocols[5] for the evaluation of a multivariate polynomial f rely on suitable randomized encodings $(y_0,\dots,y_{k-1})$ of f. Here, the single encodings y_l are built by an interactive one-round protocol that uses structured randomness. Since the y_l might have a degree larger than 2, Beaver triples are not enough and we need a type of structured randomness that allows us to build higher degree terms y_l in one round. The solution is what we call *binomial tuples* (*for* y_l) since their construction is (just like Beaver triples) based on binomial expansion. We want to briefly present binomial tuples and the corresponding MPC online protocol. A passively secure version of this protocol was used in [CWB18, Cou19].

The goal of the binomial tuple approach is to compute a polynomial f in m variables $x_0,\dots,x_{m-1}\in R$ of total degree $d=\sum_{j=0}^{m-1} d_j$ with one round of communication plus one opening round.

Let $x=(x_0,\dots,x_{m-1})$ and denote by $f_a(x)=f_{a_0,\dots,a_{m-1}}(x)=f(x_0+a_0,\dots,x_{m-1}+a_{m-1})$ a randomization. As a multi-variate polynomial f_a has the general form $\sum_{e\in E} b_e x^e$ for $x^e := \prod_{j=0}^{m-1} x_j^{e_j}$ and multi-index $e=(e_0,\dots,e_{m-1})\in \bigtimes_{j=0}^{m-1}\{0,\dots,d_j\} =: E$ and some coefficients $b_e\in R$ (which depend on the a_j). Now each party P_i receives (from the offline phase) a share $[\![b_e]\!]_i$ for all $e\in E$. We call the $(b_e)_{e\in E}$ (or the sharing $[\![b_e]\!]_{e\in E}$) a *binomial tuple*.

Additionally, assume that the parties already hold shares $[\![x_j]\!],[\![a_j]\!]$ of the input variables x_j and masks a_j. In the first round of (online) communication the parties exchange $[x_j]-[a_j]=[x_j-a_j]$ for $0\le j<m$ and reconstruct $x-a=(x_0-a_0,\dots,x_{m-1}-a_{m-1})$. Subsequently, each party P_i can locally compute a share

$$[\![f(x)]\!]_i = [\![f_a(x-a)]\!]_i = \sum_{e\in E} [\![b_e]\!]_i (x-a)^e \tag{1}$$

i.e. the parties can reconstruct $f(x)$ in the opening round.

Remark 1. If $f(x)=x^{(d_0,\dots,d_{m-1})}$ is a monomial, then

$$f_a(x) = (x+a)^{(d_0,\dots,d_{m-1})} = \sum_{e\in E}\left(\prod_{j=0}^{m-1}\binom{d_j}{e_j}\right) a^{(d_0,\dots,d_{m-1})-e} x^e.$$

Hence, we have $b_e = \left(\prod_{j=0}^{m-1}\binom{d_j}{e_j}\right) a^{(d_0,\dots,d_{m-1})-e}$. Thus, each party needs to receive a share of b_e from the preprocessing, i.e. the tuple size is $\prod_{j=0}^{m-1}(d_j+1)-1$, where the d_j+1 comes from running through the powers 0 to d_j and the final -1 corresponds to the case $e=(d_0,\dots,d_{m-1})$ where $b_e=1$ is constant and does not have to be shared explicitly. We see that the structured randomness $(b_e)_{e\in E}$ has a small size if $m=1$, but becomes exponential for monomials of many different factors, e.g. for $d_j=1$ for all $0\le j<m$ one has size $2^m-1=2^d-1$.

Although binomial tuples come with a minimal round complexity of 1+1 rounds and small bandwidth, e.g. $m+1$ ring elements for the polynomial $\prod_{j=0}^{m-1} x_j$, the often large tuple size makes

[5] The protocol will be presented later in Protocol 5.

binomial tuples too inefficient for most higher degree multivariate polynomial evaluations. Our *polytuples* (cf. Definition 2) will therefore not contain binomial tuples for high-degree polynomials, but rather combine and correlate low-degree binomial tuples to retain a small tuple size and bandwidth while keeping the round complexity minimal.

4 Our MPC Protocols for the Evaluation of Multivariate Polynomials

We now present our main technical results on randomized encodings and polytuples. In Section 4.1 we explain first what kind of randomized encodings are compatible with our MPC protocol and how they can be used in an online phase. Sections 4.2 to 4.5 construct our new family of suitable randomized encodings. It also analyzes the complexity of the randomized encodings and connects it to the bandwidth and tuple size of our MPC protocols. Finally, Section 4.6 contains our MPC protocols and the main theorems. We refer to Section 4.7 and Appendix C for a discussion on the polytuple generation in the offline phase.

4.1 MPC With Randomized Encodings

Our MPC online protocols (just like SPDZ) consider n parties $P_1, \ldots, P_n$ that receive shares of the input variables $x = (x_0, \ldots, x_{m-1})$ as well as shares of (structured) random data in the form of a structured random tuple $\hat{a} = (a_0, \ldots, a_{t-1})$ with $t \geq m$ from an offline phase. The parties can locally add the shares, but they need to interact to compute the product of two secrets like x_0x_1 or x_0a_0. In order to compute these products the parties have to exchange their shares, obviously not in plain, but in some masked form. Therefore, as in SPDZ (and for the binomial tuples in Section 3.4) we assume that the parties open $x_j - a_j, 0 \leq j < m$, in an initial round of communication. Thus after the initial communication round, all parties know the public values $x_j - a_j, 0 \leq j < m$, in addition to the shares already provided by the offline and input phase.

The parties can use this information to construct new shares $[y_l]_i$ between the initial communication round and the final opening round. They can locally multiply and add the public values $x_j - a_j$, but they cannot locally multiply the shares (in a meaningful way). Hence the $[y_l]_i$ can be polynomials in the $x_j - a_j$ with coefficients that have at most total degree 1 in $[x_0]_i, \ldots, [x_{m-1}]_i, [a_0]_i, \ldots, [a_{t-1}]_i$. E.g. $[a_2]_i (x_1 - a_1)^2$ can be computed locally by party P_i after the initial round of communication. However, $[a_1]_i \cdot [a_2]_i (x_1 - a_1)^2 \neq \left[a_1a_2(x_1 - a_1)^2\right]_i$, i.e. the degree 2 coefficient $[a_1]_i \cdot [a_2]_i$ is not sufficient to compute a share of the product locally. Instead, we need to include structure randomness $a_3 = a_1a_2$ in the tuple. Then P_i easily computes $[a_3]_i (x_1 - a_1)^2 = [a_1a_2]_i (x_1 - a_1)^2 = \left[a_1a_2(x_1 - a_1)^2\right]_i$, which now has a coefficient $[a_3]_i$ of degree 1.

After the local computation, the parties open the $[y_l]_i$ and each party gets $y_l = \sum_{i=1}^{n} [y_l]_i$. Note that the degree condition ensures that y_l turns into a polynomial in the x_j and a_j since the shares dissolve, e.g. $\sum_{i=1}^{n} [a_2]_i (x_1 - a_1)^2 = a_2(x_1 - a_1)^2$. In order to compute $f(x_0, \ldots, x_{m-1})$ privately the y_l (together with the $x_j - a_j$) must be a randomized encoding for a suitable reconstruction algorithm Rec, i.e. $\hat{f}(x_0, \ldots, x_{m-1}, a_0, \ldots, a_{t-1}) = (x_0 - a_0, \ldots, x_{m-1} - a_{m-1}, y_0, \ldots, y_{k-1})$ in the notation of Definition 1.[6] In particular, the parties can then locally apply Rec to $x_j - a_j$ and the now public y_l, to compute $f(x_0, \ldots, x_{m-1})$. Hence for a randomized encoding $\hat{f} = (y_l)_{0 \leq l < k}$ of f with

[6] In our encodings $\hat{f}$ we usually do not include the $x_j - a_j$ explicitly, since we can directly include polynomials in the $x_j - a_j$ in the y_l.

(I) y_l is a polynomial $\sum_{e(l)\in E(l)} b_{e(l)}(x,\hat{a})\cdot(x-a)^{e(l)}$ where $E(l)\subset\mathbb{N}^m$ some finite set of multi-indices and $a=(a_0,\ldots,a_{m-1})$ the input masks, and
(II) all coefficients $b_{e(l)}(x,\hat{a})$ have total degree at most 1 in $R[x_0,\ldots,x_{m-1},a_0,\ldots,a_{t-1}]$,

the parties $P_1,\ldots,P_n$ can compute $f(x_0,\ldots,x_{m-1})$ with Protocol 1 and option `continuation` = `open`. To later use our randomized encodings in multi-round online protocols (cf. Protocol 1 and Protocol 5) we furthermore require that

(III) y_0 is an additive component in the sense of Definition 1.

This allows the options `continuation` = share in Protocol 1 below to output a share $[\![f(x_0,\ldots,x_{m-1})]\!]_i$ of the result to each party P_i or to output a masked result $f(x_0,\ldots,x_{m-1})-b$ if `continuation` is a shared random value $[\![b]\!]$. We discuss the multi-round use in more detail in Section 4.6. The protocol $\Pi_{\text{polynomial}}$ for polynomial evaluations is the core part of our online phase. All other parts, e.g. the input protocol, are identical to their counterparts in SPDZ. We have included the full online protocol Π_{online} in Protocol 5.

$\Pi_{\text{polynomial}}$

Let $\hat{f}=(y_l)_{0\le l<k}$ be a randomized encoding of f that satisfies (I), (II), (III) with randomness space A. Each party has a share of $x=(x_0,\ldots,x_{m-1})$ and of some $\hat{a}=(a_0,\ldots,a_{t-1})\in A$. On input $(\hat{f},[\![x]\!],[\![\hat{a}]\!],$ `continuation`$)$ each party P_i does:

1. P_i locally computes and then opens $[\![x_j]\!]_i-[\![a_j]\!]_i$ for all $0\le j<m$. After receiving all shares, P_i locally computes x_j-a_j.
2. P_i locally computes $[\![y_l]\!]_i=\sum_{e(l)\in E(l)} b_{e(l)}([\![x]\!]_i,[\![\hat{a}]\!]_i)(x-a)^{e(l)}$ for all $0\le l<k$, $a=(a_0,\ldots,a_{m-1})$. If `continuation` $=[\![b]\!]$ then set $[\![y_0]\!]_i\leftarrow[\![y_0]\!]_i-[\![b]\!]_i$.
3. P_i opens $[\![y_l]\!]_i$ for all $i>0$ and locally computes $y_l=\sum_{i=1}^n [y_l]_i$ by summing up the received shares.
   a. If `continuation` = `share`, P_i locally constructs $[\![f(x_0,\ldots,x_{m-1})]\!]_i=[\![y_0+\mathsf{Rec}(0,y_1,\ldots,y_{k-1})]\!]_i=[\![y_0]\!]_i+\mathsf{Rec}(0,y_1,\ldots,y_{k-1})\delta_{1i}$.
   b. If `continuation` $\neq$ `share`, P_i opens and computes $y_0=\sum_{i=1}^n [y_0]_i$ and locally reconstructs $f(x_0,\ldots,x_{m-1})=\mathsf{Rec}(y_0,\ldots,y_{k-1})$.

Protocol 1: 1(+1) round interactive evaluation of a polynomial f.

Remark 2. As usual for SPDZ-like protocols, we get a passively secure version if we replace $[\![\cdot]\!]$ with a simple $[\cdot]$. We note that all constructions in this paper still work in the passive setup with this modification.

4.2 Our Randomized Encodings

We now want to construct suitable randomized encodings of arbitrary multivariate polynomials compatible with our MPC online phase, i.e. randomized encodings that satisfy (I)–(III) above. We already know from Section 3.4 that every multivariate polynomial can be computed with binomial tuples and also that these binomial tuples become too large for high-degree polynomials. Hence we will first construct low-degree randomized encodings and then use the binomial tuples from Section 3.4 to construct these low-degree terms as in (1) and Protocol 1, respectively.

We will start with homogeneous monomials $x_{0,\ldots,m-1}\coloneqq x_0\cdots x_{m-1}$, then lift our construction to arbitrary monomials, i.e. $x_0^{d_0}\cdots x_{m-1}^{d_{m-1}}$, and finally to arbitrary polynomials.

Idea of Our Construction. To construct a randomized encoding for $f(x_0,\dots,x_{m-1}) = x_{0,\dots,m-1} = \prod_{j=0}^{m-1} x_j$, we follow an iterative approach, where we first construct a degree d_1 encoding $\hat{f}^{(1)}$ of f for some $d_1 < m$, i.e. the components $y_l^{(1)}$ of $\hat{f}^{(1)}$ are polynomials of degree $\leq d_1$. We next construct a lower degree encoding $\hat{f}^{(2)}$ of degree $d_2 < d_1$, of $\hat{f}^{(1)}$ and use the composition Lemma 2 to get a new degree-d_2 encoding of f. Iteratively, we can reduce the degree of the encoding to a target degree, e.g. a degree-3 encoding.

The straightforward approach to construct a randomized encoding of $\hat{f}^{(1)}$ is to construct encodings for each of the component functions $y_l^{(1)}$ of $\hat{f}^{(1)}$ and then to concatenate the encodings with Lemma 1 to a randomized encoding of the whole $\hat{f}^{(1)}$. As mentioned before, in this paper we follow a more efficient approach, where we construct encodings $y_l^{(2)}$ that can be used in the reconstruction of *multiple* $y_l^{(1)}$.

We want to illustrate our approach with the special case $m = 2^n$. We use 3 types of encodings each linear in some monic monomial $x_{u,\dots,r-1} \coloneqq x_u \cdots x_{r-1}$ with $u < r$:

(i) with *constant prefactor* 1, i.e. of the form $f_{a_{u,\dots,r-1}}(x_u,\dots,x_{r-1}) = x_{u,\dots,r-1} - a_{u,\dots,r-1}$ and an $a_{u,\dots,r-1} \in A$;
(ii) with *one randomized prefactor* $a \in A$, i.e. of the form $g^a_{b_{u,\dots,r-1}}(x_u,\dots,x_{r-1}) = a x_{u,\dots,r-1} - b_{u,\dots,r-1}$ and a $b_{u,\dots,r-1} \in A$;
(iii) with *two randomized prefactors* $a, b \in A$, i.e. of the form $h^{a,b}_{c_{u,\dots,r-1}}(x_u,\dots,x_{r-1}) = abx_{u,\dots,r-1} - c_{u,\dots,r-1}$ and a $c_{u,\dots,r-1} \in A$.

We now want to construct randomized encodings for each of these three types of encodings which again consist of terms of type (i), (ii), or (iii), but of lower degree, i.e. with smaller $r-u$. Since our monomial $f(x_0,\dots,x_{m-1})$ is of type (i) with $u = 0, r = m$, this will allow us to construct a degree d_1 encoding $\hat{f}^{(1)} = (y_l^{(1)})$ where all $y_l^{(1)}$ are of type (i), (ii) or (iii) with x-degree $< r$. Then we can iterate.

To simplify notation we use a helper function

$$\varphi(x,y,a,b,c) \coloneqq (x-a, y-b, bx+ay-ab-c)$$

on R^5. Moreover, we choose a reconstruction $\mathsf{Rec}(y_0,y_1,y_2) \coloneqq y_0y_1 + y_2$ for output size 3 randomized encodings. Please note that y_2 is then an additive component in the sense of Definition 1. We get

$$\mathsf{Rec} \circ \varphi(x,y,a,b,c) = \varphi_0\varphi_1 + \varphi_2 = (x-a)(y-b) + bx + ay - ab - c = xy - c. \qquad (*)$$

Hence, we find for $v = (u+r)/2$ randomized encodings of $f_*, g^a_*, h^{a,b}_*$:

(1) $\hat{f}_{a_{u,\dots,r-1}}(x_u,\dots,x_{r-1},a_0,a_1) = \varphi(x_{u,\dots,v-1}, x_{v,\dots,r-1}, a_0, a_1, a_{u,\dots,r-1})$[7],
(2-1) $\hat{g}^a_{b_{u,\dots,r-1}}(x_u,\dots,x_{r-1},b_0,b_1) = \varphi(a x_{u,\dots,v-1}, x_{v,\dots,r-1}, b_0, b_1, b_{u,\dots,r-1})$,
(2-2) $\hat{g}^b_{b_{u,\dots,r-1}}(x_u,\dots,x_{r-1},b_0,b_1) = \varphi(x_{u,\dots,v-1}, b x_{v,\dots,r-1}, b_0, b_1, b_{u,\dots,r-1})$,
(3) $\hat{h}^{a,b}_{c_{u,\dots,r-1}}(x_u,\dots,x_{r-1},c_0,c_1) = \varphi(a x_{u,\dots,v-1}, b x_{v,\dots,r-1}, c_0, c_1, c_{u,\dots,r-1})$,

where $a_0, a_1, b_0, b_1, c_0, c_1 \in A$ are random numbers (not necessarily different). Note that for g we have two different cases depending on whether a randomized prefactor comes from the first component or the second.

[7] To simplify the notation we often write $*$ for the additive constant index if the index is clear from context.

While correctness follows in all four cases directly from $(*)$, we omit the security proof for now and refer to the general cases discussed in Section 4.3.

Please note that in all of these randomized encodings the components (given by some $\varphi_0, \varphi_1, \varphi_2$) are in fact linear combinations of terms of types (i), (ii) or (iii) and of x-degree $(r-u)/2 = 2^{n-1}$. Hence, we can iteratively apply the four randomized encodings again to get to an even smaller x-degree.

The first two components of (1), (2-1), (2-2), (3) (which come from some φ_0, φ_1) are simple terms of types (i)–(iii). For these we can iterate immediately, i.e. apply (1), (2-1), (2-2), (3) with either $u \leftarrow u, r \leftarrow v, v \leftarrow (u+r)/2$ or $u \leftarrow v, r \leftarrow r, v \leftarrow (u+r)/2$ to get encodings of the components of x-degree $(r-u)/2 = 2^{n-1}$ and output size 3. In the third components (corresponding to φ_2) we have sums of type (i)–(iii) terms. Here, we construct a randomized encoding for each summand (using suitable instances of (1), (2-1), (2-2), (3)) and then combine them to a randomized encoding of the sum.[8] Overall, this leads to four randomized encodings of output size 3 (as above) and x-degree 2^{n-1}: two for the first two components and two for the two summands of the third component. If we follow this path and reduce the x-degree iteratively by a factor 2 in each round, then we quickly see that we get (using concatenation Lemma 1 and composition Lemma 2) an overall randomized encoding of x-degree 1 (and a-degree ≤ 2) of output size in $\mathcal{O}(4^n) = \mathcal{O}(m^2)$ similar to the results in [CFIK03].

However, if we investigate our randomized encodings above a bit closer, then we see that we produce a significant amount of identical encodings multiple times. For example, if we set $a_1 = b_1$ then the second component of both $\hat{f}_*(x_u, \ldots, x_{r-1}, a_0, a_1)$ and $\hat{g}^a_*(x_u, \ldots, x_{r-1}, b_0, a_1)$ is $x_{v,\ldots,r-1} - a_1$. Analogously, we get a joined component for (1) and (2-2) if $a_0 = b_0$. Similarly, we see that for $b_0 = c_0$ the first components of both (2-1) $\hat{g}^a_*(x_u, \ldots, x_{r-1}, b_0, b_1)$ and (3) $\hat{h}^{a,b}_*(x_u, \ldots, x_{r-1}, b_0, c_1)$ are identical: $ax_{u,\ldots,v-1} - b_0$. Analogously, for (2-2) and (3) for $b_1 = c_1$. See also Fig. 2. Thus, if we choose the randomness suitably, it is enough to produce some of the encodings in (1), (2-1), (2-2), (3) only once and then use them in *multiple* reconstructions. E.g. this allows us to save 4 components when constructing a randomized encoding of $(f_*(x_u, \ldots, x_{r-1}), g^a_*(x_u, \ldots, x_{r-1}), g^b_*(x_u, \ldots, x_{r-1}), h^{a,b}_*(x_u, \ldots, x_{r-1}))$. Please note that while in general one cannot use the same encoding in different reconstructions without losing privacy, our construction allows the multiple use of encodings—we refer to Corollary 1 for the formal result. We can now conclude that we need for a randomized encoding of

(a) $f_*(x_u, \ldots, x_{r-1})$: 2 type (i) terms ($1^{\text{st}}$, 2^{nd} component of $\hat{f}$) and 2 type (ii) terms (summands in the 3^{rd} component of $\hat{f}$),
(b) each $g^a_*(x_u, \ldots, x_{r-1})$, $g^b_*(x_u, \ldots, x_{r-1})$: 2 additional type (ii) terms (1^{st} (2-1) or 2^{nd} (2-2) component of $\hat{g}$ + one summand in the 3^{rd} component) plus 1 type (iii) term (summand in the 3^{rd} component).
(c) $h^{a,b}_*(x_u, \ldots, x_{r-1})$: 2 additional type (iii) terms (summands in the 3^{rd} component of $\hat{h}$).

Please note that (b) assumes that (a) has been already produced; (c) assumes that both (a) and (b) have been produced. Fortunately, this is the only case that occurs in our iterative construction, i.e. whenever we need to construct a term $h^{a,b}_*$ we also need to construct the corresponding g^a_*, g^b_*, f_* linear in the same monomial. Analogous, whenever we need to construct a g^a_* or g^b_* we also need to construct an f linear in the same monomial.

[8] We omit details for this combination, which is treated in general in Corollary 1.

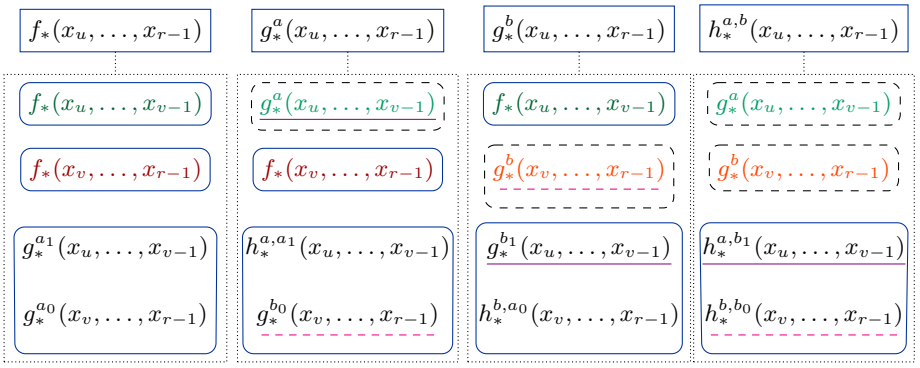


Fig. 2: Components in the encodings (1) [left], (2-a) [left-middle], (2-b) [right-middle], (3) [right]. Identical colors (apart from black) mark identical encodings, black/dashed boxed components are duplicates and therefore not produced again.

We want to briefly look at two successive iteration steps to explain why this is the case. We start with our monomial $f(x_0,\ldots,x_{m-1}) = x_{0,\ldots,m-1}$. Then the randomized encoding $\hat{f}_*(x_0,\ldots,x_{m-1},a_{0,\ldots,2v-1},a_{2v,\ldots,m-1})$ for $v=2^{n-2}, r=m=2^n, u=0$ from (1) leads to

- 2 terms $f_*(x_0,\ldots,x_{2v-1})$, $f_*(x_{2v},\ldots,x_{m-1})$ and 2 terms $g_*^{a_{2v,\ldots,m-1}}(x_0,\ldots,x_{2v-1})$, $g_*^{a_{0,\ldots,2v-1}}(x_{2v},\ldots,x_{m-1})$ in the 3rd component of $\hat{f}$ (see also left column in Fig. 2).

If we go one iteration further, i.e. apply (1), (2-1), (2-2), (3) to these four terms, $\hat{f}_*(x_0,\ldots,x_{2v-1},a_{0,\ldots,v-1},a_{v,\ldots,2v-1})$ leads again to 2 terms $f_*(x_0,\ldots,x_{v-1}), f_*(x_v,\ldots,x_{2v-1})$ and 2 terms $g_*^{a_{v,\ldots,2v-1}}(x_0,\ldots,x_{v-1}), g_*^{a_{0,\ldots,v-1}}(x_v,\ldots,x_{2v-1})$. But now we also get from $\hat{g}_*^{a_{2v,\ldots,m-1}}(x_0,\ldots,x_{2v-1},b_{0,\ldots,v-1},a_{v,\ldots,2v-1})$ in the case (2-1):[9]

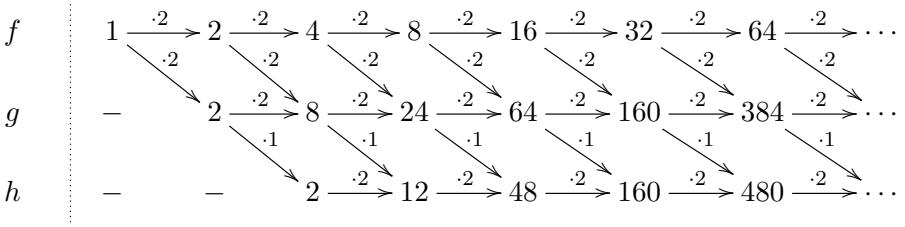
- 2 additional terms $g_*^{a_{2v,\ldots,m-1}}(x_0,\ldots,x_{v-1})$, $g_*^{b_{0,\ldots,v-1}}(x_v,\ldots,x_{2v-1})$, and one term $h_*^{a_{v,\ldots,2v-1},a_{2v,\ldots,m-1}}(x_0,\ldots,x_{v-1})$ (see Fig. 2, right-middle column).

We see that we get in fact the 4 terms $f_*, g_*^{a_{v,\ldots,2v-1}}, g_*^{a_{2v,\ldots,m-1}}, h_*^{a_{v,\ldots,2v-1},a_{2v,\ldots,m-1}}$ all linear in $x_{0,\ldots,v-1}$. Furthermore, observe that each type (ii) term only occurs with a corresponding type (i) term linear in the same monomial and that each $h_*^{a,b}$ only occurs with corresponding g_*^a, g_*^b and f_* terms all linear in the same monomial. Finally note that a type (iii) term $h_*^{a,b}(x_u,\ldots,x_{r-1})$ again leads to two type (iii) terms $h_*^{a,b_1}(x_u,\ldots,x_{v-1}), h_*^{b,b_0}(x_v,\ldots,x_{r-1})$ (Fig. 2, right). As we have seen, we also have $g_*^a(x_u,\ldots,x_{r-1}), g_*^b(x_u,\ldots,x_{r-1})$. Then we have to apply (2-1) to $g_*^a(x_u,\ldots,x_{r-1})$ and (2-2) to $g_*^b(x_u,\ldots,x_{r-1})$ (or vice versa) to ensure that $g_*^a(x_u,\ldots,x_{v-1})$ and $g_*^b(x_v,\ldots,x_{r-1})$ (or $g_*^b(x_u,\ldots,x_{v-1})$ and $g_*^a(x_v,\ldots,x_{r-1})$) are already available for the reconstruction of $h_*^{a,b}(x_u,\ldots,x_{r-1})$. The two different cases are (dashed) underlined in Fig. 2.

Overall we see that the number of needed encodings as described by (a), (b), (c) hold generally in our construction. Hence we can deduce the output complexity of our iterative approach,

[9] Analogously for (2-2).

namely:



We easily see that the number of type (i) terms f is in $\mathcal{O}(2^n) = \mathcal{O}(m)$, of type (ii) terms g is in $\mathcal{O}(2^n n) = \mathcal{O}(m \log(m))$ and type (iii) terms h is in $\mathcal{O}(2^n n^2) = \mathcal{O}(m(\log(m))^2)$. Hence we get overall complexity $\mathcal{O}(m(\log(m))^2)$ since we have to construct all of these terms. This is already a significant improvement over the currently best-known result $\mathcal{O}(m^2)$ [CFIK03].

However, the result is not ideal yet. We can further improve it by combining additive components: Consider $\mathsf{Rec}'(y_0, y_1, y_2, y_3, y_4) = y_0 y_1 + y_2 y_3 + y_4$ and

$$\begin{aligned}&\varphi'(x, y, a, b, x', y', a', b', c)\\ &\coloneqq (x - a, y - b, x' - a', y' - b', bx + ay + b'x' + a'y' - ab - a'b' - c)\end{aligned}$$

Then $\mathsf{Rec}' \circ \varphi' = xy + x'y' - c$. Thus for $v = r/4$:

$$\begin{aligned}&\hat{f}_{\mathsf{add}}(x_0, \dots, x_{r-1}, a_{0,\dots,v-1}, a_{v,\dots,2v-1}, a_{2v,\dots,3v-1}, a_{3v,\dots,r-1})\\ &= \varphi'(x_{0,\dots,v-1}, a_{2v,\dots,r-1} x_{v,\dots,2v-1}, a_{0,\dots,v-1}, b_{v,\dots,2v-1}, x_{2v,\dots,3v-1},\\ &\qquad a_{0,\dots,v-1} x_{3v,\dots,r-1}, a_{2v,\dots,3v-1}, b_{3v,\dots,r-1}, a_{0,\dots,2v-1} a_{2v,\dots,r-1})\end{aligned}$$

is a randomized encoding of the additive 3rd component of $\hat{f}_*$ (from (1)), i.e. of $a_{2v,\dots,r-1} x_{0,\dots,2v-1} + a_{0,\dots,2v-1} x_{2v,\dots,r-1} - a_{0,\dots,2v-1} a_{2v\dots,r-1}$. Note that $\hat{f}_{\mathsf{add}}$ only has 5 components compared to 6 that are needed if we construct each summand separately. Analogous results hold for the additive components of $\hat{g}^a_*, \hat{g}^b_*, \hat{h}^{a,b}_*$. Overall this reduces the output complexity down to $\mathcal{O}(m \log(m))$.

4.3 Technical Lemmas and Formal Results

We now want to present a generalization of the previous construction. The proofs to all statements in this section are available in Appendix A. We also refer to Appendix A for additional examples, e.g. Examples 1, 2, 4 and 5.

We start with the main technical Lemmas 3 to 5. The three lemmas discuss the three cases (i)–(iii) already presented above, i.e. no (Lemma 3), one (Lemma 4) or two randomized prefactors (Lemma 5). While in the previous special case the randomized encoding of $x_{0,\dots,m-1}$ consisted of terms either linear in $x_{0,\dots,t-1}$ or in $x_{t,\dots,m-1}$, the more general lemmas instead allow to construct a randomized encoding linear in any number $1 \le r_1 \le m$ of monomials $x_{S_{1,j}} \coloneqq \prod_{k \in S_{1,j}} x_k$ for $\{0, \dots, m-1\} = \dot{\bigcup}_{j \in \mathbb{Z}_{r_1}} S_{1,j}$ any disjoint union.[10] E.g. we can split $x_{0,\dots,8}$ into terms linear in the three monomials $x_{0,\dots,2}, x_{3,\dots,5}$ or $x_{6,\dots,8}$.

We will first state the lemmas and then explain how they can be combined into a low-degree encoding of any product $x_0 \cdots x_{m-1}$. Since we later apply the lemmas several times

[10] We use indices in $\mathbb{Z}_r$ because they wrap around nicely. To be more formal, we will sometimes use $\bar{i}$ for the unique representative of $i \in \mathbb{Z}_r$ in $\{0, \dots, r-1\}$.

in different degrees, they are stated in a generic degree r instead of m to avoid confusion. Furthermore, we use the following notation: For $\emptyset \neq J = \{j_0, \ldots, j_s\} \subset \mathbb{Z}_r$ with representatives $0 \leq j_0 < \cdots < j_s < r, j_{s+1} \coloneqq j_0$ and a set of functions $\{f_{ij}, (i,j) \in \mathbb{Z}_r^2\}$, define the product $f_J \coloneqq \prod_{v=0}^{s} f_{j_v, j_{v+1}-1}$. E.g. the set $J = \mathbb{Z}_5$ leads to $f_J = f_{0,0}f_{1,1}\cdots f_{4,4}$ and $J = \{2,4,5\} \subset \mathbb{Z}_6$ to $f_J = f_{2,3}f_{4,4}f_{5,1}$.

Lemma 3. *Let $f(x_0, \ldots, x_{r-1}) = x_{0,\ldots,r-1} - c$ for some consant $c \in R$. There is a randomized encoding $\hat{f}$ with randomness and output size both $r(r-1)+1$. The randomized components of $\hat{f}$ have the form $f_{ii} = x_i - a_i$, and $f_{ij} = x_i a_{i+1,\ldots,j} - a_{i,\ldots,j}$, and $f_{\mathsf{add}} = \sum_{i \in \mathbb{Z}_r} x_i a_{i+1,\ldots,i-1} - a_{\mathsf{add}}$ for randomness $a_i, a_{i+1,\ldots,j}$ for $i \neq j \neq i-1$ and $i,j \in \mathbb{Z}_r$, and $a_{\mathsf{add}} = c + \sum_{J \subset \mathbb{Z}_r, |J|>1} (-1)^{|J|} a_J$ where $a_J \coloneqq \prod_{v=0}^{s} a_{j_v,\ldots,j_{v+1}-1}$. The reconstruction function has the form $\mathsf{Rec}(f_{ij}, f_{\mathsf{add}}) \coloneqq f_{\mathsf{add}} + \sum_{J \subset \mathbb{Z}_r, |J|>1} f_J$.*

Proof. We first note that there are exactly $r^2 - r$ different factors in products f_J associated with sets J with $|J| > 1$, since a factor is defined by its start j_t and end index j_{t+1}, i.e. 2 ordered samples from $\mathbb{Z}_r$ drawn without replacement. We show first that $\prod_{j \in \mathbb{Z}_r} x_j - c = f_{\mathsf{add}} + \sum_{J \subset \mathbb{Z}_r, |J|>1} f_J \coloneqq \mathsf{Rec}(f_{ij}, f_{\mathsf{add}})$ for a suitable structured randomness a_{add} (constant in the x_j).[11] Note that apart from $\prod_{j \in \mathbb{Z}_r} x_j$ each non-constant summand in the expression on the right is of the form $x_j a_{j+1,\ldots,k-1} g$ for some specific term g and some j,k.[12] Each of these terms (for a fixed g, j and k) occurs exactly once with a positive sign for a J which contains $j_l = j, j_{l+1} = k \neq j+1$ for some l, i.e. as a summand in $f_{j_l,k-1} g = (x_{j_l} a_{j_l+1,\ldots,k-1} - a_{j_l,\ldots,k-1})g$ or f_{add} if $k = j$.[13] It occurs exactly once with a negative sign for a $J' = J \cup \{j+1\}$, i.e. as a summand in $f_{j_l,j_l} f_{j_l+1,k-1} g = (x_{j_l} - a_{j_l})(x_{j_l+1} a_{j_l+2,\ldots,k-1} - a_{j_l+1,\ldots,k-1})g$. Thus these terms cancel out. It remains only $\prod_{j \in \mathbb{Z}_r} x_j$ and constant random terms (in the x_j) which add up to $-c$ for a suitably chosen (structured) randomness a_{add}. Namely, $a_{\mathsf{add}} = c + \sum_{J \subset \mathbb{Z}_r, |J|>1} f_J(0, \ldots, 0)$ if we consider f_J as a function of the x_i. This shows the correctness of the randomized encoding.

For privacy, we first choose uniformly random (and in particular mutually independent) $a_{i,\ldots,j} \in R$ for $i \neq j+1$ and only a_{add} structured, i.e. a (deterministic) polynomial in the $a_{i,\ldots,j}$. Now the simulator samples its first $r(r-1)$ components $\tilde{f}_{ij}$ (corresponding to the f_{ij}) uniformly from R. Since each f_{ij} contains an additive random mask (and all masks are independent), the f_{ij} are also distributed uniformly if the $a_{i,\ldots,j}$ are sampled uniformly (cf. Definition 1). For the last component $\tilde{f}_{\mathsf{add}}$ (corresponding to f_{add}), the simulator computes $\tilde{f}_{\mathsf{add}} = -\mathsf{Rec}(\tilde{f}_{ij}, 0) + f(x_0, \ldots, x_{m-1})$. By construction $f_{\mathsf{add}} = -\mathsf{Rec}(f_{ij}, 0) + f(x_0, \ldots, x_{m-1})$ and $\tilde{f}_{\mathsf{add}}$ are equally distributed, which shows privacy. □

Remark 3. Observe that r of the encodings have constant leading coefficient 1 as a polynomial in $x_0, \ldots, x_{r-1}$, i.e. the f_{ii}. Moreover, there are $r(r-2)+1$ encodings where the leading coefficient is one random element, i.e. the f_{ij} for $i \neq j \neq i-1$ and f_{add}.

Lemma 4. *Let $g^a(x_0, \ldots, x_{r-1}) = a x_{0,\ldots,r-1} - c$ for some $a, c \in R$. Let $\mu \in \mathbb{Z}_r$ be a fixed index and define $T_\mu \coloneqq \{(i,j) \in \mathbb{Z}_r^2 : \overline{j-\mu} \leq \overline{i-\mu-1}\}$ and $S_\mu = \mathbb{Z}_r^2 \setminus T_\mu$. Let $f_{ij}, a_{ij}, \mathsf{Rec}$ be as in Lemma 3. Then there is a randomized encoding $\hat{g}^{a,\mu}$ of g^a with randomness and output size both $r(r-1)+1$. The randomized components of $\hat{g}^{a,\mu}$ have the form*

[11] We remark that the sum is exponential in r. We will however usually use r small enough that this local computation does not affect the overall runtime significantly.

[12] Take $j \coloneqq \min\{\bar{i} : x_i a_{i+1,\ldots,k}$ a factor of the summand for some $k \neq i+1\}$.

[13] The other elements of J are uniquely determined by g.

(i) $g_{ij}^{a,\mu} = f_{ij}$ *for* $(i,j) \in S_\mu$.
(ii) $g_{\mu\mu}^{a,\mu} = ax_\mu - b_\mu^{a,\mu}$, $g_{\mu j}^{a,\mu} = ax_\mu a_{\mu+1,\dots,j} - b_{\mu,\dots,j}^{a,\mu}$ *for* $j \neq \mu, \mu-1$.
(iii) $g_{ij}^{a,\mu} = x_i b_{i+1,\dots,j}^{a,\mu} - b_{i,\dots,j}^{a,\mu}$ *for* $(i,j) \in T_\mu \setminus (\{\mu\} \times \mathbb{Z}_r)$ *and* $j \neq i-1$.
(iv) $g_{\mathsf{add}}^{a,\mu} = ax_\mu a_{\mu+1,\dots,\mu-1} + \sum_{i \in \mathbb{Z}_r \setminus \{\mu\}} x_i b_{i+1,\dots,i-1}^{a,\mu} - b_{\mathsf{add}}^{a,\mu}$.

for randomness $b_{i,\dots,j}^{a,\mu}$ *for* $(i,j) \in T_\mu$ *and* $j \neq i-1$, *and* $b_{\mathsf{add}}^{a,\mu} = c + \sum_{J \subset \mathbb{Z}_r, |J|>1} (-1)^{|J|} b_J^{a,\mu}$ *where* $b_{i,\dots,j}^{a,\mu} \coloneqq a_{i,\dots,j}$ *for* $(i,j) \in S_\mu$.

Proof. We can simply copy the proof of Lemma 3 for the variables $(ax_\mu, x_j, j \neq \mu)$ and coefficients $b_*^{a,\mu}$ instead of a_*. Please note that again we have to choose the $b_{i,\dots,j}^{a,\mu}$ uniformly random from R and only $b_{\mathsf{add}}^{a,\mu}$ is structured. □

Remark 4. First note that we use the additional index μ to determine to which encoding the prefactor a is assigned. Moreover, observe that $r-1$ of the new terms have as leading coefficient a product of two random elements, i.e. the $ax_\mu a_{\mu+1,\dots,j}$ in $g_{\mu j}^{a,\mu}, g_{\mathsf{add}}^{a,\mu}$ for $j \neq \mu$. The other new encodings all have one random prefactor: $|T_\mu| - r$ encodings from (ii), (iii) as well as $r-1$ summands in $g_{\mathsf{add}}^{a,\mu}$.

Lemma 5. *Let* $h^{a,b}(x_0,\dots,x_{r-1}) = abx_{0,\dots,r-1} - c$ *for some* $a,b,c \in R$. *Let* $\mu,\nu \in \mathbb{Z}_r$ *be two fixed indices with* $\mu \neq \nu$. *Let* f_{ij}, $g_{ij}^{a,\mu}$, $g_{ij}^{b,\nu}$, $b_{ij}^{a,\mu}$, $b_{ij}^{b,\nu}$, S_μ, S_ν, T_μ, T_ν, Rec *be as in Lemmas 3 and 4. Then there is a randomized encoding* $\hat{h}^{a,b}$ *of* $h^{a,b}$ *with randomness and output size both* $r(r-1)+1$. *The randomized components of* $\hat{h}^{a,b}$ *have the form:*

(i) $h_{ij} = f_{ij}$ *for* $(i,j) \in S_\mu \cap S_\nu$
(ii) $h_{ij} = g_{ij}^{a,\mu}$ *for* $(i,j) \in T_\mu \setminus T_\nu$, $h_{ij} = g_{ij}^{b,\nu}$ *for* $(i,j) \in T_\nu \setminus T_\mu$
(iii) $h_{\mu j} = ax_\mu b_{\mu+1,\dots,j}^{a,\mu} - c_{\mu,\dots,j}$ *for* $(\mu,j) \in T_\nu, j \neq \mu, \mu-1$; $h_{\nu j} = bx_\nu b_{\nu+1,\dots,j}^{b,\nu} - c_{\nu,\dots,j}$ *for* $(\nu,j) \in T_\mu, j \neq \nu, \nu-1$;
(iv) $h_{ij} = x_i c_{i+1,\dots,j} - c_{i,\dots,j}$ *for* $\mu \neq i \neq \nu$ *and* $(i,j) \in T_\mu \cap T_\nu$ *and* $j \neq i, i-1$
(v) $h_{\mathsf{add}} = ax_\mu b_{\mu+1,\dots,\mu-1}^{a,\mu} + bx_\nu b_{\nu+1,\dots,\nu-1}^{b,\nu} + \sum_{i \in \mathbb{Z}_r \setminus \{\mu,\nu\}} x_i c_{i+1,\dots,i-1} - c_{\mathsf{add}}$

for randomness $c_{i,\dots,j}$ *for* $(i,j) \in T_\mu \cap T_\nu \wedge (j \neq i-1)$ *and* $c_{\mathsf{add}} = c + \sum_{J \subset \mathbb{Z}_r, |J|>1} (-1)^{|J|} c_J$ *where* $c_{i,\dots,j} \coloneqq a_{i,\dots,j}$ *for* $(i,j) \in S_\mu \cap S_\nu$, $c_{i,\dots,j} \coloneqq b_{i,\dots,j}^{a,\mu}$ *for* $(i,j) \in T_\mu \setminus T_\nu$, $c_{i,\dots,j} \coloneqq b_{i,\dots,j}^{b,\nu}$ *for* $(i,j) \in T_\nu \setminus T_\mu$.

Proof. Note that we can in fact consistently set $c_{i,\dots,j} = a_{i,\dots,j}$ for $(i,j) \in S_\mu \cap S_\nu$, since then $(i+1,j) \in S_\mu \cap S_\nu$ if $i \neq j$. Set $c_{i,\dots,j} = b_{i,\dots,j}^{a,\mu}$ for $(i,j) \in T_\mu \setminus T_\nu$, since then $(i+1,j) \in T_\mu \setminus T_\nu$ apart from $i \neq \mu$. Analogously $c_{i,\dots,j} = b_{i,\dots,j}^{b,\nu}$ for $(i,j) \in T_\nu \setminus T_\mu$. Furthermore, $(i,j) \in T_\mu \cap T_\nu \Rightarrow (i+1,j) \in T_\mu \cap T_\nu$ for $\mu \neq i \neq \nu$. In particular, $(i,i-1) \in T_\mu \cap T_\nu$. The claim now follows as in Lemma 3 with variables $(ax_\mu, bx_\nu, x_j : \mu \neq j \neq \nu)$ and randomness c_* instead of a_*. □

Remark 5. Observe that the two indices μ and ν are again used to assign the two prefactors a and b to the encodings linear in x_μ and in x_ν. Moreover, note that the number of new terms with two randomized prefactors is r, i.e. the terms $h_{\mu j}$ for $\overline{j-\nu} \leq \overline{\mu-\nu-1}$ and $h_{\nu j}$ for $\overline{j-\mu} \leq \overline{\nu-\mu-1}$. All other new encodings and summands thereof have one variable prefactor.

Remark 6. Please also note that in the previous lemmas, we always get one (unstructured) random number for each new component apart from the additive component. For the additive component, we get one structured random number.

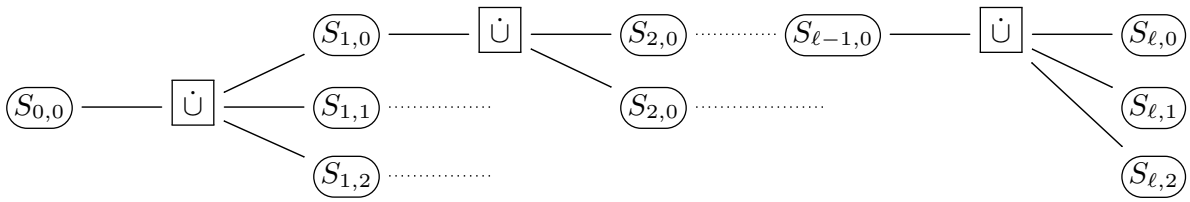


Fig. 3: Tree-like structure of a series of refinements of partitions.

The previous technical lemmas are combined as in the special case in Section 4.1. Namely, we partition our variables $x_0, \ldots, x_{m-1}$ into $r_1 \leq m$ sets, i.e. we choose a partition $\{0, \ldots, m-1\} =: S_{0,0} = \dot\bigcup_{i \in \mathbb{Z}_{r_1}} S_{1,i}$ and consider monomials $x_{S_{1,i}} = \prod_{j \in S_{1,i}} x_j$. Obviously we have $f(x_0, \ldots, x_{m-1}) = x_0 \cdots x_{m-1} = \prod_{j \in \mathbb{Z}_{r_1}} x_{S_{1,i}}$. Hence we can apply Lemma 3 with $r \leftarrow r_1, x_i \leftarrow x_{S_{1,i}}$. We receive encodings $(f_{ij}^{(1)}, f_{\mathsf{add}}^{(1)})$ which are linear in the $x_{S_{1,i}}$. Some of these encodings have no randomized leading coefficient (e.g. the $f_{ii}^{(1)}$). For these terms, we can apply Lemma 3 again by partitioning $S_{1,i}$ into smaller sets. For terms with one randomized leading coefficient like the $f_{ij}^{(1)}$ $(i \neq j \neq i-1)$ we analogously apply Lemma 4. By repeatedly applying the Lemmas 3 to 5 we then get encodings linear in some target elementary monomials $x_{S_{\ell,i}}$.

Formally, this approach corresponds to a series of refinements $S_{0,0} = \dot\bigcup_{i \in \mathbb{Z}_{r_k}} S_{k,i}$ of disjoint unions of non-empty sets for $1 = r_0 < r_1 < \cdots < r_\ell \leq m$, i.e. $\forall 0 < k \leq \ell \; \forall i \in \mathbb{Z}_{r_k} \; \exists i_0 \in \mathbb{Z}_{r_{k-1}} : S_{k,i} \subseteq S_{k-1,i_0}$. We get a tree structure visualized in Figure 3 where $I_{k,i} \coloneqq \{j \in \mathbb{Z}_{r_{k+1}} : S_{k+1,j} \subseteq S_{k,i}\}$ is the number of children of $S_{k,i}$. To later map the indices of these refinements to the generic indices in the lemmas, we fix a bijective map $\psi_{ki} : \mathbb{Z}_{|I_{k,i}|} \to I_{k,i}$ for all $0 \leq k \leq \ell$ and $0 \leq i < r_k$.

In terms of these general refinements, our construction (so far) defines for each monomial $x_{S_{k,i}}$ that occurs in the construction, a randomized encoding linear in the $x_{S_{k+1,j}}$ for $j \in I_{k,i}$. Now in order to combine these single randomized encodings into a randomized encoding of the whole $f(x_0, \ldots, x_{m-1}) = x_0 \cdots x_{m-1}$ we need to use concatenation and composition as described before. However, the classical concatenation Lemma 1 assumes independent randomized encodings to be concatenated. In constrast, our constructions in Lemmas 3 to 5 use the same encodings, e.g. the f_{ij} in Lemma 3 and in Lemma 4 (i), for different components. Fortunately, for the encodings that occur in Lemmas 3 to 5 this still leads to a secure concatenation, e.g. $((f_{ij})_{i-1 \neq j}, f_{\mathsf{add}}, (g_{ij}^{\mu,a})_{(i,j) \notin S_\mu}, g_{\mathsf{add}}^{\mu,a})$ is a randomized encoding of the concatenation $(f, g^a) = (x_0 \cdots x_{r-1}, a x_0 \cdots x_{r-1})$.[14] Formally, this property is described by:

Corollary 1. *Let f, g be two functions. Let $\hat f$ be a randomized encoding of f with additive component $\hat f_0$ and simulator Sim_f. Furthermore, let $\hat g$ be a randomized encoding of g with additive component $\hat g_0$ and simulator Sim_g. Assume that for all $i, j > 0$ $(\mathsf{Sim}_f)_i$ and $(\mathsf{Sim}_g)_j$ are independent uniformly random numbers. Let $J = \{j > 0 | \exists i > 0 : \hat f_i = \hat g_j\}$. Then $((\hat f_i)_{0 \leq i}, (\hat g_j)_{j \notin J})$ is a randomized encoding of $(\hat f, \hat g)$ with output size $k + k' - |J|$. Moreover, if $\hat f_0, \hat g_0$ map to the same (additive) group then $((\hat f_i)_{0<i}, (\hat g_j)_{j \notin J \cup \{0\}}, \hat f_0 + \hat g_0)$ is a randomized encoding of $f + g$ with output size $k + k' - |J| - 1$ and additive component $\hat f_0 + \hat g_0$.*

[14] Using encodings in different reconstructions is in general not secure (at least not for the straightforward combination of the simulators)—see Example 3.

Remark 7. We can repeatedly apply Corollary 1 to find a randomized encoding of the concatenation of many functions. E.g. if we use the randomized encoding of our monomial $x_0 \cdots x_{m-1}$ we get from Lemma 3 (beyond others) the components $f' = (x_{S_{1,i}} - a_i, (x_{S_{1,i}} a_{i+1,\ldots,j} - a_{i,\ldots,j})_{j \in \mathbb{Z}_{r_1} : j \neq i, i-1})$. If we now apply Lemma 3 to the first component and Lemma 4 (with some fixed $\mu \in \mathbb{Z}_{|I_{1,i}|}$) to all other components, then Corollary 1 (applied $(r_1 - 2)$ times) leads to a randomized encoding of the concatenation f' of output size $|I_{1,i}|^2 - |I_{1,i}| + 1 + (r_1 - 2)(|T_\mu| - |I_{1,i}| + 1)$. Please also note that exactly as in the special case, we see that the terms (i) in Lemma 4 are always already constructed by the corresponding Lemma 3 randomized encoding linear in the same terms; analogously for Lemma 5.

Remark 8. We can also use Corollary 1 to find a randomized encoding for additive terms like f_{add}. For example, if we take a randomized encoding of our monomial $x_0 \cdots x_{m-1}$ using Lemma 3 we get the additive component $f_{\mathsf{add}} = \sum_{i \in \mathbb{Z}_{r_1}} x_{S_{1,i}} a_{i+1,\ldots,i-1} - a_{\mathsf{add}}$. We assume that Lemma 3 has already been applied to the $x_{S_{1,i}}$, e.g. as part of the randomized encoding of f_{ii} and we are only interested, how many new outputs are needed to also construct f_{add}. Thus, if we apply Lemma 4 to each summand $x_{S_{1,i}} a_{i+1,\ldots,i-1}$ (where we consider once $x_{S_{1,0}} a_{1,\ldots,-1} - a_{\mathsf{add}}$ to account for the final constant), then we only need to construct the terms from (ii),(iii),(iv), since we assumed that (i) is already accounted for. Overall these are $r_1(|T_\mu| - r_1) + 1$ (additional) terms, where $r_1(|T_\mu| - r_1)$ comes from using Lemma 4 (ii), (iii) for each summand and the $+1$ comes from the sum of the r_1 additive (iv) terms that can be combined by Corollary 1 into one additive component.

Remark 9. Please also note that the previous Corollary 1 also applies to the randomness used in the additive components. Namely, if a_{add} is a summand of the additive component $\hat{f}_0$ and b_{add} is summand of the additive component $\hat{g}_0$, then $(a_{\mathsf{add}} + b_{\mathsf{add}})$ is obviously a summand of $\hat{f}_0 + \hat{g}_0$, although slightly more structured. In particular, even after applying the Corollary, we still have exactly one structured random number for each additive component and one unstructured random number for each other component of the overall randomized encoding.

In summary, we generate with our Lemma 3 a randomized encoding $\hat{f}^{(1)}$ of f linear in the $x_{S_{1,j}}$. We then generate for each component $\hat{f}^{(1)}_j$ of $\hat{f}^{(1)}$ a randomized encoding $\hat{f}^{(2)}_j$ linear in the $x_{S_{2,i}}$ using Lemmas 3 to 5 (and in the case of an additive term also Corollary 1). Corollary 1 allows us to concatenate the $\hat{f}^{(2)}_j$ into a randomized encoding $\hat{f}^{(2)}$ of $\hat{f}^{(1)}$. Finally the two encodings $\hat{f}^{(1)}$ and $\hat{f}^{(2)}$ can be composed with Lemma 2 to a randomized encoding of f linear in the $x_{S_{1,i}}$. We iterate over the previous steps until we arrive at a randomized encoding of f linear in the $x_{S_{\ell,j}}$. An algorithmic version of our construction is included in Protocol 2, where the output set contains the encodings of f linear in $x_{S_{\ell,j}}$. Please also consider Figure 8 which illustrates how the different encodings are combined under concatenation and composition.

4.4 Recursive Formula for Output Size

We next want to compute the output and randomness for *each* of our randomized encodings of $x_{S_{0,0}}$, i.e. for each choice of a series of refinements of partitions of $S_{0,0}$ or equivalently for each tree structure as in Figure 3. Since our randomized encodings were constructed iteratively, we will also develop an iterative formula first. To this end, let $N^0_{S_{k,j}}$ be the number of level ℓ encodings linear in $x_{S_{\ell,i}}, 0 \leq i < r_\ell$, needed to compute $x_{S_{k,j}} - c$ for some $c \in R$. Furthermore, let $N^1_{S_{k,j}}$ be the number of additional encodings needed to also construct $a x_{S_{k,j}} - c'$ for some $a, c' \in R$.

Finally, let $N^2_{S_{k,j}}$ be the number of yet additional encodings needed to construct $abx_{S_{k,j}} - c''$ for some $a, b, c'' \in R$. Recall that these are just the cases (a), (b), (c) discussed in the special case above. From Lemma 3 we then get

$$\begin{aligned} N^0_{S_{k,j}} &= \sum_{i \in I_{k,j}} N^0_{S_{k+1,i}} + (|I_{k,j}| - 2) \sum_{i \in I_{k,j}} N^1_{S_{k-1,i}} + \sum_{i \in I_{k,j}} (N^1_{S_{k+1,i}} - 1) + 1 \\ &= \sum_{i \in I_{k,j}} N^0_{S_{k+1,i}} + (|I_{k,j}| - 1) \sum_{i \in I_{k,j}} N^1_{S_{k+1,i}} - |I_{k,j}| + 1 \end{aligned} \tag{2}$$

where the first sum corresponds to the $f^{(k+1)}_{ii}$. The factor $(|I_{k,j}| - 2)$ comes from choices of $j \neq i, i-1$ for each i in the $f^{(k+1)}_{ij}$. The third sum accounts for the additive term as in Corollary 1 and Remark 8, i.e. $\sum_{i \in I_{k,j}} (N^1_{S_{k+1,i}} - 1)$ for (ii),(iii) in Lemma 4 plus one additional additive term. We further get

$$N^1_{S_{k,j}} = \sum_{i \in I_{k,j} \setminus \{\mu\}} N^1_{S_{k+1,i}} |T'_\mu \cap M_i| + (|I_{k,j}| - 1) N^2_{S_{k+1,\mu}} + N^1_{S_{k+1,\mu}} - |I_{k,j}| + 1 \tag{3}$$

where $M_\iota = \{\iota\} \times I_{k,j}$ and the $T'_\mu \coloneqq \psi_{kj}(T_{\psi^{-1}_{kj}(\mu)}), T'_\nu \coloneqq \psi_{kj}(T_{\psi^{-1}_{kj}(\nu)})$ are defined as in Lemma 4 using the natural identifications $\psi_{kj} : \mathbb{Z}_{|I_{k,j}|} \to I_{k,j}$.[15] We receive this term again from Lemmas 3 to 5, where the additive term contributes $N^2_{S_{k+1,\mu}} + \sum_{i \in I_{k,j} \setminus \{\mu\}} N^1_{S_{k+1,i}} - |I_{k,j}| + 1$. The intersection $|T'_\mu \cap M_i \cap \{(i,j) : j \neq i-1\}| = |T'_\mu \cap M_i| - 1$ together with the sum over $i \neq \mu$ accounts for the cases (iii) in Lemma 4, the single $N^1_{S_{k+1,\mu}}$ for $g^{(k+1),a,\mu}_{\mu\mu}$. We also have the additional $(|I_{k,j} - 2|) N^2_{S_{k+1,\mu}}$ for the $g^{(k+1),a,\mu}_{\mu j}$ for $j \neq \mu, \mu - 1$ in (ii) of Lemma 4. Altogether we get Equation (3). Furthermore, we have

$$N^2_{S_{k,j}} = \sum_{i \in I_{k,j}} N^{1+|\{i\} \cap \{\mu,\nu\}|}_{S_{k+1,i}} |M_i \cap T'_\mu \cap T'_\nu| - |I_{k,j}| + 1 \tag{4}$$

The additive term is again constructed as before, where the $-|I_{k,j}| + 1$ results from using a sum over all (v) components as in Corollary 1. The summands of the additive term are combined as before with the cases $j \neq i - 1$ of (iii) and (iv) of Lemma 5. Similarly, the $h^{(k+1)}_{\mu\mu}, h^{(k+1)}_{\nu\nu}$ terms complement the exclusions $j \neq \mu$ in the other cases. Using $M_\mu \cap T'_\mu = M_\mu$ and $M_\nu \cap T'_\nu = M_\nu$ one can quickly deduce Equation (4).

4.5 Application in MPC Protocols and Asymptotic Behavior

From Protocol 1 we already know how to use the new randomized encodings $\hat{f} = (y_l)_{0 \leq l < k}$ of $f(x_0, \ldots, x_{m-1}) = x_{S_{0,0}}$ in an MPC protocol. Following the discussion above, we know that the y_l consist of terms linear in $x_{S_{k,\ell}}$ for $j \in \mathbb{Z}_{r_\ell}$ and are of the form $f^{(l)}_*, g^{(l),*}_*, h^{(l)}_*$. Hence if we set $N^\gamma_{S_{\ell,j}} = 1$ for all $\gamma = 0, 1, 2$ (one for each y_l) Equations (2) to (4) allows us to compute the output size k. In our MPC Protocol 1 we have to send the resulting $k = N^0_{S_{0,0}}$ encodings plus the initial $|S_{0,0}| = m$ masked values $x_j - a_j$, i.e. we get bandwidth $N^0_{S_{0,0}} + m$.

[15] While $N^1_{S_{1,i}}$ and $N^2_{S_{k,j}}$ below depend on μ and ν, these indices can be chosen freely, i.e. we can choose to which components we want to assign the prefactors. For this reason, we decided to not mark the two numbers with another μ or ν index.

We have seen in Sections 4.2 and 4.3 that the y_l are multivariate polynomials in the input variables $x_0, \dots, x_{m-1}$. They do not necessarily satisfy (I)–(III) in Section 4.1 yet. However, recall that we can rewrite multivariate polynomials like y_l in terms of the masked values $x_j - a_j$ as in (1) and then (I)–(III) are satisfied. The coefficients b_e of this expansion in the $x_j - a_j$ are binomial tuples, which are polynomials in the a_j and the randomized prefactors of y_l. In addition to the (structured) randomness in these binomial tuples, our construction also needs the randomness from the $a_{ij}, b_{ij}, c_{ij}, a_{\mathsf{add}}, b_{\mathsf{add}}, c_{\mathsf{add}}$ that result from Lemmas 3 to 5 and Corollary 1.[16] Hence we can define a *polytuple* as follows:

Definition 2. *Let f be a multivariate polynomial in $x_0, \dots, x_{m-1}$ and $\hat{f}(x_0, \dots, x_{m-1}, \tilde{a}_0, \dots, \tilde{a}_{t'}) = (y_l)_{0 \le l < k}$ a randomized encoding of f constructed with our iterative approach, i.e. the $\tilde{a}_j$ are the $a_{i,\dots,j}, b_{i,\dots,j}, c_{i,\dots,j}, a_{\mathsf{add}}, b_{\mathsf{add}}, c_{\mathsf{add}}$ which result from Lemmas 3 to 5 and Corollary 1. Then a* polytuple $[\![\hat{a}]\!]$ *to $\hat{f}$ consists of a shared structured random number $[\![\tilde{a}_j]\!]$ for each $\tilde{a}_j$, $0 \le j \le t'$, and one binomial tuple for each y_l, $0 \le l < k$.*

Remark 10. Recall from Section 3.4 that a term $x_S - a_S$ can be computed with a $2^{|S|}-1$ binomial tuple for any finite set S; a term $ax_S - b_S$, as well as a term $abx_S + c_S$ for randomness $a, b, b_S, c_S \in R$, each need a binomial tuple of size $2^{|S|}$ compensating for the additional prefactor(s), i.e. in the notation of Section 3.4 a tuple $(ab_e)_{e \in E}$ or $(abb_e)_{e \in E}$.

Since we know from Lemmas 3 to 5 and the subsequent remarks that for each encoding we get exactly one new (possibly structured) random variable, we can also use the iterative formulas in Equations (2) to (4) to compute the polytuple size. Namely, if we replace $N^{\gamma}_{S_{k,j}}, \gamma = 0, 1, 2$, in Equations (2) to (4) by the corresponding tuple sizes $T^{\gamma}_{S_{k,j}}$ and set $T^0_{S_{\ell,j}} + 1 = T^1_{S_{\ell,j}} = T^2_{S_{\ell,j}} = 2^{|S_{\ell,j}|}$, then $T^{\gamma}_{S_{0,0}}$ will be the tuple size needed to compute $x_{S_{0,0}} = x_0 \cdots x_{m-1}$.

Please note that the size of a polytuple, as well as the output size of the randomized encoding strongly depend on the chosen tree structure (cf. Figure 3), i.e. partitions. To better understand how the tree structure affects the asymptotic behavior of the bandwidth and tuple size, we consider trees with a fixed number $b = |I_{k,j}| \ge 1$ of factors multiplied in each node. Hence we can compute $x_{0,\dots,m-1}$ for $m = \lambda b^n$ iteratively with $S_{n-k,j} = \{\lambda b^k \cdot j + i : 0 \le i < \lambda b^k\}, 0 \le j < b^{n-k}, 0 \le k \le n$, i.e. each degree b^k term splits into b encodings of degree b^{k-1} until we reach a level of elementary randomized encodings of degree $\lambda \ge 1$. For an explicit calculation in the special case $b = 2, n = 3, \lambda = 2$ we refer to Example 5. Now we can state the main result on the asymptotic behavior, which we prove in Appendix A.

Theorem 1. *Let $\lambda, b, S_{k,j}$ be defined as before. A product of $m = \lambda b^n$ shared inputs can be constructed with a polytuple of size $\mathcal{O}\left(2^{\lambda}\left(\frac{b^2+1}{2}\right)^n\right)$ with bandwidth $\mathcal{O}\left(\left(\frac{b^2+1}{2}\right)^n\right)$. In the special case $b = 2$, one only needs a tuple of size $2^{n-2}((2^{\lambda}-1)n^2 + (2^{\lambda+2} - 2^{\lambda} + 1)n + 4(2^{\lambda} - 2)) + 1$. For $b = 2$, the bandwidth becomes $2^n n + 1 + m$.*

Remark 11. If we fix λ small, e.g. $\lambda \le 3$, the case $b = 2$ leads to a bandwidth in $\mathcal{O}(m \log(m))$ and a tuple size in $\mathcal{O}(m \log(m)^2)$ while in all cases $b > 2$ both values are not even in $\mathcal{O}(m^2)$ (cf. Proof Theorem 1 and Lemma 6 in Appendix A). Furthermore, we remark that for a mixed number of factors going into a node the complexity will be dominated by the largest degree that occurs in a significant fraction of encodings. Finally, note that the complexity analysis also covers the case of a binomial tuple for $b = 1$.

[16] Recall from Section 3 that we also include terms deterministic in random variables in our randomness space.

Polynomials in Several Variables. Up to this point, we mainly discussed the computation of products $x_0 \cdots x_{m-1}$. However, the previous results directly transfer to general monomials $x^{\boldsymbol{d}} = x_0^{d_0} \cdots x_{m-1}^{d_{m-1}}, \boldsymbol{d} = (d_0, \ldots, d_{m-1})$ simply by replacing the variables x_i in the randomized encoding by $x_i^{d_i}$. A component of the randomized encoding will then be linear in $\prod_{s \in S_{\ell,j}} x_s^{d_s}$ and can still be constructed using a binomial tuple. From Section 3.4 we know that $T^0_{S_{\ell,j}} = T^1_{S_{\ell,j}} - 1 = T^2_{S_{\ell,j}} - 1 = \prod_{s \in S_{\ell,j}} (d_s + 1) - 1$. For the special case where $|S_{\ell,j}| = 1$, e.g. $S_{\ell,j} = \{j\}$, we have $T^0_{\{j\}} = d_j + 1$, i.e. $[\![x_j^{d_j} - a'_{j,d_j}]\!] = -[\![a'_{j,d_j}]\!] + \sum_{i=0}^{d_j} [\![a_j^i]\!](x_j - a_j)^{d_j - i}$ for a new mask a'_{j,d_j}. Then the tuple size needed to compute $x_0^{d_0} \cdots x_{m-1}^{d_m}$ follows recursively from Equations (2) to (4). If $d_j = d/m \in \mathbb{N}$ the tuple size to compute $x_0^{d_0} \cdots x_{m-1}^{d_{m-1}}$ for $m = 2^n$ becomes $2^{n-2}((\frac{d}{m})n^2 + (3\frac{d}{m} + 4)n + 4\frac{d}{m} - 4) + 1$. For details we refer to the proof of Theorem 1 in Appendix A which contains the formulas (and proof thereof) whenever $T^1_{S_{\ell,j}} = T^2_{S_{\ell,j}}$. The result shows that in the total degree $d = \sum_{j=0}^{m-1} d_j$ we can get down to complexity $\mathcal{O}(d \log(m)^2)$ in the tuple size. The same bound on the complexity also holds for all other cases with $d = \sum_{j=0}^{m-1} d_j$ since we can choose μ, ν in Equations (3) and (4) always such that the encodings with randomized coefficients are linear in those $x_{S_{\ell,j}}$ for which $T^1_{S_{\ell,j}} = T^2_{S_{\ell,j}}$ is minimal, i.e. from the cases with $d_j \leq d/m$. Please recall that $x_0^{d_0} \cdots x_{m-1}^{d_{m-1}}$ was already discussed in the introduction in Table 1.

Finally, we can combine the randomized encodings (and corresponding polytuples) for different monomials in a general polynomial f with Corollary 1. Namely, if we have two randomized encodings f, g (as constructed before), we need to generate common components only once and we can add the components corresponding to $\hat{f}_0$ and $\hat{g}_0$ in Corollary 1. Observe that all our encodings have the specific form expected by Corollary 1, i.e. have an additive component. Overall we find for any multivariate polynomial f a randomized encoding and a corresponding polytuple.

4.6 Composability and Security

From Section 4.2 we know how to evaluate a polynomial $f(x_0, \ldots, x_{m-1})$ in a single round using polytuples. With our MPC protocol $\Pi_{\text{polynomial}}$ presented in Section 4.1 (cf. also Protocol 5), we are able to do this in three different ways: (i) compute $f(x_0, \ldots, x_{m-1})$ publicly (i.e. the result is an output of the function to be evaluated with MPC), (ii) compute $[\![f(x_0, \ldots, x_{m-1})]\!]$ (this can be used in other subprotocols that require their inputs as shares), and (iii) compute $f(x_0, \ldots, x_{m-1}) - b$ where b is part of the tuple for another polynomial g; this allows our protocol to be used in a multi-round fashion. While (i) and (ii) are straightforward applications of the results from the previous subsections, we want to take a closer look at the multi-round use, which allows a different form of tradeoff. Namely, we allow a (slightly) larger number of communication rounds but can therefore further reduce the tuple size and bandwidth.

Multi-Round Evaluation. Assume the parties have agreed on a series of polynomials $f_j, 0 \leq j < m$ with input tuples X_j (not-necessarily disjoint) and a polynomial f in m variables. They want to compute $f(f_0(X_0), \ldots, f_{m-1}(X_{m-1}))$. Thus, they agree on one of our randomized encodings for each f_j and f. The parties construct the corresponding polytuples $[\![A_j]\!], 0 \leq j < m$ (for each f_j) and $[\![A]\!]$ (for f) in the preprocessing phase and receive inputs $[\![X_j]\!]$ in the input phase. They run $\Pi_{\text{polynomial}}(X_j, f_j, \mathsf{continuation} \coloneqq (f, j))$ in parallel to receive $(x_\iota - a_\iota), 0 \leq \iota < |X_j|, 0 \leq j < m$ in a single broadcast round. Then the parties locally compute the shares of the elementary encodings and adjust an additive component by $[\![a_j]\!]$ such that after the next broadcast every party can locally compute the public values $z_j \coloneqq f_j(X_j) - a_j$. Finally,

they call $\Pi_{\text{polynomial}}((z_1, \ldots, z_m), f, \mathsf{continuation} \coloneqq \text{open})$. Observe that in this call, the first step of $\Pi_{\text{polynomial}}$ does not require any opening of elements as all z_j are already public masked values.

Remark 12. Our protocol is also compatible with techniques used in Turbospeedz [BENO19] and ABY2.0 [PSSY21] that use function-dependent preprocessing. This allows to reduce the online bandwidth even more. As an extreme case, one would only have to open the randomized encoding without the $x_j - a_j$ which are then already accounted for. Using only Beaver multiplication (or binomial tuples), this would exactly correspond to the complexity of ABY2.0 or Turbospeedz.

In Sections 4.2 to 4.5 we have seen that by suitably choosing the randomized encodings and corresponding polytuples, we can trade-off bandwidth and tuple size while keeping the round complexity minimal. The multi-round feature adds additional flexibility to our online phase. In particular, it allows us to increase the round complexity slightly to prevent possible performance bottlenecks in bandwidth and tuple size. Figure 1 illustrates this tradeoff between round complexity, bandwidth, and tuple size. Please also see Example 1 in Appendix A for an explicit example. We remark that once the polynomial to be evaluated and the network setup are known, a compiler can use the exact calculations of tuple size and bandwidth from Equation (2) to determine the best performing polytuple solution before the actual computation starts. Furthermore, ideal solutions for classical and regularly used setups can be hard-coded.

Security. Our protocol $\Pi_{\text{polynomial}}$ and the resulting full online protocol[17] Π_{online} (cf. Protocol 5) are secure and composable in the sense of *universal composability* (UC) [Can01], i.e. they can be combined with other MPC protocols, while still giving the same guarantees as an idealized protocol (a so-called functionality). The corresponding ideal functionalities are included in Appendix B.

Let $[\![X]\!]$ be a tuple of authenticated inputs to a polynomial f and $[\![A]\!]$ the respective tuple. Intuitively, the security of our approach can be argued as follows: All opened values apart from one additive component of the randomized encoding are masked with a new random element from $[\![A]\!]$, i.e. they are encrypted with a one-time pad and hence are information-theoretically secure. The final additive encoding contains the result minus a public constant (constructed from the other (pseudo)random components of the randomized encoding). In particular, it contains no more information than the result itself.

All values that are opened are authenticated and thus their integrity can be checked with the usual aggregated MAC check (cf. Protocol 7; recall that we now consider R to be a finite field). In particular, our MAC check Π_{CheckMAC} is chosen identical to the classical MAC-check in [DKL$^+$13]. Formally, we then have the following security result for the online protocol[18] Π_{online} in Protocol 5:

Theorem 2. *The protocol Π_{online} realizes $\mathcal{F}_{\text{online}}$ in the $(\mathcal{F}_{[\![\cdot]\!]}, \mathcal{F}_{\text{random}}, \mathcal{F}_{\text{commit}})$-hybrid model with statistical security against any active adversary corrupting up to $n-1$ parties.*

Proof. The proof of this theorem is mostly the same as the security proofs for the corresponding online protocols in [DKL$^+$13, DPSZ12]. Both construct a suitable simulator, e.g. [DKL$^+$13, Fig. 22]. The only difference for a simulator in our protocol is in polynomial operations that are opened (i.e. calls to $\Pi_{\text{polynomial}}$ with $\mathsf{continuation} = \text{open}$). Recall that the simulator works on

[17] Recall that apart from the $\Pi_{\text{polynomial}}$ subprotocol, our online protocol Π_{online} coincides with the online protocols from other SPDZ-like protocols like [DKL$^+$13, KPR18].

random inputs (instead of the real inputs for honest (input) parties) and simulates the protocol run with these inputs. It will then receive an output z of the simulation that is most likely wrong. However, the ideal functionality $\mathcal{F}_{\text{online}}$ provides the simulator with the real output y. The simulator adjusts the share of the additive encoding[19] y_0 of one (simulated) honest party P_i by $\Delta = y - z$, i.e. $[y_0]_i \to [y_0]_i + \Delta$. Since the simulator also knows the MAC key α, it can change $[\alpha y_0]_i \to [\alpha y_0]_i + \alpha\Delta$. Thus the MAC check for the result will pass (if corrupted parties did not misbehave) and the result will be the same in the real and ideal world. □

4.7 The Generation of Polytuples

In the previous paragraphs, we have seen how to build an actively secure MPC online Protocol 5 which consumes polytuples. Of course, the polytuples have to be generated first in an offline phase, which can run well before the actual input data (the x_j) becomes available. Since polytuples are entry-wise just multivariate polynomials in random numbers, the parties can invoke any MPC protocol that can provide (authenticated) shares of such terms. For example, for an actively secure offline phase we can plug in any of the protocols [DPSZ12,KOS16,KPR18,Rei+23] to first generate a sufficient number of Beaver triples. The parties can then use these Beaver triples to multiply shared random numbers, e.g. they run the standard online protocol within the offline phase on the random numbers (instead of actual inputs). Hence they can construct each entry of the polytuple.

The number of Beaver triples needed for this approach can again be computed by an iterative formula. The result are the Eqs. (2) to (4) each shifted $+|I_{k,j}|$. Recall from Corollary 1 that we did combine all additive terms into one constant and hence reduced the output and tuple size by $-|I_{k,j}|$. At the same time, the new additive term became more complex, namely a sum of the original monomials in the separate additive components. Even after combining the additive terms, we still need to build each of these monomials with Beaver triples. Thus the reduction of output and tuple size does not carry over to this generic offline approach and we have to add $|I_{k,j}|$ in the iterative formulas.

Exactly as in the proof of Theorem 1 we can then deduce that the number of Beaver triples needed (in the case $b = 2$ of binary trees) is still in $\mathcal{O}(m \log(d)^2)$ but with slightly larger constant. For example, in the case $b = 2$ we then need $2^{n-2}((\frac{d}{m}+1)n^2 + (3\frac{d}{m}-1)n + 4\frac{d}{m}) - 1$ Beaver triples if we use $d_i - 1$ Beaver triples to compute $\left[a^{d_1}\right]$ from $[a]$—of course, this is a rough estimate given that we often can compute the power with around $\log(d_1)$ Beaver triples (cf. also Remark 14 in Appendix A).

To simply plugin established offline protocols comes with certain advantages, e.g. that implementations already exist and that we can profit from their future optimizations. However, this approach is not optimized for the use with polytuples. In Appendices C and D we therefore present different new solutions for an actively secure tuple generation (e.g. an extended sacrificing Protocol 10).

Finally, please recall that our approach is not restricted to the case of binary trees or 1(+1) round protocols. In particular, if the generation of $\mathcal{O}(m \log(d)^2)$ Beaver triples is too slow, the parties can use a different number of rounds and different randomized encodings to get an ideal performance for their use case.

[19] Recall that our construction always comes with an additive encoding.

Fig. 4: The left diagram shows the bandwidth overhead of the polytuple plugin offline phase compared to classical SPDZ-like protocols for the computation of $x_0 \cdots x_{m-1}$. The right diagram shows the corresponding runtime overhead. For the blue line we used the LowGear offline protocol [KPR18], for the red dotted line the MASCOT protocol [KOS16].

5 Implementation and Evaluation

To illustrate the practicality of our approach, we implemented the online phase in the MP-SPDZ framework [Kel20] and ran several benchmarks. Furthermore, we implemented the plugin offline phase from Section 4.7 which uses Beaver triples to generate the polytuples. Our implementations are available at [Code]. These first benchmarks show that we can outperform the standard Beaver triple-based approach in the online phase for all tested applications. Our benchmarks include (i) evaluation of multivariate polynomials, (ii) establishing a ranking of inputs (e.g. for auctions or e-voting), and (iii) evaluating neural networks. We ran the experiments on a single machine (laptop with an i7-8565U CPU, 1.80 GHz) where each party runs on a single core/thread. We simulated different network settings for $n = 2$ parties with standard Linux tools (see Appendix G for details). All tested latency settings are rather conservative and roughly correspond to parties located in the same country or continent. The tested latencies are significantly lower than the 40 ms assumed in the WAN setting (e.g. in [ON20]). The trends in all benchmarks show that our approach will perform even better in such a setting.

With our implementation, we added elementary operations for powers and products to MP-SPDZ. We use polytuples of minimal tuple size as in Theorem 1 for $b = 2$. Furthermore, we implemented the case $b = m$, i.e. the case where polytuples become binomial tuples. For both variants, we also implemented a prefix variant (along Appendix E) used for comparison in our benchmarks below. Moreover, our implementation supports MP-SPDZ's parallelism model: arbitrarily many operations of the same type can be combined and executed in one step (reducing the number of communication rounds).

Next, we describe our test applications and discuss the results of our benchmarks. We always compare our implementation for $b = 2$ against the state-of-the-art implementation from MP-SPDZ. We do not compare to the binomial tuples case since first benchmarks showed that the local computation times for the tuple production are beyond practical (as expected by the large tuple size).

Polynomial Evaluation. As an example of a polynomial evaluation, we chose the power series expansion of a multivariate Gauss functions $\exp(-\langle x, x\rangle/2)$ up to degree d in each variable. This polynomial is then simply evaluated by computing all needed (prefix) powers of all variables and multiplying them with our polytuples. We compare this to the same computation with standard (Beaver triple-based) tools included in MP-SPDZ. Figure 5 and Figure 9 show the results for this benchmark. Our approach has a clear advantage in runtime—even for very small network delays of only 2 ms. Note that also the bandwidth is lower with our approach. For the Beaver-based implementation, we can clearly see the effect of a logarithmic number of rounds on the runtime, while our approach has an almost constant runtime (in the degree of the polynomial).

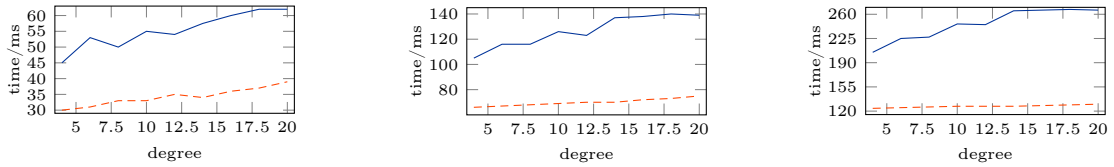
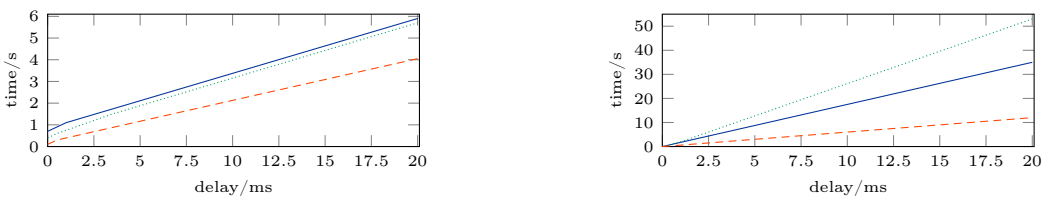


Fig. 5: Benchmarks for Gaussian with 32 variables with 2ms (left), 5ms (middle), 10ms (right) delay; (blue: default MP-SPDZ implementation, orange/dashed: ours).

Rankings. For auctions (or e-voting), one often needs to compute a ranking of the bids (or votes) and reveal the top k results (e.g. with $k = 1$ only the highest bid or the candidate with the most votes). There are several established methods to compute these rankings. For our evaluation, we chose two approaches, one purely based on inequality tests and one which uses equality and inequality tests. In order to use our new protocols to speed up the comparison we use bit-wise comparisons as in [DFK+06] which allow us to employ polytuples. For details, we refer to Appendices E and F. We benchmarked both approaches with our polytuples-based protocol and compare them to the respective default implementation in MP-SPDZ (based on the protocols with logarithmic complexity in [CdH10]; with and without edabits [EGK+20] to speed up the comparison). We compute rankings of $m = 40$ items (bids or candidates). The benchmark results in Fig. 6 show that our new approach is faster than the others.

Remark 13. SPDZ is a protocol originally designed for an arithmetic circuit evaluation and not for comparisons. In particular, there other MPC approaches better suited for some types of comparisons. However, our goal is to extend SPDZ and hence in particular to avoid expensive conversations to some other scheme. We therefore decided to compare our evaluation for comparisons also to SPDZ, although there are other competitive MPC protocols.

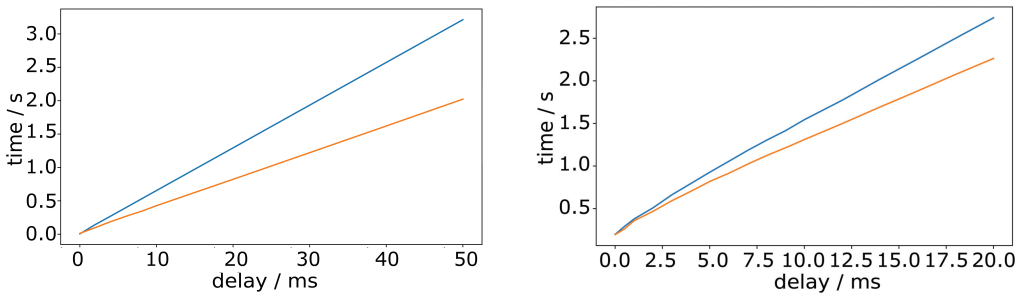
Neural Networks. Among others, MP-SPDZ [Kel20] contains examples of deep neural networks. For our benchmarks, we ran the networks labeled A [MZ17], B [LJLA17], C [LBBH98], and D [RWT+18] (as in [KS21, WGC19]). Each of these networks has a final ArgMax layer (see Appendix F for the specific layers). Replacing *only this single layer* with a polytuple-based comparison (see Appendix F for details) can already have a noticeable impact on the overall runtime of the network, as can be seen in Fig. 7. We also remark that a bandwidth rate restriction does not affect the performance and hence the theoretical bandwidth overhead of the polytuples approach is negligible in our example (see e.g. Fig. 10 in Appendix G).



(a) Using pairwise inequality tests.

(b) Using inequality and equality tests.

Fig. 6: Benchmarks for rankings (blue: default MP-SPDZ implementation, orange/dashed: ours, green/dotted: MP-SPDZ with edabits [EGK+20]).



(a) ArgMax Layer, unlimited rate. **(b)** Network A [MZ17].

Fig. 7: Benchmarks for an ArgMax layer and the evaluation of a sample neural network included in MP-SPDZ [Kel20] as network A (cf. [RWT+18]; blue: default MP-SPDZ, orange: ours) both without bandwidth restriction. For further benchmarks see Fig. 10.

Tuple Generation. Finally, we also benchmarked the offline phase for the plugin approach described in Section 4.7. Our first results in Fig. 4 confirm our theoretical results of Section 4, i.e. we get a log-linear overhead over SPDZ independent of the employed offline protocol (Overdrive LowGear [KPR18] and MASCOT [KOS16]). As our focus is on applications where the offline phase is not time-critical, we leave further benchmarking of the offline phase and possibly improving the polytuple generation (e.g. as in Appendix C) to future work.

Overall, our evaluation shows that our approach has a clear performance advantage over SPDZ in the online phase for classical sample applications like the evaluation of multivariate polynomials or comparisons.

Acknowledgments. This research was supported by the CRYPTECS project founded by the German Federal Ministry of Education and Research under Grant Agreement No. 16KIS1441 and by the French National Research Agency under Grant Agreement No. ANR-20-CYAL-0006 and by Advantest as part of the Graduate School "Intelligent Methods for Test and Reliability" (GS-IMTR) at the University of Stuttgart. Additionally, this research was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Fundation) – 411720488. Toomas Krips was partly supported by the Estonian Research Council, ETAG, through grant PRG 946. We furthermore thank Simon Egger for his help and valuable remarks.

References

ABT18. B. Applebaum, Z. Brakerski, and R. Tsabary. Perfect secure computation in two rounds. In *Theory of Cryptography*, pages 152–174. Springer, 2018.

ABT19. B. Applebaum, Z. Brakerski, and R. Tsabary. Degree 2 is complete for the round-complexity of malicious mpc. In *EUROCRYPT 2019*, pages 504–531. Springer, 2019.

AIK06. B. Applebaum, Y. Ishai, and E. Kushilevitz. Cryptography in NC°. *SIAM Journal on Computing*, 36(4):845–888, 2006.

AKP20. B. Applebaum, E. Kachlon, and A. Patra. The round complexity of perfect MPC with active security and optimal resiliency. In *IEEE FOCS*, pages 1277–1284, 2020.

BB89. J. Bar-Ilan and D. Beaver. Non-cryptographic fault-tolerant computing in constant number of rounds of interaction. In *PODC 1989*, pages 201–209. ACM, 1989.

BBC+19. D. Boneh, E. Boyle, H. Corrigan-Gibbs, N. Gilboa, and Y. Ishai. Zero-knowledge proofs on secret-shared data via fully linear pcps. In *CRYPTO 2019*, pages 67–97. Springer, 2019.

BCG+18. C. Boura, I. Chillotti, N. Gama, D. Jetchev, S. Peceny, and A. Petric. High-precision privacy-preserving real-valued function evaluation. In *FC 2018*, pages 183–202. Springer, 2018.

BCG+19. E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, and P. Scholl. Efficient Pseudorandom Correlation Generators: Silent OT Extension and More. In *CRYPTO 2019*, pages 489–518. Springer, 2019.

BCS20. C. Baum, D. Cozzo, and N. P. Smart. Using TopGear in Overdrive: A more efficient ZKPoK for SPDZ. In *SAC 2019*, pages 274–302. Springer, 2020.

BD19. D. Bitan and S. Dolev. Optimal-Round Preprocessing-MPC via Polynomial Representation and Distributed Random Matrix (extended abstract). *IACR Cryptol. ePrint Arch.*, 2019:1024, 2019.

BDG+17. X. Bultel, M. L. Das, H. Gajera, D. Gérault, M. Giraud, and P. Lafourcade. Verifiable Private Polynomial Evaluation. In *ProvSec 2017*, pages 487–506. Springer, 2017.

BDO14. C. Baum, I. Damgård, and C. Orlandi. Publicly Auditable Secure Multi-Party Computation. In *SCN 2014*, pages 175–196. Springer, 2014.

BDOZ11. R. Bendlin, I. Damgård, C. Orlandi, and S. Zakarias. Semi-homomorphic Encryption and Multiparty Computation. In *EUROCRYPT*, pages 169–188. Springer, 2011.

Bea92. D. Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO '91*, pages 420–432. Springer, 1992.

BENO19. A. Ben-Efraim, M. Nielsen, and E. Omri. Turbospeedz: Double Your Online SPDZ! Improving SPDZ Using Function Dependent Preprocessing. In *ACNS 2019*, pages 530–549. Springer, 2019.

BGV12. Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (Leveled) Fully Homomorphic Encryption Without Bootstrapping. In *ITCS 2012*, pages 309–325. ACM, 2012.

BNTW12. D. Bogdanov, M. Niitsoo, T. Toft, and J. Willemson. High-performance secure multi-party computation for data mining applications. *Int. J. Inf. Sec.*, 11(6):403–418, 2012.

BOS16. C. Baum, E. Orsini, and P. Scholl. Efficient Secure Multiparty Computation with Identifiable Abort. In *TCC 2016-B*, pages 461–490, 2016.

Can01. R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *FOCS 2001*, pages 136–145. IEEE Computer Society, 2001.

CB17. H. Corrigan-Gibbs and D. Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *NSDI 2017*, pages 259–282. USENIX Association, 2017.

CD01. R. Cramer and I. Damgård. Secure Distributed Linear Algebra in a Constant Number of Rounds. In *CRYPTO 2001*, pages 119–136. Springer, 2001.

CdH10. O. Catrina and S. de Hoogh. Improved Primitives for Secure Multiparty Integer Computation. In *SCN 2010*, pages 182–199. Springer, 2010.

CFIK03. R. Cramer, S. Fehr, Y. Ishai, and E. Kushilevitz. Efficient multi-party computation over rings. In E. Biham, editor, *Advances in Cryptology — EUROCRYPT 2003*, pages 596–613, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

CFY17. R. K. Cunningham, B. Fuller, and S. Yakoubov. Catching MPC Cheaters: Identification and Openability. In *ICITS 2017*, pages 110–134. Springer, 2017.

CKR+20. H. Chen, M. Kim, I. P. Razenshteyn, D. Rotaru, Y. Song, and S. Wagh. Maliciously Secure Matrix Multiplication with Applications to Private Deep Learning. In *ASIACRYPT 2020*, pages 31–59. Springer, 2020.

Code. P. Reisert, M. Rivinius, T. Krips, S. Hasler, and R. Küsters. Implementation to *Actively Secure Polynomial Evaluation from Shared Polynomial Encodings*, 2024. Website of the Insitute of Information Security Stuttgart.

Cou19. G. Couteau. A note on the communication complexity of multiparty computation in the correlated randomness model. In *EUROCRYPT*, pages 473–503. Springer, 2019.

CS10. O. Catrina and A. Saxena. Secure Computation with Fixed-Point Numbers. In *FC 2010*, pages 35–50. Springer, 2010.

CWB18. H. Cho, D. Wu, and B. Berger. Secure genome-wide association analysis using multiparty computation, supplementary notes 3. *Nat. Biotechnol.*, 36(6):547–551, 2018.

CZC+21. H. Cui, K. Zhang, Y. Chen, Z. Liu, and Y. Yu. MPC-in-Multi-Heads: A Multi-Prover Zero-Knowledge Proof System. In *ESORICS*, pages 332–351. Springer, 2021.

Dah17. M. Dahl. Cryptography and machine learning, 2017. Blog on the SPDZ protocol - part 2.

DFK+06. I. Damgård, M. Fitzi, E. Kiltz, J. B. Nielsen, and T. Toft. Unconditionally Secure Constant-Rounds Multi-party Computation for Equality, Comparison, Bits and Exponentiation. In *TCC 2006*, pages 285–304. Springer, 2006.

DKL+13. I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart. Practical covertly secure MPC for dishonest majority – or: Breaking the SPDZ limits. In *ESORICS 2013*, pages 1–18. Springer, 2013.

DMRY11. D. Dachman-Soled, T. Malkin, M. Raykova, and M. Yung. Secure Efficient Multiparty Computing of Multivariate Polynomials and Applications. In *ACNS 2011*, pages 130–146, 2011.
DPSZ12. I. Damgård, V. Pastro, N. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, pages 643–662. Springer, 2012.
EGK+20. D. Escudero, S. Ghosh, M. Keller, R. Rachuri, and P. Scholl. Improved Primitives for MPC over Mixed Arithmetic-Binary Circuits. In *CRYPTO 2020*, pages 823–852. Springer, 2020.
EOYN21. R. Eriguchi, K. Ohara, S. Yamada, and K. Nuida. Non-interactive Secure Multiparty Computation for Symmetric Functions, Revisited: More Efficient Constructions and Extensions. In *CRYPTO 2021*, pages 305–334. Springer, 2021.
FM10. M. K. Franklin and P. Mohassel. Efficient and Secure Evaluation of Multivariate Polynomials and Applications. In *ACNS 2010*, pages 236–254, 2010.
FM19. D. Falamas and K. Marton. Performance impact analysis of rounds and amounts of communication in secure multiparty computation based on secret sharing. In *RoEduNet 2019*, pages 1–6. IEEE, 2019.
Gen09. C. Gentry. *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford University, 2009.
GHM+21. K. Gjøsteen, T. Haines, J. Müller, P. B. Rønne, and T. Silde. Verifiable Decryption in the Head. *IACR Cryptol. ePrint Arch.*, page 558, 2021.
GHS12. C. Gentry, S. Halevi, and N. P. Smart. Fully homomorphic encryption with polylog overhead. In *EUROCRYPT 2012*, pages 465–482. Springer, 2012.
GM09. G. Gavin and M. Minier. Oblivious Multi-variate Polynomial Evaluation. In *INDOCRYPT 2009*, pages 430–442. Springer, 2009.
GMRW13. S. D. Gordon, T. Malkin, M. Rosulek, and H. Wee. Multi-party Computation of Polynomials and Branching Programs without Simultaneous Interaction. In *EUROCRYPT 2013*, pages 575–591. Springer, 2013.
GPS12. H. Ghodosi, J. Pieprzyk, and R. Steinfeld. Multi-party computation with conversion of secret sharing. *Des. Codes Cryptogr.*, 62(3):259–272, 2012.
HHPV21. S. Halevi, C. Hazay, A. Polychroniadou, and M. Venkitasubramaniam. Round-Optimal Secure Multiparty Computation. *J. Cryptol.*, 34(3):19, 2021.
HIJ+16. S. Halevi, Y. Ishai, A. Jain, E. Kushilevitz, and T. Rabin. Secure Multiparty Computation with General Interaction Patterns. In *ITCS*, pages 157–168. ACM, 2016.
HIJ+17. S. Halevi, Y. Ishai, A. Jain, I. Komargodski, A. Sahai, and E. Yogev. Non-Interactive Multiparty Computation Without Correlated Randomness. In *ASIACRYPT 2017*, pages 181–211. Springer, 2017.
HRRK24. S. Hasler, P. Reisert, M. Rivinius, and R. Küsters. Multipars: Reduced-Communication MPC over Z2k. *Proceedings on Privacy Enhancing Technologies*, (2):5–28, 2024.
IK00. Y. Ishai and E. Kushilevitz. Randomizing polynomials: A new representation with applications to round-efficient secure computation. In *FOCS*, pages 294–304, 2000.
IKM+13. Y. Ishai, E. Kushilevitz, S. Meldgaard, C. Orlandi, and A. Paskin-Cherniavsky. On the Power of Correlated Randomness in Secure Computation. In *TCC 2013*, pages 600–620. Springer, 2013.
Ish13. Y. Ishai. Randomization techniques for secure computation. In *Secure Multi-Party Computation*, 2013.
KBTJ19. R. Karl, T. Burchfield, J. Takeshita, and T. Jung. Non-Interactive MPC with Trusted Hardware Secure Against Residual Function Attacks. In *SecureComm 2019*, pages 425–439. Springer, 2019.
Kel20. M. Keller. MP-SPDZ: A Versatile Framework for Multi-Party Computation. In *CCS '20*, pages 1575–1590. ACM, 2020.
KLM+20. R. Küsters, J. Liedtke, J. Müller, D. Rausch, and A. Vogt. Ordinos: A Verifiable Tally-Hiding E-Voting System. In *EuroS&P 2020*, pages 216–235. IEEE, 2020.
Kol05. V. Kolesnikov. Gate evaluation secret sharing and secure one-round two-party computation. In *ASIACRYPT 2005*, pages 136–155. Springer, 2005.
KOS16. M. Keller, E. Orsini, and P. Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In *CCS 2016*, pages 830–842. ACM, 2016.
KPR18. M. Keller, V. Pastro, and D. Rotaru. Overdrive: Making SPDZ great again. In *EUROCRYPT 2018*, pages 158–189. Springer, 2018.
KS21. M. Keller and K. Sun. Secure Quantized Training for Deep Learning. *CoRR*, abs/2107.00501, 2021, 2107.00501.
KTM+21. R. Karl, J. Takeshita, A. Mohammed, A. Striegel, and T. Jung. Cryptonomial: A Framework for Private Time-Series Polynomial Calculations. In *SecureComm 2021*, pages 332–351. Springer, 2021.
LBBH98. Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

LJLA17. J. Liu, M. Juuti, Y. Lu, and N. Asokan. Oblivious Neural Network Predictions via MiniONN Transformations. In *CCS 2017*, pages 619–631. ACM, 2017.
LL22. H. Lin and T. Liu. Two-Round MPC Without Round Collapsing Revisited – Towards Efficient Malicious Protocols. In *CRYPTO*, pages 353–382. Springer, 2022.
LLW20. H. Lin, T. Liu, and H. Wee. Information-theoretic 2-round MPC without round collapsing: adaptive security, and more. In *TCC 2020*, pages 502–531. Springer, 2020.
LNS21. V. Lyubashevsky, N. K. Nguyen, and G. Seiler. Shorter Lattice-Based Zero-Knowledge Proofs via One-Time Commitments. In *PKC 2021*, pages 215–241. Springer, 2021.
LYKM22. D. Lu, A. Yu, A. Kate, and H. K. Maji. Polymath: Low-Latency MPC via Secure Polynomial Evaluations and Its Applications. *PETS 2022*, (1):396–416, 2022.
MF06. P. Mohassel and M. K. Franklin. Efficient Polynomial Operations in the Shared-Coefficients Setting. In *PKC 2006*, pages 44–57. Springer, 2006.
MZ17. P. Mohassel and Y. Zhang. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *SP 2017*, pages 19–38. IEEE Computer Society, 2017.
NLV11. M. Naehrig, K. Lauter, and V. Vaikuntanathan. Can homomorphic encryption be practical? In *CCSW 2011*, pages 113–124. ACM, 2011.
NO07. T. Nishide and K. Ohta. Multiparty Computation for Interval, Equality, and Comparison Without Bit-Decomposition Protocol. In *PKC 2007*, pages 343–360. Springer, 2007.
NP99. M. Naor and B. Pinkas. Oblivious Transfer and Polynomial Evaluation. In *STOC 1999*, pages 245–254. ACM, 1999.
ON20. S. Ohata and K. Nuida. Communication-Efficient (Client-Aided) Secure Two-Party Protocols and Its Application. In *FC 2020*, pages 369–385. Springer, 2020.
Ore22. Ø. Ore. Über höhere kongruenzen. *Norsk Mat. Forenings Skrifter*, 1(7):15, 1922.
PSSY21. A. Patra, T. Schneider, A. Suresh, and H. Yalame. ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation. In *USENIX Security 2021*, pages 2165–2182. USENIX Association, 2021.
Rei09. T. I. Reistad. Multiparty Comparison - An Improved Multiparty Protocol for Comparison of Secret-shared Values. In *SECRYPT*, pages 325–330. INSTICC Press, 2009.
Rei+22. P. Reisert, M. Rivinius, T. Krips, S. Hasler, M. Rivinius, and R. Küsters. Arithmetic Tuples for MPC. Crypt. ePrint 2022/667, 2022.
Rei+23. P. Reisert, M. Rivinius, T. Krips, and R. Küsters. Overdrive LowGear 2.0: Reduced-Bandwidth MPC without Sacrifice. In *ACM ASIA CCS 2023*, 2023.
RRHK23. M. Rivinius, P. Reisert, S. Hasler, and R. Küsters. Convolutions in Overdrive: Maliciously Secure Convolutions for MPC. In *PETS 2023*, 2023.
RWT+18. M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar. Chameleon: A Hybrid Secure Computation Framework for Machine Learning Applications. In *AsiaCCS 2018*, pages 707–721. ACM, 2018.
SA19. S. Sahraei and A. S. Avestimehr. INTERPOL: Information Theoretically Verifiable Polynomial Evaluation. In *ISIT 2019*, pages 1112–1116. IEEE, 2019.
Sil21. T. Silde. Verifiable decryption for bgv. Crypt. ePrint 2021/1693, 2021.
TJB13. T. Tassa, A. Jarrous, and Y. Ben-Ya'akov. Oblivious evaluation of multivariate polynomials. *J. Math. Cryptol.*, 7(1):1–29, 2013.
Vai21. V. Vaikuntanathan. Secure Computation and PPML: Progress and Challenges. PPML 3rd Privacy-Preserving Machine Learning Workshop 2021, 2021.
WGC19. S. Wagh, D. Gupta, and N. Chandran. SecureNN: 3-Party Secure Computation for Neural Network Training. *Proc. Priv. Enhancing Technol.*, 2019(3):26–49, 2019.

A Technical Proofs for Theoretical Bandwidth and Tuple Size Computations

In this appendix we present the proofs to results from Section 4. We will also include tuple-size-optimized polytuples for the product of $m \leq 2$ elements in Table 2. We will first recall the statements of Section 4 and then prove them.

Corollary 1. *Let f, g be two functions. Let $\hat{f}$ be a randomized encoding of f with additive component $\hat{f}_0$ and simulator Sim_f. Furthermore, let $\hat{g}$ be a randomized encoding of g with additive component $\hat{g}_0$ and simulator Sim_g. Assume that for all $i, j > 0$ $(\mathsf{Sim}_f)_i$ and $(\mathsf{Sim}_g)_j$ are independent uniformly random numbers. Let $J = \{j > 0 | \exists i > 0 : \hat{f}_i = \hat{g}_j\}$. Then $((\hat{f}_i)_{0 \leq i}, (\hat{g}_j)_{j \notin J})$ is a*

randomized encoding of $(\hat{f},\hat{g})$ *with output size* $k+k'-|J|$*. Moreover, if* $\hat{f}_0,\hat{g}_0$ *map to the same (additive) group then* $((\hat{f}_i)_{0<i},(\hat{g}_j)_{j\notin J\cup\{0\}},\hat{f}_0+\hat{g}_0)$ *is a randomized encoding of* $f+g$ *with output size* $k+k'-|J|-1$ *and additive component* $\hat{f}_0+\hat{g}_0$*.*

Proof. Correctness follows trivially since one can still construct both $\hat{f}$ and $\hat{g}$. For privacy simulate all component apart from $\hat{f}_0$ and $\hat{g}_0$ by independent uniformly random numbers r_i, r'_j, i.e. $r_i=(\mathsf{Sim}_f)_i, r'_j=(\mathsf{Sim}_g)_j$ for $i>0$ and $j\notin J\cup\{0\}$. For simplicity set $r'_j \coloneqq r_i$ whenever $f_i=g_j$, i.e. $j\in J$. Then simulate f_0 by $f-\mathsf{Rec}_f(0,(r_i)_{1\le i})$ and g_0 by $g-\mathsf{Rec}_g(0,(r'_j)_{1\le j})$ where Rec_f is the reconstruction of $\hat{f}$ and Rec_g is the reconstruction of $\hat{g}$. For the sum we proceed analogously but set the final component to $f+g-\mathsf{Rec}_f(0,(r_i)_{1\le i})-\mathsf{Rec}_g(0,(r'_j)_{1\le j})$. □

We will next prove our main theorem Theorem 1 from Section 4:

Theorem 1. *Let* $\lambda,b,S_{k,j}$ *be defined as before. A product of* $m=\lambda b^n$ *shared inputs can be constructed with a polytuple of size* $\mathcal{O}\left(2^\lambda\left(\frac{b^2+1}{2}\right)^n\right)$ *with bandwidth* $\mathcal{O}\left(\left(\frac{b^2+1}{2}\right)^n\right)$*. In the special case* $b=2$*, one only needs a tuple of size* $2^{n-2}((2^\lambda-1)n^2+(2^{\lambda+2}-2^\lambda+1)n+4(2^\lambda-2))+1$*. For* $b=2$*, the bandwidth becomes* $2^n n+1+m$*.*

Proof. We will only need the case $N^1_{S_n}=N^2_{S_n}=2N^{1/2}_{S_n}$ to treat bandwidth and tuples size simultaneously. Hence, we slightly restrict our setup to this case from now on. In fact using binomial tuples we can construct the base encodings with tuple sizes $T^0_{S_{n,j}}=2^\lambda-1, T^1_{S_{n,j}}=T^2_{S_{n,j}}=2^\lambda$. For the output size we trivially have $N^0_{S_{n,j}}=N^1_{S_{n,j}}=N^2_{S_{n,j}}=1$. Now choose $\mu=bj+b-1$ to get $T_\mu=\{(\iota,\kappa)\in\{bj,\dots,bj+b-1\}^2:\kappa\le\iota\}$ and $\nu=bj+\lfloor\frac{b-1}{2}\rfloor$ to get $T_\mu\cap T_\nu=\{(\iota,\kappa)\in\{bj,\dots,bj+b-1\}^2:(\kappa\le\iota\le\nu)\vee(\nu+1\le\iota,\kappa\le\iota-\nu-1)\}$. In particular, $|T_\mu\cap M_{bj+i}|=i+1$ and $|T_\mu\cap T_\nu\cap M_{bj+i}|=i+1$ for $0\le i\le\nu-bj$, $|T_\mu\cap T_\nu\cap M_{i+\nu+1}|=i+1$ for $0\le i<\mu-\nu$. Hence we have

$$N^0_{S_k}=bN^0_{S_{k+1}}+(b-1)(bN^1_{S_{k+1}}-1) \tag{5}$$

$$N^1_{S_k}=\frac{(b-1)b+2}{2}N^1_{S_{k+1}}+(b-1)(N^2_{S_{k+1}}-1) \tag{6}$$

$$N^2_{S_k}=uN^1_{S_{k+1}}+b(N^2_{S_{k+1}}-1)+1 \tag{7}$$

with $u=\frac{(b-1)^2-1}{4}$ for b even and $u=\frac{(b-1)^2}{4}$ for b odd. Note that the we could remove unnecessary indices due to the symmetry of the $S_k\coloneqq S_{k,j}$. In order to get the required upper bound it will be enough to consider the odd case, which obviously leads to higher numbers. For the case $u=\frac{(b-1)^2}{4}$ we have

$$b(N^2_{S_{n-k}}-1)=((b+1)N^{1/2}_{S_n}-1)\left(\frac{b^2+1}{2}\right)^k+\frac{(b^2-1)(N^{1/2}_{S_n}-1)}{2}\left(\frac{b+1}{2}\right)^{k-1} \tag{8}$$

$$bN^1_{S_{n-k}}=2\left((b+1)N^{1/2}_{S_n}-1\right)\left(\frac{b^2+1}{2}\right)^k-2(N^{1/2}_{S_n}-1)\left(\frac{b+1}{2}\right)^k \tag{9}$$

$$(b-1)(N^0_{S_k}-1)=4b(N^2_{S_k}-1)+((b-1)(N^0_{S_n}-1)-4b(2N^{1/2}_{S_n}-1))b^{n-k} \tag{10}$$

Note that it is enough to proof these formulas for $N^0_{S_n} = T^0_{S_n} = 2^\lambda - 1, N^1_{S_n} = T^1_{S_n} = N^2_{S_n} = T^2_{S_n} = 2^\lambda$ to get the estimates on the tuple size, and for $N^0_{S_n} = N^1_{S_n} = N^2_{S_n} = 1$ for the bandwidth estimate. Recall that the bandwidth is just $N^0_{S_0} + m$, which accounts for the outputs of the randomized encoding as well as the bandwidth of the first round of interaction, i.e. the m terms $x_i - a_i$. This does not affect the asymptotic behavior, but the special formula in the case $b = 2$ discussed below. We will prove the explicit formula (8), (9), (10) by induction on $k \to k-1$ with start $k = n$:[20] $b(N^2_{S_n} - 1) = (b+1)N^{1/2}_{S_n} - 1 + (b-1)(N^{1/2}_{S_n} - 1)$, $bN^1_{S_n} = 2((b+1)N^{1/2}_{S_n} - 1) - 2(N^{1/2}_{S_n} - 1)$ and $(b-1)(N^0_{S_n}) = 4b(N^2_{S_n} - 1) + (b-1)(N^0_{S_n} - 1) - 4b(2N^{1/2}_{S_n} - 1)$. Hence, we get

$$
\begin{aligned}
&b(N^2_{S_{k-1}} - 1)\\
&= u \cdot 2\left((b+1)N^{1/2}_{S_n} - 1\right)\left(\frac{b^2+1}{2}\right)^{n-k} - u \cdot 2(N^{1/2}_{S_n} - 1)\left(\frac{b+1}{2}\right)^{n-k}\\
&\quad + b\left(((b+1)N^{1/2}_{S_n} - 1)\left(\frac{b^2+1}{2}\right)^{n-k} + \frac{(b^2-1)(N^{1/2}_{S_n} - 1)}{2}\left(\frac{b+1}{2}\right)^{n-k-1}\right)\\
&= ((b+1)N^{1/2}_{S_n} - 1)\left(\frac{b^2+1}{2}\right)^{n-k+1} + \frac{(b^2-1)(N^{1/2}_{S_n} - 1)}{2}\left(\frac{b+1}{2}\right)^{n-k}
\end{aligned}
$$

where we used $2u + b = \frac{b^2-2b+1+2b}{2} = \frac{b^2+1}{2}$ and $b(b^2-1)N^{1/2}_{S_n} - 1) - 2(N^{1/2}_{S_n} - 1)u(b+1) = \frac{b+1}{2}(N^{1/2}_{S_n} - 1)(2b(b-1) - (b-1)^2) = (b^2-1)(N^{1/2}_{S_n} - 1)\frac{b+1}{2}$. Analogously

$$
\begin{aligned}
&b(N^1_{S_{k-1}})\\
&= \frac{(b-1)b+2}{2}\left(2\left((b+1)N^{1/2}_{S_n} - 1\right)\left(\frac{b^2+1}{2}\right)^{n-k} - 2(N^{1/2}_{S_n} - 1)\left(\frac{b+1}{2}\right)^{n-k}\right)\\
&\quad + \frac{b-1}{2}\left(2((b+1)N^{1/2}_{S_n} - 1)\left(\frac{b^2+1}{2}\right)^{n-k} + (b^2-1)(N^{1/2}_{S_n} - 1)\left(\frac{b+1}{2}\right)^{n-k-1}\right)
\end{aligned}
$$

where we used $(b-1)b + b + 1 = b^2 + 1$ and $(b-1)b + 2 - \frac{2(b-1)(b^2-1)}{2(b+1)} = b(b-1) - (b-1)^2 + 2 = b+1$. Finally,

$$
\begin{aligned}
&(b-1)(N^0_{S_{k-1}} - 1)\\
&= b(b-1)(N^0_{S_k} - 1) + b(b-1)^2 N^1_{S_k}\\
&= 4b^2(N^2_{S_k} - 1) + 4ubN^1_{S_k} + ((b-1)(N^0_{S_n} - 1) - 4b(2N^{1/2}_{S_n} - 1))b^{n-k+1}\\
&= 4b(N^2_{S_{k-1}} - 1) + ((b-1)(N^0_{S_n} - 1) - 4b(2N^{1/2}_{S_n} - 1))b^{k+1}
\end{aligned}
$$

where we used the explicit formula for $N^2_{S_k}$ which was already proved before. This completes the proof of the first part of the statement.

The second part concerns the case $b = 2$. In particular, we have $u = 0$ in (7) and $N^1_{S_k}$ and $N^1_{S_k}$ decouple partly. Thus we get $N^2_{S_k} = (2N^{1/2}_{S_n} - 1) \cdot 2^{n-k} + 1$. Next, $N^1_{S_{k-1}} = 2^{n-k}((2N^{1/2}_{S_n} - 1)(n-k+1) + 4N^{1/2}_{S_n}) = 2 \cdot 2^{n-k-1}((2N^{1/2}_{S_n} - 1)(n-k) + 4N^{1/2}_{S_n}) + ((2N^{1/2}_{S_n} - 1) \cdot 2^{n-k} + 1) - 1 = 2N^1_{S_k} + N^2_{S_k} - 1$

[20] We will keep the N^*_* notation for the rest of the proof. For the tuple size substitute the corresponding T^*_*.

follows inductively from induction start $N^1_{S_n} = 2^{-1}((2N^{1/2}_{S_n} - 1)(-1 + 1) + 4N^{1/2}_{S_n}) = 2N^{1/2}_{S_n}$. Altogether we get the tuple size

$$\begin{aligned}
N^0_{S_{k-1}} &= 2^{n-k-1}((2N^{1/2}_{S_n} - 1)(n-k+1)^2 + (6N^{1/2}_{S_n} + 1)(n-k+1) \\
&\quad + 4(N^0_{S_n} - 1)) + 1 \\
&= 2^{n-k-1}((2N^{1/2}_{S_n} - 1)((n-k)^2 + 2(n-k)) + (6N^{1/2}_{S_n} + 1)(n-k) \\
&\quad + 8N^{1/2}_{S_n} + 4(N^0_{S_n} - 1)) + 1 \\
&= 2 \cdot (2^{n-k-2}((2N^{1/2}_{S_n} - 1)(n-k)^2 + (6N^{1/2}_{S_n} + 1)(n-k) \\
&\quad + 4(N^0_{S_n} - 1)) + 1) + 2 \cdot 2^{n-k-1}((2N^{1/2}_{S_n} - 1)(n-k) + 4N^{1/2}_{S_n}) - 1 \\
&= 2N^0_{S_k} + 2N^1_{S_k} - 1
\end{aligned}$$

where we started our induction with $N^0_{S_n} = \frac{1}{4} \cdot 4(N^0_{S_n} - 1) + 1$. □

Remark 14. If we add $|I_{j,k}|$ to Eqs. (2) to (4), e.g. to compute the number of Beaver triples needed in the offline phase, the formulas change slightly. We then get for $b = 2$: $N^0_{k-1} = 2N^0_k + 2N^1_k + 1, N^1_{k-1} = 2N^1_k + N^1_k + 1, N^2_{k-1} = 2N^2_k + 1$. One easily sees that then $N^2_{S_k} = (2N^{1/2}_{S_n} + 1) \cdot 2^{n-k} - 1$, $N^1_{S_{k-1}} = 2^{n-k}((2N^{1/2}_{S_n} + 1)(n-k+1) + 4N^{1/2}_{S_n})$. For $N^0_{S_{k-1}}$ we get

$$\begin{aligned}
N^0_{S_{k-1}} &= 2^{n-k-1}((2N^{1/2}_{S_n} + 1)(n-k+1)^2 + (6N^{1/2}_{S_n} - 1)(n-k+1) \\
&\quad + 4(N^0_{S_n} + 1)) - 1 \\
&= 2^{n-k-1}((2N^{1/2}_{S_n} + 1)((n-k)^2 + 2(n-k)) + (6N^{1/2}_{S_n} - 1)(n-k) \\
&\quad + 8N^{1/2}_{S_n} + 4(N^0_{S_n} + 1)) - 1 \\
&= 2 \cdot (2^{n-k-2}((2N^{1/2}_{S_n} + 1)(n-k)^2 + (6N^{1/2}_{S_n} - 1)(n-k) \\
&\quad + 4(N^0_{S_n} + 1)) - 1) + 2 \cdot 2^{n-k-1}((2N^{1/2}_{S_n} + 1)(n-k) + 4N^{1/2}_{S_n}) + 1 \\
&= 2N^0_{S_k} + 2N^1_{S_k} + 1
\end{aligned}$$

where we started our induction this time with $N^0_{S_n} = \frac{1}{4} \cdot 4(N^0_{S_n} + 1) - 1$.

We see that the case $b = 2$ leads to a tuple size in $\mathcal{O}(m(\log m)^2)$ while in all other cases $b > 2$ the tuple size is not even in $\mathcal{O}(n^2)$. We did not show this last fact in the previous proof for even $b > 2$ explicitly. It follows however from the next lemma—again we use N^*_* for both the output and the tuple size:

Lemma 6. *Let $b > 2$ be even. Define*

$$\tilde{N}^2_{S_k} = \frac{1}{2}\left(\frac{b^2}{2}\right)^{n-k} + 1, \quad \tilde{N}^1_{S_k} = \left(\frac{b^2}{2}\right)^{n-k}, \quad \tilde{N}^0_{S_k} = \left(\frac{b^2}{2}\right)^{n-k}$$

Then $\tilde{N}^l_{S_k} \leq N^l_{S_k}$ for $l = 0, 1, 2$ and all $k \geq 0$.

Proof. For the even case we have $u = \frac{(b-1)^2-1}{2}$. Also note, that the cases $k = 0$ are trivial. The statement is by definition correct for $k = n$. Inductively we get

$$\tilde{N}^2_{S_{k-1}} = \frac{1}{2}\left(\frac{b^2}{2}\right)^{n-k+1} + 1 = \frac{b(b-2) + 2b}{4}\left(\frac{b^2}{2}\right)^{n-k} + 1 = u\left(\frac{b^2}{2}\right)^{n-k} + \frac{b}{2}\left(\frac{b^2}{2}\right)^{n-k} + 1$$

$$= u\tilde{N}^1_{S_k} + b(\tilde{N}^2_{S_k} - 1) + 1 \leq uN^1_{S_k} + b(N^2_{S_k} - 1) + 1 = N^2_{S_{k-1}}$$

For $\tilde{N}^1_{S_{k-1}}$ we proceed similarly:

$$\begin{aligned}\tilde{N}^1_{S_{k-1}} = \left(\frac{b^2}{2}\right)^{n-k+1} &\leq \frac{(b-1)b+2}{2}\left(\frac{b^2}{2}\right)^{n-k} + \frac{b-1}{2}\left(\frac{b^2}{2}\right)^{n-k} \\ &= \frac{(b-1)b+2}{2}\tilde{N}^1_{S_k} + (b-1)(\tilde{N}^2_{S_k} - 1) \\ &\leq \frac{(b-1)b+2}{2}N^1_{S_k} + (b-1)(N^2_{S_k} - 1) = bN^1_{S_{k-1}}\end{aligned}$$

Finally,

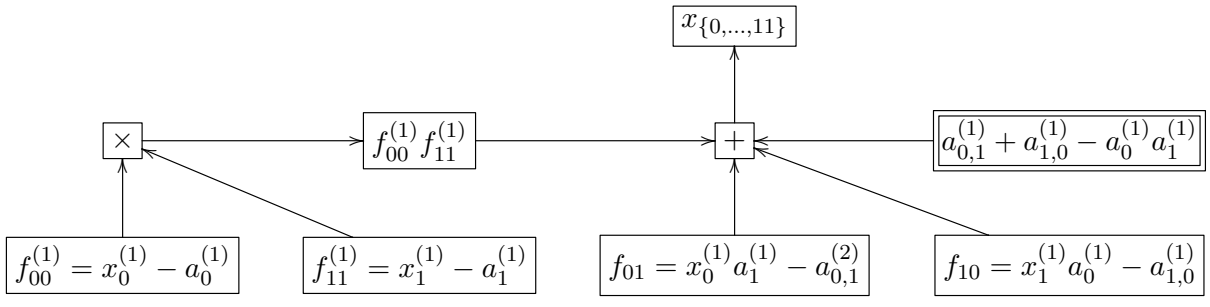
$$\begin{aligned}\tilde{N}^0_{S_{k-1}} = \left(\frac{b^2}{2}\right)^{n-k+1} &\leq b\left(\frac{b^2}{2}\right)^{n-k} + (b^2-b)\left(\frac{b^2}{2}\right)^{n-k} - b + 1 \\ &\leq b\tilde{N}^0_{S_k} + (b-1)(b\tilde{N}^1_{S_k} - 1) \leq bN^0_{S_k} + (b-1)(bN^1_{S_k} - 1) = N^0_{S_{k-1}}\end{aligned}$$

since $b - 1 \leq \left(\frac{b^2}{2}\right)^{k+1}$ for $k \geq 0, b \geq 4$. □

Example 1. This example discusses different trade-off of round complexity, bandwidth and tuple size in the special case of a product of $m = 12$ variables. There if we impose no restrictions on the offline phase, e.g. if a trusted third party provides the offline data for the online phase, then we can choose a binomial tuple (i.e. $\ell = 0$ in our randomized encodings) which has the small bandwidth of 13 ring elements and round complexity 1(+1), but a tuple size of 4095. For time-critical offline phases the classical Beaver multiplication approach needs a comparably small tuple size of $3(m-1) = 33$ but needs $\lceil \log m \rceil (+1) = 4(+1)$ rounds of communication and a bandwidth of $2m - 1 = 23$ ring elements. Polytuples with only 2-factor multiplication gates (cf. Table 2) provide an intermediate solution of tuple size 95, bandwidth 29 and 1(+1) round of communication.[21] If we accept slightly more communication rounds, say 2, then a combination of 4 binomial tuples for degree 3 and one polytuple for degree 4 in the second round will only need bandwidth $4 \cdot 4 + 3 = 19$ and tuple size $4 \cdot 7 + 13 = 41$. If we replace the second round with two rounds of classical Beaver multiplication we still have bandwidth $4 \cdot 4 + 3 = 19$ but tuple size $4 \cdot 7 + 9 = 37$.

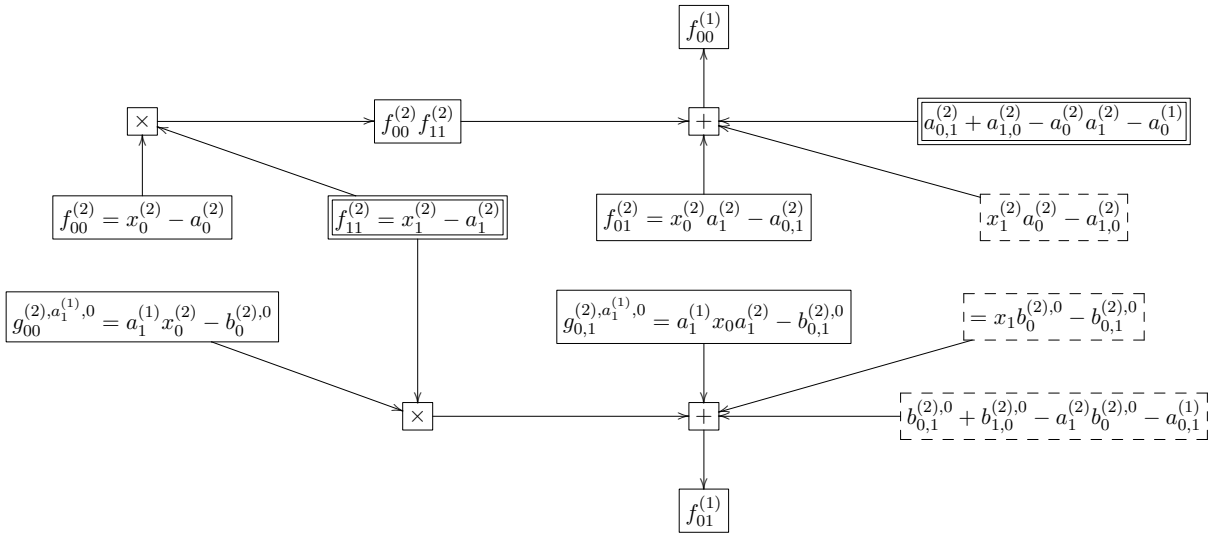
Example 2. In this example we visualize Example 1 by diagrams. We will use double frames for base encodings. To account for the sums in the additive parts f_{add} we will only mark one summand of this encoding with the double frame and the other parts with a dashed frame to make counting easier.

Take $S_{0,0} = \{0, \ldots, 11\}$, $S_{1,0} = \{0, \ldots, 5\}, S_{1,1} = \{6, \ldots, 11\}$ and set $x_0^{(1)} = x_{S_{1,0}}, x_1^{(1)} = x_{S_{1,1}}$



[21] Please see Example 2 for a visualization of the polytuple construction for $m = 12$.

Next set $S_{2,0} = \{0,\dots,3\}, S_{2,1} = \{4,5\}, S_{2,2} = \{6,\dots,9\}, S_{2,3} = \{10,11\}$. Consider first $x_0^{(2)} = x_{S_{2,0}}, x_1^{(2)} = x_{S_{2,1}}$ and



Of course we get the analogous decomposition for $f_{11}^{(1)}$ and $f_{10}^{(1)}$ if we set $x_0^{(2)} = x_{S_{2,2}}, x_1^{(2)} = x_{S_{2,3}}$. Finally consider $S_{3,0} = \{0,1\}, S_{3,1} = \{2,3\}, S_{3,2} = \{4,5\} = S_{2,1}, S_{3,3} = \{6,7\}, S_{3,4} = \{8,9\}, S_{3,5} = \{10,11\} = S_{2,3}$. One first notes we do not further decompose $f_{11}^{(2)}, f_{10}^{(2)}, g_{1,0}^{(2),*,0}$ for both choices $S_{3,2} = S_{2,1}, S_{3,5} = S_{2,3}$. Again by symmetry it will be enough to consider $x_0^{(3)} = x_{S_{3,0}}, x_1^{(3)} = x_{S_{3,1}}$.

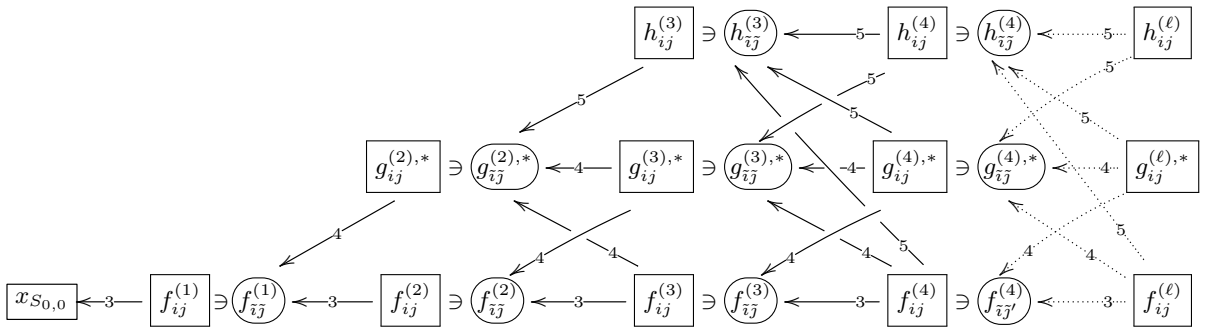
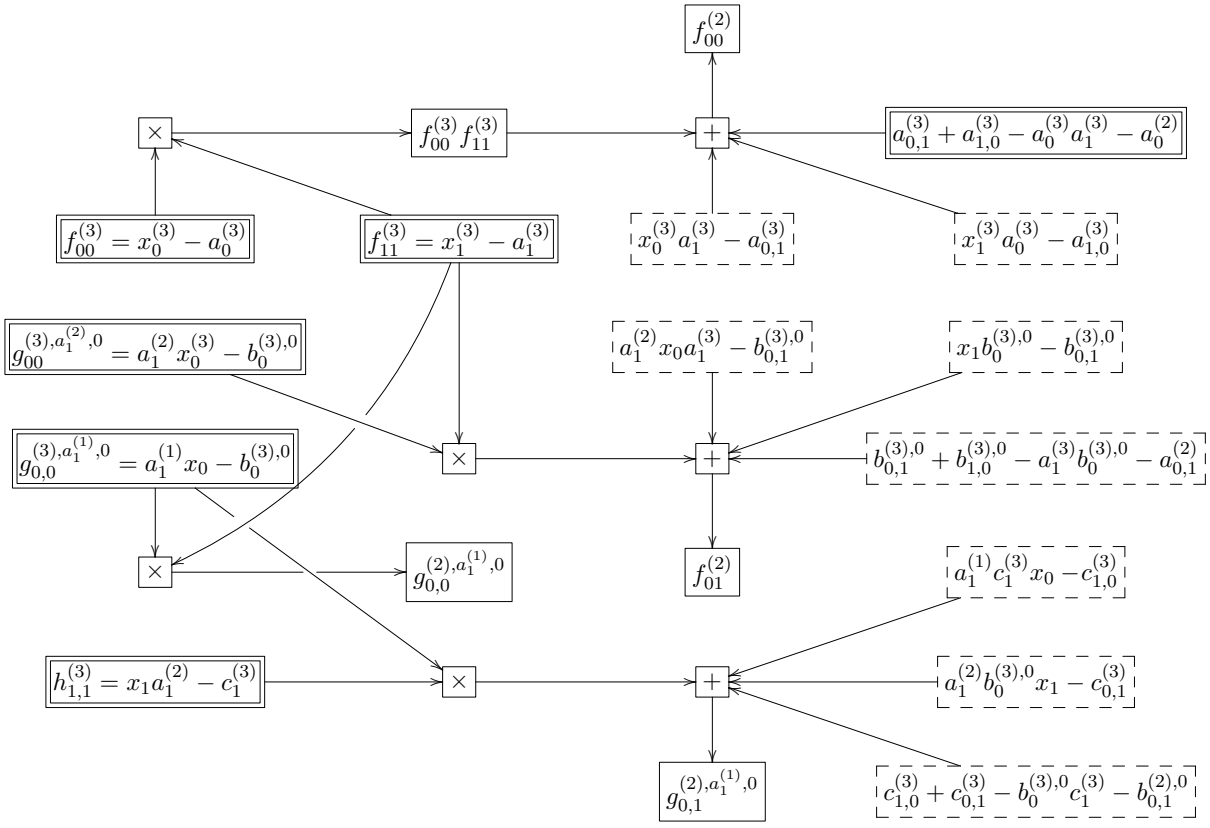


Fig. 8: The diagram illustrates which types of lower-degree randomized polynomials are used to build higher degree terms. E.g. $g_{ij}^{(k),*}$ denotes a type (ii) term linear term in $x_{S_{k,i}}$ which arose after k iterations—a $*$ substitutes for different choice of prefactor and μ, ν. Boxes, e.g. around $f_{ij}^{(k)}$, denote sets over $i, j \in I_{k+1,\tilde{\imath}}$ with $\tilde{\imath}$ the specific index one level higher, e.g. the index of $f_{\tilde{i}\tilde{j}}^{(k+1)}$; round framings denote single elements. The labels of the arrows refer to the numbers of the Lemmas 3 to 5. Note that in some special cases, e.g. $f_{ii}^{(k)}$, not all arrows are necessary, e.g. only Lemma 3. Furthermore, the $g_{ij}^{(k),*}$ already contain some $f_{ij}^{(k)}$ which will not be constructed multiple times. Analogously for $h_{ij}^{(k)}$. We omitted additive terms for clarity.

Let $\hat{F}_\kappa^{(k)} = \hat{F}_{\mathsf{add}}^{(k)} = \emptyset$ be sets of functions for all $0 \le k \le l, 0 \le \kappa < 3$. Set $\hat{F}_0^{(0)} \leftarrow \hat{F}_0^{(0)} \cup \{(x_{0,\dots,m-1}, 0)\}$.
for $0 \le k < l$ **do**
  for $0 \le \kappa < 3$ **do**
    for $(f, j) \in F_\kappa^{(k)}$ **do**
      if f has κ random prefactors as a polynomial in $x_{S_{k,j}}$ **then**
        Apply Lemma $\kappa + 2$ with $r \leftarrow I_{k,j}, x_i \leftarrow x_{S_{k+1,\psi_{kj}(i)}}$ for $i \in \mathbb{Z}_r$
        Receive $\hat{f} = (\hat{f}_{i\chi}, \hat{f}_{\mathsf{add}})$ for $i - 1 \neq \chi$ with terms linear in $x_{S_{k+1,\psi_{kj}(i)}}$.
      for $0 \le \kappa' < 3$ **do**
        if $\hat{f}_{ij}$ has κ' random prefactors as a polynomial in $x_{S_{k+1,\psi_{kj}(i)}}$ **then**
          $F_{\kappa'}^{(k+1)} \leftarrow F_{\kappa'}^{(k+1)} \cup \{(\hat{f}_{ij}, \psi_{kj}(i))\}$
      $F_{\mathsf{add}}^{(k+1)} \leftarrow F_{\mathsf{add}}^{(k+1)} \cup \{(\hat{f}_{\mathsf{add}}, k, j)\}$
  for $(f_{\mathsf{add}}, k', j) \in F_{\mathsf{add}}^{(k)}, f_{\mathsf{add}} = \sum_{i:S_{k,i} \subset S_{k',j}} g_i + c$ with g_i monomial in $x_{S_{k,i}}$ **do**
    for i such that $S_{k,i} \subset S_{k',j}$ **do**
      for $0 \le \kappa < 3$ **do**
        if g_i has κ random prefactors as a monomial in $x_{S_{k,i}}$ **then**
          Apply Lemma $\kappa + 2$ with $r \leftarrow I_{k,i}, x_\chi \leftarrow x_{S_{k+1,\psi_{ki}(\chi)}}$ for $\chi \in \mathbb{Z}_r$
          Receive $\hat{g}_i = (\hat{g}_{i,\chi\xi}, \hat{g}_{i,\mathsf{add}})$ for $\chi - 1 \neq \xi$ with terms linear in $x_{S_{k+1,\psi_{ki}(\chi)}}$.
      for $0 \le \kappa' < 3$ **do**
        if $\hat{g}_{i,\chi\xi}$ has κ' random prefactors as a monomial in $x_{S_{k+1,\psi_{ki}(\chi)}}$ **then**
          $F_{\kappa'}^{(k+1)} \leftarrow F_{\kappa'}^{(k+1)} \cup \{(\hat{g}_{i,\chi\xi}, \psi_{ki}(\chi))\}$
    $F_{\mathsf{add}}^{(k+1)} \leftarrow F_{\mathsf{add}}^{(k+1)} \cup \{(\sum_{i:S_{k,i} \subset S_{k',j}} \hat{g}_{i,\mathsf{add}} + c, k', j)\}$
return $F_{\mathsf{add}}^{(l)} \cup F^{(l)}$

Protocol 2: Algorithm to construct our randomized encoding.



Observe that we got $1 + 2 \cdot 2 + 2 \cdot 6$ base encodings and hence with the initial 12 masked inputs $x_i - a_i$ we get as expected bandwidth 29. We leave it to the reader to check the tuple size 95. See also Table 2 below.

The following example shows that using one encoding to construct different terms generally does not lead to a secure randomized encoding of the concatenation.

Example 3. Take $f(x_1, x_2) = x_1 + x_2, g(x_1, x_3) = x_1 + x_3$. Let $\hat{f} = (x_1 - a_1, x_2 - a_2, a_1 + a_2)$ and $\mathsf{Rec}_f(y_0, y_1, y_2) = y_0 + y_1 + y_2 = x_1 + x_2$ be a randomized encoding of f. Let $\hat{g} = (x_1 - a_1, x_3 - a_3, a_1 + a_3)$ and $\mathsf{Rec}_g = \mathsf{Rec}_f$ be a randomized encoding of g. We have the following simulators $\mathsf{Sim}_f = (a, b, f(x_1, x_2) - a - b)$ and $\mathsf{Sim}_g = (g(x_1, x_3) - a' - b', a', b')$ for randomness a, b, a', b' chosen by the simulator. Now $(\mathsf{Sim}_{f,1}, \mathsf{Sim}_{f,2}, \mathsf{Sim}_{g,2}, \mathsf{Sim}_{f,3}, \mathsf{Sim}_{g,3}) = (a, b, a', f(x_1, x_2) - a - b, b')$ is not a simulator for $(\hat{f}_1, \hat{f}_2, \hat{g}_2, \hat{f}_3, \hat{g}_3)$ of (f, g) since it contains no information on the actual result $g(x_1, x_3)$ but $\hat{f}_1 + \hat{g}_2 + \hat{g}_3 = g(x_1, x_3)$.

The following example describe our MPC protocol in the special case $m = 4$:

Example 4. Consider $f(x_0, x_1, x_2, x_3) = x_0 \cdot x_1 \cdot x_2 \cdot x_3$. Here, each party receives a structured polytuple of size 13:

$$\begin{aligned}[\![a]\!] \coloneqq ([\![a_0]\!],[\![a_1]\!],[\![a_2]\!],[\![a_3]\!],[\![a_0a_1]\!],[\![a_{01}]\!],[\![a_{01}a_2]\!],[\![a_{01}a_3]\!],[\![a_2a_3]\!],\\ [\![a_{23}]\!],[\![a_{23}a_0]\!],[\![a_{23}a_1]\!],[\![a_{01}a_2a_3 + a_{23}a_0a_1 - a_{01}a_{23}]\!])\end{aligned} \tag{11}$$

from the offline phase. The parties then proceed accorong to Section 4.1. In this special case this reduces to Protocol 3. The randomized encoding is then $\hat{f}(x_0, x_1, x_2, x_3, a) = (y_0, y_1, y_2)$

1. P_i computes and opens $[\![x_j]\!]_i - [\![a_j]\!]_i$ for all $0 \le j < 4$.
2. P_i computes locally and opens
(i) $[\![y_1]\!]_i = [\![x_0x_1 - a_{01}]\!]_i = (x_0 - a_0)[\![x_1]\!]_i + [\![a_0]\!]_i(x_1 - a_1) + [\![a_0a_1]\!]_i - [\![a_{01}]\!]_i$
(ii) $[\![y_2]\!]_i = [\![x_2x_3 - a_{23}]\!]_i = (x_2 - a_2)[\![x_3]\!]_i + [\![a_2]\!]_i(x_3 - a_3) + [\![a_2a_3]\!]_i - [\![a_{23}]\!]_i$
(iii) $[\![y_0]\!]_i = [\![a_{23}x_0x_1 + a_{01}x_2x_3 - a_{01}a_{23}]\!]_i = (x_0 - a_0)(x_1 - a_1)[\![a_{23}]\!]_i + (x_0 - a_0)[\![a_1a_{23}]\!]_i + [\![a_0a_{23}]\!]_i(x_1 - a_1) + [\![a_{01}]\!]_i(x_2 - a_2)(x_3 - a_3) + (x_2 - a_2)[\![a_3a_{01}]\!]_i + [\![a_2a_{01}]\!]_i(x_3 - a_3) + [\![a_0a_1a_{23} + a_{01}a_2a_3 - a_{01}a_{23}]\!]_i$.
3. P_i computes the result $x_0x_1x_2x_3 = y_{01}y_{23} + y_{0123}$.

Protocol 3: Protocol to compute the product $x_0 \cdots x_3$.

and the reconstruction algorithm is $\mathsf{Rec}(y_0, y_1, y_2) = y_1y_2 + y_0$. This example corresponds to the classical arithmetic circuit for the multiplication of four variables, i.e. we first compute (in parallel) x_0x_1 and x_2x_3 and in the second step the product of all four variables. To stay secure however, we cannot open x_0x_1 or x_2x_3, so we mask these products with fresh randomness a_{01} and a_{23}, respectively. This leads to the unwanted mixed term $a_{23}x_0x_1 + a_{01}x_2x_3 - a_{01}a_{23}$ in the second level multiplication. We remove this mixed term with the final encoding y_0. The privacy of the randomized encoding follows from the general results in Section 4.

The following example compute the bandwidth and polytuple size for a different number of communication rounds:

Example 5. As an example we want to compute the product of $x_0, \ldots, x_{15}$ using the partitions by $S_{3-k,j} = \{2^{k+1} \cdot j + i : 0 \le i < 2^{k+1}\}$ and $0 \le j < 2^{3-k}$. In particular, we will get elementary

Table 2: Polytuples to compute $x_0 \cdots x_{m-1}$ optimized for tuple size. Numbers indicate the degree of the polynomial encodings (soley in x_j), brackets are evaluated from the inside, e.g. $((2,2),3)$ first generates a degree 4 term an then a degree 7 term.

m	Tuple Size	Bandwidth	Circuit
1	1	1	(1)
2	3	3	(1,1)
3	7	4	(1,1,1)
4	13	7	(2,2)
5	21	8	(3,2)
6	29	9	(3,3)
7	38	13	((2,2),3)
8	47	17	((2,2),(2,2))
9	59	18	((3,2),(2,2))
10	71	19	((3,2),(3,2))
11	83	24	(((2,2),2),(3,2))
12	95	29	(((2,2),2),((2,2),2))
13	108	34	(((2,2),2),((2,2),(2,1)))
14	121	39	(((2,2),(2,1)),((2,2),(2,1)))
15	135	40	(((2,2),(2,2)),((2,2),(2,1)))
16	149	41	(((2,2),(2,2)),((2,2),(2,2)))
17	165	42	(((3,2),(2,2)),((2,2),(2,2)))
18	180	48	((((2,2),2),(2,2)),((2,2),(2,2)))
19	196	49	((((2,2),2),(2,2)),((3,2),(2,2)))
20	211	55	((((2,2),2),(2,2)),(((2,2)),2),(2,2)))

encodings linear in $x_{\{2j,2j+1\}}, 0 \leq j < 8$. For the cases $y_{\{2j,2j+1\}}$ we need a $2^2 - 1 = 3$-tuple, for the one prefactor case $ax_{\{2j,2j+1\}} - b_{\{2j,2j+1\}}$ a 4-tuple. Thus we can construct the degree 4 terms $x_{S_{2,j}} - a_{S_{2,j}}$ by a tuple of size $T^0_{S_{2,j}} = 3 + 3 + (2-1)(4+4) - 2 + 1 = 13$. Moreover, for $\mu = 2j + 1$ we have $T_\mu = \{(\iota, \kappa) \in \{2j, 2j+1\}^2 : \kappa \leq \iota\}$ and $T^1_{S_{2,j}} = 4 \cdot 1 + 1 \cdot 4 + 4 - 2 + 1 = 11$. Thus $T^0_{S_{1,j}} = 13 + 13 + (2-1) \cdot 22 - 2 + 1 = 47$. Next, we compute again the mixed terms with $\mu = 2j + 1, \nu = 2j$ and hence $T_\mu \cap T_\nu = \{(\mu, \mu), (\nu, \nu)\}$: $T^2_{S_{2,j}} = T^2_{S_{3,j}} + T^2_{S_{3,j}} - 2 + 1 = 7$ and $T^1_{S_{1,j}} = 11 \cdot 1 + (2-1) \cdot 7 + 11 - 2 + 1 = 28$. Thus, $T^0_{S_{0,0}} = 47 + 47 + (2-1) \cdot 56 - 2 + 1 = 149$, i.e. we can construct a $x_{\{0,\ldots,15\}}$ in one masking round and one opening round with a 149-tuple. In comparision, a binomial tuple for the computation of $x_{\{0,\ldots,15\}}$ has size $2^{16} - 1$.

B Functionalities

In this section we present the ideal functionalities and security proofs. We assume that the parties have access to a functionality $\mathcal{F}_{\text{random}}$ to produce random elements from R and a commitment functionality $\mathcal{F}_{\text{commit}}$—possible realization can be found e.g. in [DKL+13].

Remark 15. To describe $\mathcal{F}_{\text{online}}$, we used a modification of the functionality $\mathcal{F}_{\text{AMPC}}$ from [DPSZ12]. One can also use $\mathcal{F}_{\text{Online}}$ from [DKL+13] or similar functionalities.

Remark 16. The functionality $\mathcal{F}_{[\![\cdot]\!]}$ can be realized as in [KPR18, Fig. 4].

Lemma 7. *The protocol* CheckMAC *is correct and sound. It rejects with probability* $1 - \frac{2}{|R|}$ *if at least one value is not computed correctly.*

Proof. Identical to the corresponding proof in [DPSZ12]. □

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

Initialize. On input $(\text{Initialize}, p)$ from all parties, the functionality stores p.

Input. On input $(\text{Input}, P_i, \text{id}_x, x)$ from P_i and $(\text{Input}, P_i, \text{id}_x)$ from all others, the functionality stores (id_x, x). id_x has to be a new identifier.

Polynomial. On input $(\text{Polynomial}, (\text{id}_{x_k})_{0 \le k < m}, f, \text{id}_z)$ for a polynomial f (with m inputs) from all parties with id_z new, the functionality retrieves $(\text{id}_{x_k}, x_k)_{1 \le k < m}$ and stores $(\text{id}_z, f(x_0, \ldots, x_{m-1}))$.

Output. On input $(\text{Output}, \text{id}_x)$ for id_x defined, from all honest parties, the functionality retrieves (id_x, x) and outputs it to the adversary. If the adversary replies by ok, then x is output to all players, otherwise output $\perp$ to all players.

Protocol 4: Ideal functionality for the online phase.

Π_{online}

Initialize. The parties agree on a multivariate polynomial f to be evaluated and a randomized encodings $\hat{f}$ of f. The parties call $\mathcal{F}_{[\![\cdot]\!]}$ to get a sufficient number of (shared) random data (including (poly)tuples, MAC key shares).
Input. On input a tuple X_i of inputs of party P_i, the parties invoke $\mathcal{F}_{[\![\cdot]\!]}.\text{Input}$. They receive $[\![X_i]\!]$. Denote x the tuple of all inputs (from all parties).
Polynomial. The parties invoke Protocol 1 with $(\hat{f}, [\![x]\!], [\![\hat{a}]\!], \texttt{continuation})$.
Check. Call Π_{CheckMAC} for all values opened up until now.

Protocol 5: Online Protocol.

C Tuple Production

There are various established methods for generating correlated randomness in the offline phase. The most prominent ones are the following: somewhat homomorphic encryption (SHE; SPDZ [DKL+13, DPSZ12] utilizes BGV [BGV12]), oblivious transfer (as used in MASCOT [KOS16]), or linear homomorphic encryption (LHE; as used in Overdrive [KPR18, Rei+23]). These mostly focus on generating Beaver triples. We present two ways to generate polytuples based on these following methods: one directly uses the generated Beaver triples for tuple production and the other generalizes the underlying techniques to generate higher order randomness. We focus on a LHE-based offline phase based on Overdrive for the latter and present a leveled homomorphic polytuple generation in Appendix D.3.

C.1 Plugin Approach

We can use the structured randomness, i.e. Beaver triples, generated by existing protocols to construct polytuples in an actively secure offline phase. Each entry of a polytuple is a sharing of some polynomial in random variables, i.e. the additive masks for the single encoding components. These polynomials can be computed with the SPDZ online phase. This means, we produce in our offline phase a sufficient number of Beaver triples to run the SPDZ online phase (still within our offline phase) to compute the tuple entries. This straightforward generation corresponds nicely with the idea to shift as much computation from the online phase into the offline phase. The advantage of using already existing offline phases is that efficient implementations like [Kel20] available and that further improvements of these offline phases will be directly available to the production of polytuples as well.

$\mathcal{F}_{[\![\cdot]\!]}$

Initialize. On input $(\text{Initialize}, p)$ from all parties, store p and compute $[\alpha]_i$ for honest P_i and receive $[\alpha]_j$ for corrupted P_j; then set $\alpha \coloneqq \sum_{i=1}^{n} [\alpha]_i$.

Input. On input $(\text{Input}, P_i, \text{id}_x, x)$ from P_i and $(\text{Input}, P_i, \text{id}_x)$ from all others, sample $[x]_i$ for honest P_i under the constraint $x = \sum_{i=1}^{n} [x]_i$ (for $[x]_j$ received by Adv for corrupted P_j) and authenticate the shares. Send $[\![x]\!]_i$ to the respective P_i.

Tuple. On input (Tuple, f) by all parties for a polynomial $f : R^m \to R$. Sample random masks $(a_1, \ldots, a_m)$ and compute the tuple $(a_1, \ldots, a_k)$ for the respective randomized encoding.[a] Authenticate the tuple and send $[\![(a_1, \ldots, a_k)]\!]_i$ to the respective P_i.

Abort. On input $\perp$ from Adv, send $\perp$ to all parties.

[a] $m \leq k$ to mask at least all inputs. To construct the randomized encoding one usually needs more entries $k > m$.

Protocol 6: Preprocessing functionality.

Π_{CheckMAC}

Every party P_i has $[\![y_j]\!]_i = ([y_j]_i, [\alpha y_j]_i [\alpha]_i)$, $1 \leq j \leq m$. $y = (y_1, \ldots, y_m) \in R^m$ is public and has to be checked.[a]

1. The parties sample a random $r \in R^m$.
2. Every party computes $[\sigma]_i = r^t([\alpha y]_i - [\alpha]_i y)$ for r^t the transpose of r.
3. Call $\mathcal{F}_{\text{commit}}$ with $(\text{Commit}, [\sigma]_i)$ and receive handle τ_i.
4. After each party has committed, call $\mathcal{F}_{\text{commit}}$ with (Open, τ_i) to open $[\sigma]_i$.
5. If $\sum_{i=1}^{n} [\sigma]_i \neq 0$ then abort.

[a] $[\alpha y]_i = ([\alpha y_1]_i, \ldots, [\alpha y_m]_i)$.

Protocol 7: CheckMAC

Remark 17. Please also note that a polytuple does not have to contain complex correlated randomness of high degree. In fact, as we have seen in Section 4, we often only need randomized encodings with up to two random prefactors linear in a monomial of some low degree d. E.g. for $d = 2$ to compute abx_1x_2 a polytuple contains $[\![a_1]\!], [\![a_2]\!], [\![ab]\!], [\![aba_1]\!], [\![aba_2]\!], [\![aba_1a_2]\!]$.

C.2 Linear Homomorphic Encryption

We now propose a new multi-round offline protocol for generating polytuples based on linear homomorphic encryption. We construct a protocol similar to Overdrive's multiplication protocol [KPR18] but which extends it to multiple rounds (to compute higher order randomness). In contrast to Appendix C.1, where Overdrive is one method to produce Beaver triples, one can also run several rounds of Overdrive to produce higher order randomness. E.g. after one round of Overdrive, which needs two rounds of communication, the parties have shares $[ab]$, $[cd]$ and after a second round of Overdrive they get shares of $[abcd]$ and so on. To produce a degree m term we then need $\lceil \log(m) \rceil$ rounds of Overdrive resulting in $2\lceil \log(m) \rceil$ rounds of communication—in each Overdrive round a party P_j first sends a ciphertext $\text{Enc}_{\text{pk}_j}([a]_j)$ to P_i and then receives back a term $\text{Enc}_{\text{pk}_j}([a]_j)\,[b]_i + \text{Enc}'_{\text{pk}_j}(r_{ji})$ which they decrypt to $[a_j]\,[b]_i + r_{ji}$. Here, Enc' has larger noise than Enc (cf. [KPR18] for further details).

Our adaption removes the second step. Instead of returning $\text{Enc}_{\text{pk}_j}([a]_j\,[b]_i + r_{ij})$ to P_j, this ciphertext is sent on to all parties that multiply their secrets onto the ciphertext. By the linear

property of the encryption scheme, the new factors again move into the ciphertext. When the ciphertext of the product arrives back at the initial party, they can decrypt the product. Thus far, this description mostly resembles the original Overdrive multiplication protocol. In our multi-round version, we additionally make the parties prove (in zero-knowledge) that the resulting ciphertext $\mathrm{Enc}_{\mathrm{pk}_j}([a]_j\,[b]_i + r_{ij})$ is still "fresh enough" (i.e. contains a low amount of noise) to be used in another round. An example of this approach is shown in Protocol 8, where the parties have to prove correct multiplication (and adding of "small" additional noise) which implies that the total noise in the ciphertext is small as well. Correctness and privacy of our construction follows similarly to Overdrive [KPR18]. The additional ZKP of correct multiplication (using $\mathcal{F}_{\text{ZK-mul}}$) guarantees privacy by giving parties provable upper-bounds on the noise contained in ciphertexts. Then, they can choose the randomness in Enc' large enough to hide any information about their own shares. Observe that after one round of $\Pi_{\text{LHE-rd}}$ every party P_i has their share $[c]_i$ of the product c and encryptions of all shares $\mathrm{Enc}_{\mathrm{pk}_j}([c]_j)$ for each $1 \le j \le n$, i.e. we can iterate protocol $\Pi_{\text{LHE-rd}}$, as seen in Protocol 9. In particular, each product of m shares can be computed in $\lceil \log(m) \rceil$ rounds.

$\Pi_{\text{LHE-rd}}$

Each party P_i holds $\mathrm{Enc}_{\mathrm{pk}_j}([a]_j)$ for each $1 \le j \le n$, $[b]_i$. Each P_i does:

1. For each $j \neq i$ sample r_{ij}. Set $r_{ii} \coloneqq -\sum_{j \neq i} r_{ij}$.
2. Broadcast $\hat{d}_{ji} \coloneqq \mathrm{Enc}_{\mathrm{pk}_j}([a]_j)\,[b]_i - \mathrm{Enc}'_{\mathrm{pk}_j}(r_{ij})$ for each $1 \le j \le n$ with $\mathcal{F}_{\text{ZK-mul}}$.
3. Decrypt $\hat{d}_{ij}$ to d_{ij} for all $1 \le j \le n$.
   Set $[c]_i = \sum_{j=1}^{n} d_{ij}$ and $\mathrm{Enc}_{\mathrm{pk}_j}([c]_j) = \sum_{k=1}^{n} \hat{d}_{jk}$ for all $1 \le j \le n$.

Protocol 8: Multiplication using an LHE scheme.

We have included several variations of our technique in Appendix D.2 that achieve provable upper bounds on the ciphertext noise but are based on ZKPs for different relations (e.g. ZKPs for verifiable decryption). With this, we can benefit from future improvements of various types of zero-knowledge proofs which are then also transferable to our approach. The tradeoff of our construction is a larger ciphertext size: Since noise adds up every round and is not canceled out by intermediate decryptions, the ciphertext size will grow more and more. However, as mentioned in Remark 17, we generally do not need to compute randomness of very high degree. Additionally, if the noise reaches a level that is too high to continue the computation on ciphertexts, the secret-key holder can decrypt and provide a new ciphertext with fresh small randomness at cost of one intermediate communication round. Also note that the reduction in round complexity, which was the main motivation for our adaption to Overdrive, suggests that our approach is best employed in settings with (moderately) high network latency and sufficient bandwidth to handle the larger ciphertexts.

Once the shares of the tuples are created, they are authenticated using $\mathcal{F}_{[\![\cdot]\!]}$. The parties then use the new extended sacrificing technique to check that the tuples are well formed. Details can be found in Appendix D.1.

Remark 18. Our MP-SPDZ implementation [Code] currently only covers the online phase. Based on the log-linear overhead in tuple size, the overhead in runtime for the offline phase will be in the same range (e.g. based on Beaver triples as discussed in Appendix C). As our focus in on

Π_{LHE}

Let f be a degree d polynomial in m variables $x_0, \cdots, x_{m-1}$. Each party P_i holds $[x_j]_i$ for each $0 \leq j < m$ and computes $[f(x_0, \ldots, x_{m-1})]_i$ with the following protocol:

1. Broadcast $\text{Enc}_{\text{pk}_i}([x_j]_i)$ for $0 \leq j < m$ and proof that it is well-formed with the zero-knowledge proof $\mathcal{F}^S_{\text{ZKPoP}}$ from [KPR18].
2. Compute $[f(x_0, \ldots, x_{m-1})]_i$ in $\lceil \log d \rceil$ rounds of $\Pi_{\text{LHE-rd}}$.

Protocol 9: Triple production with multi-round LHE.

applications where the offline phase is not time-critical, we leave benchmarking (and optimizing) the offline phase to future work.

D Further Results on the Offline Phase

This section contains results that can be used in our offline phase. It starts with a subsection on sacrificing for binomial and polytuples which can be applied to most of the currently implemented actively secure offline phases like the once in [DPSZ12] or [KPR18]. In Appendix D.2 we offer options on how to realize the ZKP functionality used in Appendix C.2. Finally we shortly discuss polytuple production with leveled homomorphic encryption.

D.1 Extended Sacrificing Technique

This appendix contains a new sacrificing technique for binomial and polytuples which can also be applied to most of the currently implemented actively secure offline phases like the once in [DPSZ12] or [KPR18].

To make sure that the binomial tuples are indeed in the right form, we need to extend the well-know sacrificing technique from [DPSZ12] in Protocol 10.[22]

$\Pi_{\text{sacrificing}}$

Let (r_e) and (r'_e) be two binomial tuples for $e = (e_0, \ldots, e_{m-1}), 0 \leq e_j \leq d_j, 0 \leq j < m$.

1. The parties use $\mathcal{F}_{\text{rand}}$ to get random $s_0, \ldots, s_{m-1}$.
2. The parties compute and open $t_j = s_j \left[r'_j\right]_i - [r_j]_i$ for $0 \leq j < m$.
3. The parties compute

$$[\text{sac}]_i = \sum_{\substack{0 \leq j < m \\ 0 \leq e_j \leq d_j}} \left(s^e \cdot \left[r^{e\prime}\right]_i - \sum_{\substack{0 \leq j < m \\ 0 \leq f_j \leq e_j}} \left[r^f\right]_i \prod_{k=0}^{m-1} t_k^{e_k - f_k} \right)$$

   with $s^e = \prod_{j=1}^m s_j^{e_j}$. The parties commit to and open $[\text{sac}]_i$ with $\mathcal{F}_{\text{commit}}$.
4. If $\sum_{i=1}^n [\text{sac}]_i = 0$ return (r'_e), otherwise abort.

Protocol 10: Verification for binomial tuples by sacrificing.

[22] We will use the index notation from Section 3.4.

Inspecting Protocol 10, we see that it is obviously correct. We use the simple identity

$$\sum_{\substack{0\le j<m\\ 0\le f_j\le e_j}} \left[r^f\right]_i \prod_{k=0}^{m-1} t_k^{e_k-f_k} = \prod_{j=0}^{m-1} (s_j r'_j)^{e_j} = \sum_{i=1}^{n} s^e \cdot \left[r^{e\prime}\right]_i$$

where the first equality holds if (r^e) is correct and the second one if $(r^{e\prime})$ is a correct binomial tuple. Then we get $\sum_{i=1}^{n} [\mathrm{sac}]_i = 0$. On the other hand, if $\sum_{i=1}^{n} [\mathrm{sac}]_i = 0$, s is a zero of a polynomial in m variables. If at least one party is honest, the coefficients of the polynomial are random (by the guarantees of $\mathcal{F}_{\mathrm{rand}}$). The Schwartz-Zippel Lemma for finite fields [Ore22] guarantees that the probability that a random s is a zero of a total degree $d \ge 0$ polynomial f is smaller than $\frac{d}{p}$: $\Pr(f(s_0,\ldots,s_{m-1}) = 0 \mid r_s \in \mathbb{F}_p) \le \frac{d}{p}$ for $f \neq 0$ and $d = \sum_{j=0}^{m-1} d_j$. For a sufficiently large field size, this probability is negligible, which shows the security of our sacrificing protocol.

Remark 19. Recall from Section 4 that elementary encodings in our protocol are constructed using binomial tuples — respectively tuples of the form (a, aa^e) or (ab, aba^e) for additional random tuple entries a, b plus some random additive terms. For all these binomial tuple within two polytuple of the same type for the same function, we can simply apply the sacrificing step for two binomial tuples from Protocol 10.

D.2 Further Results For A Linear Homomorphic Offline Phase

In this section, we present three approaches to efficiently construct a linear homomorphic offline phase. All these approaches require the parties to prove certain parts of their computation in zero-knowledge in each round.[23] Protocols 11, 13 and 15 depict the (exposition-only) functionalities used in Protocols 8, 12 and 14, respectively. The main reason for the ZKPs is that the parties need to know that the ciphertexts $\mathrm{Enc}_{\mathrm{pk}_j}([c]_j)$ can be used in the next round. For this, an upper bound on the noise contained in these ciphertexts has to be known. This enables maskings with r_{ij} in the next round to be chosen large enough so $\tilde{d}_{ji}$ does not leak information about $[b]_i$ to P_j.

The first approach (Protocol 12) requires parties to prove correct decryption in zero-knowledge. This protocol leaves the responsibility for proving that $\mathrm{Enc}_{\mathrm{pk}_j}([c]_j)$ has small noise with P_j. Verifiable decryption can be achieved efficiently with recent protocols [GHM+21,LNS21, Sil21].

$\mathcal{F}_{\text{ZK-dec}}$

On input $\mathrm{Enc}_{\mathrm{pk}_i}(a)$ by party P_i:

Send $(\mathrm{Enc}_{\mathrm{pk}_i}(a), P_i, \mathrm{ok})$ to all parties P_j if the noise in the ciphertext is small wrt. the bound B_{dec} (also send a to P_i). Otherwise, send $(\mathrm{Enc}_{\mathrm{pk}_i}(a), P_i, \mathrm{fail})$.

Protocol 11: Ideal functionality for verifiable decryption.

The second approach (Protocol 14) requires parties to prove correct decryption in zero-knowledge but it is done differently to the approach taken in Protocol 12. Instead of proving

[23] Results that are not used as the input to subsequent rounds do not require proofs. As Overdrive is a one-round protocol, it only needs ZKPs for the initial ciphertexts.

$\Pi_{\text{LHE-rd-dec}}$

Each party P_i holds $\text{Enc}_{\text{pk}_j}([a]_j)$ for each $1 \leq j \leq n$, $[b]_i$. Each P_i does:

1. For each $j \neq i$ sample r_{ij}. Set $r_{ii} \coloneqq -\sum_{j \neq i} r_{ij}$.
2. Broadcast $\hat{d}_{ji} \coloneqq \text{Enc}_{\text{pk}_j}([a]_j)\,[b]_i - \text{Enc}'_{\text{pk}_j}(r_{ij})$ for each $1 \leq j \leq n$.
3. Set $\text{Enc}_{\text{pk}_j}([c]_j) = \sum_{k=1}^{n} \hat{d}_{jk}$ for all $1 \leq j \leq n$.
   Decrypt $\text{Enc}_{\text{pk}_i}([c]_i)$ to $[c]_i$ with $\mathcal{F}_{\text{ZK-dec}}$ (and broadcast the proof).

Protocol 12: Multiplication using an LHE scheme with ZKP of decryption.

that they can decrypt $\text{Enc}_{\text{pk}_j}([c]_j)$ with small noise, P_j could instead prove knowledge of a small witness (plaintext and randomness) that encrypts to $\text{Enc}_{\text{pk}_j}([c]_j)$. For this, we would need a decryption algorithm that also recovers (some) randomness that matches the ciphertext. This is modelled in Protocol 13.

$\mathcal{F}_{\text{ZK-dec}}$

On input $\text{Enc}_{\text{pk}_i}(a)$ by party P_i:
Send $(a, \tilde{a})$ to P_i with $\text{Enc}_{\text{pk}_i}(a, \tilde{a}) = \text{Enc}_{\text{pk}_i}(a)$.

Protocol 13: Ideal functionality for decryption with extraction.

$\Pi_{\text{LHE-rd-ext}}$

Each party P_i holds $\text{Enc}_{\text{pk}_j}([a]_j)$ for each $1 \leq j \leq n$, $[b]_i$. Each P_i does:

1. For each $j \neq i$ sample r_{ij}. Set $r_{ii} \coloneqq -\sum_{j \neq i} r_{ij}$.
2. Broadcast $\hat{d}_{ji} \coloneqq \text{Enc}_{\text{pk}_j}([a]_j)\,[b]_i - \text{Enc}'_{\text{pk}_j}(r_{ij})$ for each $1 \leq j \leq n$.
3. Set $\text{Enc}_{\text{pk}_j}([c]_j) = \sum_{k=1}^{n} \hat{d}_{jk}$ for all $1 \leq j \leq n$.
   Decrypt $\text{Enc}_{\text{pk}_i}([c]_i)$ to $([c]_i, \rho)$ with $\mathcal{F}_{\text{dec-ext}}$.
   Use $\mathcal{F}_{\text{ZK}}$ to prove $\text{Enc}_{\text{pk}_i}([c]_i) = \text{Enc}([c]_i, \rho)$ (and broadcast the proof).

Protocol 14: Multiplication using an LHE scheme with randomness extraction.

The third and maybe most straighforward approach (used in Protocol 8) requires parties to prove correct multiplication in zero-knowledge. Examples for protocols that provide verifiable multiplication can, for example, be found in the BDOZ [BDOZ11] and below.

A Modified Zero-Knowledge Proof. In order for the protocol Π_{LHE} to be secure we have to realize the functionality $\mathcal{F}_{\text{ZK-mul}}$ used in Protocol 9. This can be done by slightly adapting existing zero-knowledge proofs, e.g. from [DPSZ12], [DKL+13], [KPR18] or [BCS20]. As an example we present suitable modifications to [DPSZ12], Fig. 9 with slight simplifications to the bounds similar to [KPR18], Fig. 10. Note that this interactive proof can be transformed into a non-interactive proof in the usually way using the Fiat-Shamir heuristic—we refer to [DPSZ12] for non-interactive variants.

$\mathcal{F}_{\text{ZK-mul}}$

On input $(\text{Enc}_{\text{pk}}(a), \text{Enc}_{\text{pk}}(c), b, r, \tilde{r})$ by party P_i:

Send $(\text{Enc}_{\text{pk}}(a), \text{Enc}_{\text{pk}}(c), P_i, \text{ok})$ to all parties P_j if $\text{Enc}_{\text{pk}}(c) = \text{Enc}_{\text{pk}}(a) \cdot b - \text{Enc}_{\text{pk}}(r, \tilde{r})$ and $b, r, \tilde{r}$ are short w.r.t. the respective bounds $B_{\text{plain}}, B'_{\text{plain}}, B'_{\text{rand}}$. Otherwise, send $(\text{Enc}_{\text{pk}}(a), \text{Enc}_{\text{pk}}(c), P_i, \text{fail})$.

Protocol 15: Ideal functionality for verifiable multiplication.

Let B^{τ}_{plain} and B^{ρ}_{rand} be the ZKP bounds introduced in [DPSZ12], i.e. $B^{\tau}_{\text{plain}} = 2^{\text{sec}}\rho$ and $B^{\rho}_{\text{rand}} = 2^{\text{sec}}\rho$. Let $V = 2\,\text{sec} - 1$, $M_e \in \{0,1\}^{V \times \text{sec}}$ the matrix associated to a challenge $e \in \{0,1\}^{\text{sec}}$ with $(M_e)_{ij} = e_{i-j+1}$ for $1 \leq i - j + 1 \leq \text{sec}$ and zero otherwise. Let $U, X \in R^{\text{sec}}$ be a plaintext vector, $R \in R^{\text{sec} \times 3}$ the encryption randomness. $\text{Enc}_{\text{pk}}(X, R) = (\text{Enc}_{\text{pk}}(X_i, R_i))_{1 \leq i \leq \text{sec}}$ denotes a vector where each row is a ciphertext. The ciphertext $\text{Enc}_{\text{pk}}(b)$ with $\|b\|_\infty \leq B^{\tau}_{\text{plain}}$ is public and has been verified as in [KPR18] or with some previous instance of this proof—note that $\text{Enc}_{\text{pk}}(b)$ is one-dimensional in the ciphertext space. We want to give a zero-knowledge proof of plaintext-knowledge for the following relation

$$\begin{aligned}\text{Rel}_{\xi,\chi\rho} = \{(E, w) : \; & E = (\text{pk}, C), w = (X, U, R) \in R^{\text{sec}} \times R^{\text{sec}} \times R^{\text{sec} \times 3} \text{ s.t.} \\ & C \leftarrow \text{Enc}_{\text{pk}}(B)U + \text{Enc}_{\text{pk}}(X, R), \|U\|_\infty \leq B^{\xi}_{\text{plain}}, \\ & \|X\|_\infty \leq B^{\chi}_{\text{plain}}, \|R\|_\infty \leq B^{\rho}_{\text{rand}}\}\end{aligned}$$

Completeness and zero-knowledge are only guaranteed for $\|U\|_\infty \leq \xi$, $\|X\|_\infty \leq \chi$ and $\|R\|_\infty \leq \rho$. τ, ξ, χ, ρ are negligible w.r.t sec compared to $B^{\tau}_{\text{plain}}, B^{\xi}_{\text{plain}}, B^{\chi}_{\text{plain}}, B^{\rho}_{\text{rand}}$.

Π_{ZKP}

1. The prover samples $W, Y \in R^V$ and randomness $S \in R^{V \times 3}$ such that $\|W_i\|_\infty \leq B^{\xi}_{\text{plain}}, \|Y_i\|_\infty \leq B^{\chi}_{\text{plain}}$ and $\|S_i\|_\infty \leq B^{\rho}_{\text{rand}}$ for all $1 \leq i \leq V$. The prover sends $A = \text{Enc}_{\text{pk}}(b)W + \text{Enc}_{\text{pk}}(Y, S)$ to the verifier.
2. The verifier selects $e \in \{0,1\}^{\text{sec}}$ and sends it to the prover.
3. The prover sets $Z = Y + M_e X$, $T = S + M_e R$, $Q = W + M_e U$ and sends it to the verifier.
4. The verifier sets $D = \text{Enc}_{\text{pk}}(b)Q + \text{Enc}_{\text{pk}}(Z, T)$ and accepts if Q, Z represent valid plaintexts and $D = A + M_e C$ and $\|Q_i\|_\infty \leq B^{\xi}_{\text{plain}}, \|Z_i\|_\infty \leq B^{\chi}_{\text{plain}}, \|T_i\|_\infty \leq B^{\rho}_{\text{rand}}$.

Protocol 16: The Zero-Knowledge Protocol.

Proof. Completeness follows as in [DPSZ12], Theorem 5:

$$\begin{aligned}D &= \text{Enc}_{\text{pk}}(b)Q + \text{Enc}_{\text{pk}}(Z, T) \\ &= \text{Enc}_{\text{pk}}(b)(W + M_e U) + \text{Enc}_{\text{pk}}(Y + M_e X, S + M_e R) \\ &= \text{Enc}_{\text{pk}}(b)W + \text{Enc}_{\text{pk}}(Y, S) + M_e(\text{Enc}_{\text{pk}}(b)U + \text{Enc}_{\text{pk}}(X, R)) \\ &= A + M_e C\end{aligned}$$

Also Q, Z, T are in the correct range with overwhelming probability. Given two transcripts one can find suitable X, R as in [DPSZ12]. To account for the additional U, we note that $(M_e - M_{e'})U = Q$ obviously has a solution. Hence we get soundness. Finally, honest verifier zero knowledge follows since W and $W + M_e U$ are indistinguishable. □

Please note that the increase in noise will result in a larger ciphertext modulus and hence will inrease bandwidth.

Remark 20. Please note that $\mathrm{Enc}(b)$ is the same in all components. In particular, the aggregated proof technique only uses its full potential if the parties need $bU_i + X_i$ for a high number of different U_i and X_i. This is e.g. the case for high degree polynomials.

At the end of this section we want to point to less oblivious techniques that work under certain circumstances, e.g. in an honest majority setup. For example, approaches like [CB17] and [BBC+19] use single-prover-multi-verifier proof systems where the statement is t-secret-shared between the verifiers and thus no group of $t-1$ verifiers knows anything about the statement. This suggests an approach where we would make all the intermediate results part of the statement in which case the verifying circuit would be quite straightforward and short. One party, P_j, could be the prover and all the others would be the verifiers. However, since we are interested in the case where $n-1$ out of n parties might be malicious, we run into a problem. Namely, [BBC+19] gives a negative result stating that it is unlikely that we obtain a protocol where both the prover and all-but-one of the verifiers are malicious.

Yet another alternative approach is to use multi-prover-single-verifier proof systems. The paper [CZC+21] suggests a proof system where for a publicly known statement x, the witness is split into several parts $w_1, \ldots, w_k$ where every prover knows just one of the witness parts. The provers prove that for a verifying circuit C, $C(x, w_1, \ldots, w_k) = 1$. This seems again naturally applicable to our case, as here P_i knows a_i and the randomness used to encrypt a_i, P_j knows $b_j, r_{i,j}$ and the randomness used to encrypt $r_{i,j}$ and so on. However, it is based on the MPC-in-the-head approach which suggests a considerable overhead as the protocol must be rerun a significant amount of times for privacy amplification.

Remark 21. To reduce the overhead introduced by noise flooding is an ongoing research task. There are however promising results as the ones announced in [Vai21] that might be applied in our setup, too.

D.3 Leveled Homomorphic Enryption

Given an encryption scheme Enc that is homomorphic with respect to at least $m-1$ multiplications, a binomial tuple can be produced by Protocol 17:[24]

Π_{SHE}

1. Each player P_i generates $[a_j]_i \in R$ for $0 \le j < m$ and $[f^e]_i \in R$ for $e = (e_0, \ldots, e_{m-1})$ and $0 \le e_j \le d_j$.
2. P_i computes and broadcasts $\mathrm{Enc}([a_j]_i)$ and $\mathrm{Enc}([f^e]_i)$ for all j and e as above.
3. P_i invokes a zero-knowledge functionality of plaintext knowledge $\mathcal{F}_{\mathrm{ZK}}$ as a prover for the created ciphertexts (cf. [DPSZ12]).
4. Compute locally $\mathrm{Enc}(a_j) \leftarrow \sum_{i=1}^{n} \mathrm{Enc}([a_j]_i)$, $\mathrm{Enc}(f^e) \leftarrow \sum_{i=1}^{n} \mathrm{Enc}([f^e]_i)$.
5. Compute locally $\mathrm{Enc}(a_e) = \prod_{j=0}^{m-1} \mathrm{Enc}([a_j])^{e_j}$ and $\mathrm{Enc}(a^e + f^e) = \mathrm{Enc}(a^e) + \mathrm{Enc}(f^e)$.
6. Decrypt $\mathrm{Enc}(a^e + f^e)$ to get $a^e + f^e$.
7. Set $[a^e]_1 \leftarrow a^e + f^e - [f^e]_1$ and $[a^e]_i \leftarrow -[f^e]_i$ for $2 \le i \le n$.

Protocol 17: Protocol for generation of $[a]^e$ for all $0 \le e_j \le d_j$ using leveled homomorphic encryption.

[24] We use the index notation from Section 3.4.

Once the shares of the tuples are created, they are authenticated using $\mathcal{F}_{[\![\cdot]\!]}$. The parties then use the new extended sacrificing technique to check that the tuples are well formed. Details can be found in Appendix D.1.

As in Remark 17 we remark that for our construction it is often enough to consider low degree polynomials that contain products with at most 5 factors. In these cases a homomorphic encryption scheme that supports 4 homomorphic multiplications is enough. The lowest degree polytuples that can be used to evaluate an arbitrary multivariate polynomial have entries which need at most 2 multiplications.

We remark that this approach profits from future improvements of the encryption scheme. Already existing optimizations like packing methods (e.g. [NLV11]) or using the natural action of the Galois group in case R is a underlying cyclotomic field extension (cf. [GHS12]), can be used to improve the performance of the offline phase.

E Prefix Products with Polytuples

Here, we describe how one can add on the approach presented in Section 4 to additionally get all the prefix products. For simplicity, we assume that we have to compute prefix products for m factors where m is a power of two. This is usually the case for comparisons where m is the number of bits used to represent values (e.g. when working with 32 bit or 64 bit numbers) and the construction presented next applies (with small modifications) to m of any shape.

First, note that the polytuples approach gives us $x_0 - a_0, x_0x_1 - a_{01}, \ldots, x_0 \cdots x_{m'-1} - a_{0,\ldots,m'-1}$ for all $m' < m$ that are again powers of two. $x_0 - a_0$ is an initial masked value and the other terms are randomized endodings or are publicly computed from them. We get similarly structured terms (again as masked value, randomized encoding, or publicly computed) with shifted indices, e.g. $x_2x_3 - a_{23}, x_4x_5 - a_{45}, x_4x_5x_6x_7 - a_{4567}$. The following construction either converts these terms directly to shares, or uses them with binomial tuples to compute the remaining terms. Note that all these terms are already masked and we can compute products with binomial tuples without an additional computation round. For example, we compute $[\![x_0x_1]\!] = x_0x_1 - a_{01} + [\![a_{01}]\!]$ and $[\![x_0x_1x_3]\!]$ by multiplying $x_0x_1 - a_{01}$ and $x_3 - a_3$ (by treating these values as the ones opened for normal multiplication with binomial tuples).

With the following (recursive) construction, we can compute all the prefix products: Assume we can get shares of the prefix products $p_{l,h,i}$ of $x_l, \ldots, x_h$ with $p_{l,h,i} = \prod_{j=l}^{i} x_j, l \leq i \leq h$ and have masked values as described above (computed with the polytuples approach of Section 4). Then, we can compute the shared prefix products of $x_0, \ldots, x_{2m-1}$ as follows:

1. Compute shares of the prefix products for $x_0, \ldots, x_{m-1}$ and $x_m, \ldots, x_{2m-1}$.
2. Compute $[\![p_{0,2m-1,m+i}]\!] = [\![p_{1,m-1,m-1}]\!] \cdot [\![p_{m,2m-1,m+i}]\!], 0 \leq i < m-1$.
3. The final $[\![p_{0,2m-1,2m-1}]\!]$ can be computed from an opened value as above.

Instead of computing these products in step 2 directly, we simply add one factor to the binomial tuple (or add a new degree-2 binomial tuple if $p_{m,2m-1,i} - a$ (for some mask a) was directly computed by our approach of Section 4).

By construction, we know that we need binomial tuples of a logarithmic degree. Additionally, we see that if the degree of the tuple for $p_{0,m-1,i}$ is smaller than the one for $p_{0,m-1,i+1}$, it is already covered by the latter one, decreasing the number of tuples we need to add. For powers of two ($m = 2^n$), we observe that we need $2^{n-1} - 1$ additional binomial tuples of degree at most n. We prove the latter (the number of additional tuples; the degree is fixed by construction) by

induction: The number of tuples for the case 2^{n+1} is what we need for $p_{0,2^n-1,i}$ ($2^{n-1}-1$) and for $p_{2^n,2^{n+1}-1,i}$. For the latter, we need 2^{n-1} tuples. This has the following reason. Of the 2^n-1 remaining prefixes to cover, $2^{n-1}-1$ already have a tuple candidate assigned to them. Of the remaining 2^{n-1} prefixes, $2^{n-1}-1$ are covered when expanding the previously mentioned tuples by one factor ($p_{0,2^n-1,2^n-1}$; this factor is also the only factor required for terms that did not have a tuple assigned to them before). In total, we have to add 2^{n-1} tuples for $p_{2^n,2^{n+1}-1,i}$ and get 2^n-1 tuples to compute prefixes $p_{0,2^{n+1}-1,i}$.

F Applications

Our polytuple approach is clearly relevant for applications where polynomials, arithmetic circuits, or products (of many factors) need to be computed. But it can also be used in applications which might seem less obvious.

Here, we present some example scenarios where polytuples can be used and in Section 5 we sketch our implementation and first benchmarks. These show improvements due to our approach for all tested applications.

As mentioned, the most natural application is to use our tuples as primitives in MPC protocols (e.g. SPDZ [DPSZ12] and similar protocols) to compute polynomials in $\mathbb{F}_p$. Most applications that perform operations on integer-valued data can benefit from polytuples directly. In certain real-world applications, e.g. to compute the soft-max function in privacy-preserving machine learning, polynomials are also evaluated on fixed-point representations of real numbers $\mathbb{R}$. Since fixed-point numbers often require rescaling intermediate results (truncation) after a few multiplications to avoid overflow in the underlying finite field representation. Polytuples of small polynomial degree as discussed in Theorem 1 and Remark 17 could be a good fit for these applications. A detailed discussion on polynomial evaluations over $\mathbb{R}$ with polytuples is, however, left to future work. As we demonstrate next, there are applications where our polytuples approach can be applied to both integer-valued and fixed-point data immediately.

Comparisons. Our approach can also be used to speed-up comparisons, i.e. equality tests ($x = y$) and inequality tests ($x < y$, $x \leq y$, etc.). Comparisons are an ubiquitous operation in MPC, for example, in secure online auctions, linear programming, secure clustering, secure floating-point addition, private decision tree schemes, private sorting, and electronic voting, to name just a few. Also in machine learning applications, we find comparisons, e.g. in ReLU, MaxPool, or ArgMax layers of deep neural networks.

Classical approaches for comparisons are built on evaluating k-ary symmetric boolean functions (e.g., AND and OR; cf. [CdH10, CS10, NO07]). They often use (not maliciously secure) techniques as in [BB89, DFK+06, LYKM22] to get constant-round protocols. Instead, we can express these boolean operations as multiplications ($[\![x \wedge y]\!] = [\![x \cdot y]\!]$, $[\![x \vee y]\!] = [\![1-(1-x)\cdot(1-y)]\!]$) and evaluate them with our tuples. Some also need prefix operations, e.g. prefix-ORs, which we can simply represent as prefix products. Details on how to use our polytuples to compute prefix products can be found in Appendix E.

To give a concrete example, we briefly look at a standard approach for equality and less-than tests [CdH10, NO07], where comparing two secret-shared values is reduced to two basic operations: bit-wise equality tests and bit-wise less-than tests with one shared and one public input (see Protocols 18 and 19).

Checking equality of $[\![x]\!]_i$ and $[\![y]\!]_i$ is a straightforward zero test of $[\![x-y]\!]_i$, which in turn is an equality test of a public value $c = x - y + r$ and a (bit-wise) shared value r (cf. Protocol 18).

Π_{EQ}

1. Let $[\![r_j]\!]_i$ and c_j be the inputs (bit-decomposed; index $0 \leq j < k$ for the jth bit).
2. Let $[\![e_j]\!]_i = (c_j = [\![r_j]\!]_i) = 1 - [\![r_j]\!]_i - c_j + 2c_j[\![r_j]\!]_i$ for $0 \leq j < k$.
3. Let $[\![e]\!]_i = \bigwedge_{j=0}^{k-1} [\![e_j]\!]_i = \prod_{j=0}^{k-1} [\![e_j]\!]_i$.
4. Return $[\![e]\!]_i$.

Protocol 18: Bit-wise equality test protocol [NO07].

Π_{LT}

1. Let $[\![r_j]\!]_i$ and c_j be the inputs (bit-decomposed; index $0 \leq j < k$ for the jth bit).
2. Let $[\![d_j]\!]_i = c_j \oplus [\![r_j]\!]_i = c_j + [\![r_j]\!]_i - 2c_j[\![r_j]\!]_i$ for $0 \leq j < k$.
3. Let $[\![f_{k-1}]\!]_i, \ldots, [\![f_0]\!]_i = \text{PrefixOR}([\![d_{k-1}]\!]_i, \ldots, [\![d_0]\!]_i)$.
4. Let $[\![g_{k-1}]\!]_i = [\![f_{k-1}]\!]_i$ and $[\![g_j]\!]_i = [\![f_j]\!]_i - [\![f_{j+1}]\!]_i$ for $0 \leq j < k-1$.
5. Let $[\![h_j]\!]_i = c_j[\![g_j]\!]_i$ for $0 \leq j < k$.
6. Let $[\![h]\!]_i = \sum_{j=0}^{k-1} [\![h_j]\!]_i$.
7. Return $[\![h]\!]_i$.

Protocol 19: Bit-wise less-than protocol [DFK$^+$06].

We see that this protocol involves two operations with communication: (i) a masked opening (not pictured in Protocol 18) and (ii) a multiplication of k shares. The latter is a native operation with our tuple-based approach. All other operations are local operations on shares.

Inequality tests of $[\![x]\!]_i$ and $[\![y]\!]_i$ (to compute $[\![x \leq y]\!]_i$) can be done as in [CdH10]. This also involves a masked opening and a bit-wise comparison. We only depict the core of the inequality protocol, the bit-wise less-than protocol (Protocol 19). The version shown here is based on the classical less-than protocol in [DFK$^+$06] and turns out to be more efficient than the ones of [CdH10, Rei09] (as we can avoid one round of communication that is needed to work with information-leaking (passively secure) constant-round multiplication protocols). Only the single prefix-OR in Protocol 19, which can be expressed as a single prefix multiplication with inverted inputs and outputs, requires communication—the other operations are linear, and thus, can be done locally on shares. A prefix-OR is again a native operation with our tuples.

Evaluating comparisons with our tuples is more efficient standard techniques in SPDZ-like protocols as we can now use constant-round techniques based on our constant-round (prefix) multiplication. Please note that there are MPC protocols specically crafted to optimize comparisions. However, to use these protocols together with SPDZ expensive conversations are needed and separate benchmarks for comparision can hence not be easily compared. We therefore decided to restrict our comparision to two efficient protocols for comparision included in MP-SPDZ.

Rankings. For auctions (or e-voting), one often needs to compute a ranking of the bids (or votes) and reveal the top k results (e.g. with $k = 1$ only the highest bid or the candidate with the most votes). Obviously, one can also compute arbitrary functions in MPC of this result before revealing it (e.g. for tally-hiding e-voting [KLM$^+$20]). Note that e-voting (or auctions) might require additional security properties (e.g. public verifiability or identifiable abort) that are not directly provided by our protocol. However, this can be achieved with extension to SPDZ that have these properties [BDO14, BOS16, CFY17]. Our approach is fully compatible with these SPDZ-based protocols.

There are several ways to compute a ranking. Computing a comparison matrix (containing $x \leq y$ for all pairs x, y) is most versatile as one can compute many functions from it [KLM+20]. Two straightforward ways of computing the matrix include computing the matrix directly and computing it from two triangular matrices $(x_i \leq x_j)_{0<i<j}$ and $(x_i = x_j)_{0<i<j}$. Hence we can use the construction above compute the (in)equality test with polytuples. We tested both approaches and compare them to the respective default implementation in MP-SPDZ (based on the protocols with logarithmic complexity in [CdH10]; with and without edabits [EGK+20] to speed up the comparison). The benchmark results were included in Fig. 6.

Neural Networks. We also provide benchmarks for ML Applications. Namely, we use 4 benchmark programs available in MP-SPDZ. These programs combine different ML layers to reflect different architecture common in dense and convolutional network:

- Benchmark Net A contains the following layers in this order: Dense, Square, Dense, Square, Dense, ArgMax.
- Benchmark Net B contains the following layers in this order: 2d Convolution, MaxPool, ReLU, 2d Convolution, MaxPool, ReLU, Dense, ReLU, Dense, ArgMax.
- Benchmark Net C has layers as B but with different dimensions.
- Benchmark Net D contains the following layers in this order: 2d Convolution, ReLU, Dense, ReLU, Dense, ArgMax.

For further specifics on the layers, e.g. number of inputs, we refer to the corresponding programs in MP-SPDZ [Kel20]. Please note that each program comes with an ArgMax layer and hence uses comparisons. With the previous construction we can therefore use polytuples to speed of these computations. The benchmarks were/are included in Fig. 7 and Fig. 10.

G Further Specifics of the Implementation and Evaluation

The results of Figs. 5 to 7 were obtained by averaging 32 program runs for each parameter setting (e.g. fixed delay, number of variables and, degree). In all our experiments we introduced an artificial network delay/latency using the tc(8) Linux tool. This gives us control to simulate various network settings. Our first benchmark (evaluation of multivariate polynomials) was tested with 2 ms, 5 ms and 10 ms delay to also show the effect parameters besides the delay (the number of variables and the maximum degree in each variable).

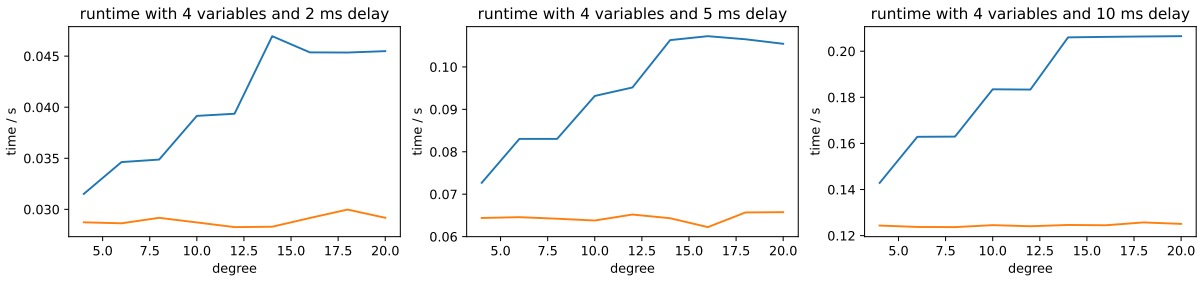


Fig. 9: Further Benchmarks for polynomial evaluation (blue: default MP-SPDZ implementation, orange: ours).

The other benchmarks (rankings and neural networks) were run with delays from 0 ms to 20 ms (in steps of 1 ms below 10 ms and 2 ms steps above 10 ms delay). For the ranking benchmark, we chose to compare our implementation to MP-SPDZ with edabits [EGK+20] as it is MP-SPDZ's recommendation for our test program. We can see that this is indeed an improvement over the standard implementation,[25] however our new tuple-based approach clearly beats both existing approaches.

For the Machine Learning benchmark, we chose to not vary any parameters of the models. Instead, networks A and D correspond to smaller/simpler models, while networks B and C are larger/more complex (approximately ordered by size/complexity: A < D < B < C).

Finally, our implementation lacks certain features that would (when implemented correctly) only speed-up any application. This includes finding optimal partitions for products; currently, polytuples are created naively by simply splitting products in half recursively instead of finding tree structures (cf. Figure 3) with optimal size and/or better bandwidth. Bandwidth and/or size optimal partitions could be implemented on top of our results from Section 4 instead. Another optimization opportunity is the one shown in Protocol 5 (combining the evaluation of a polynomial with the masking step of the next polynomial evaluation). This would allow us to combine the opening round of one computation with polytuples and the input round of another computation. Currently, every operation based on polytuples[26] takes two rounds as we always create a share of the result; the sequential composition of two such operations takes four rounds and so on.

Effect of Bandwidth Rate Restrictions. To better understand the effect of our approach on neural networks we also give the benchmarks for the ArgMax Layer seperately. Additionally this evalution was done with different bandwidth restrictions imposed—50 Mbit/s, 1 GB/s, unlimited. The results in Figures 7 and 10 show that there is no significant impact of the bandwidth overhead in this example. Similar results hold for all our evaluations.

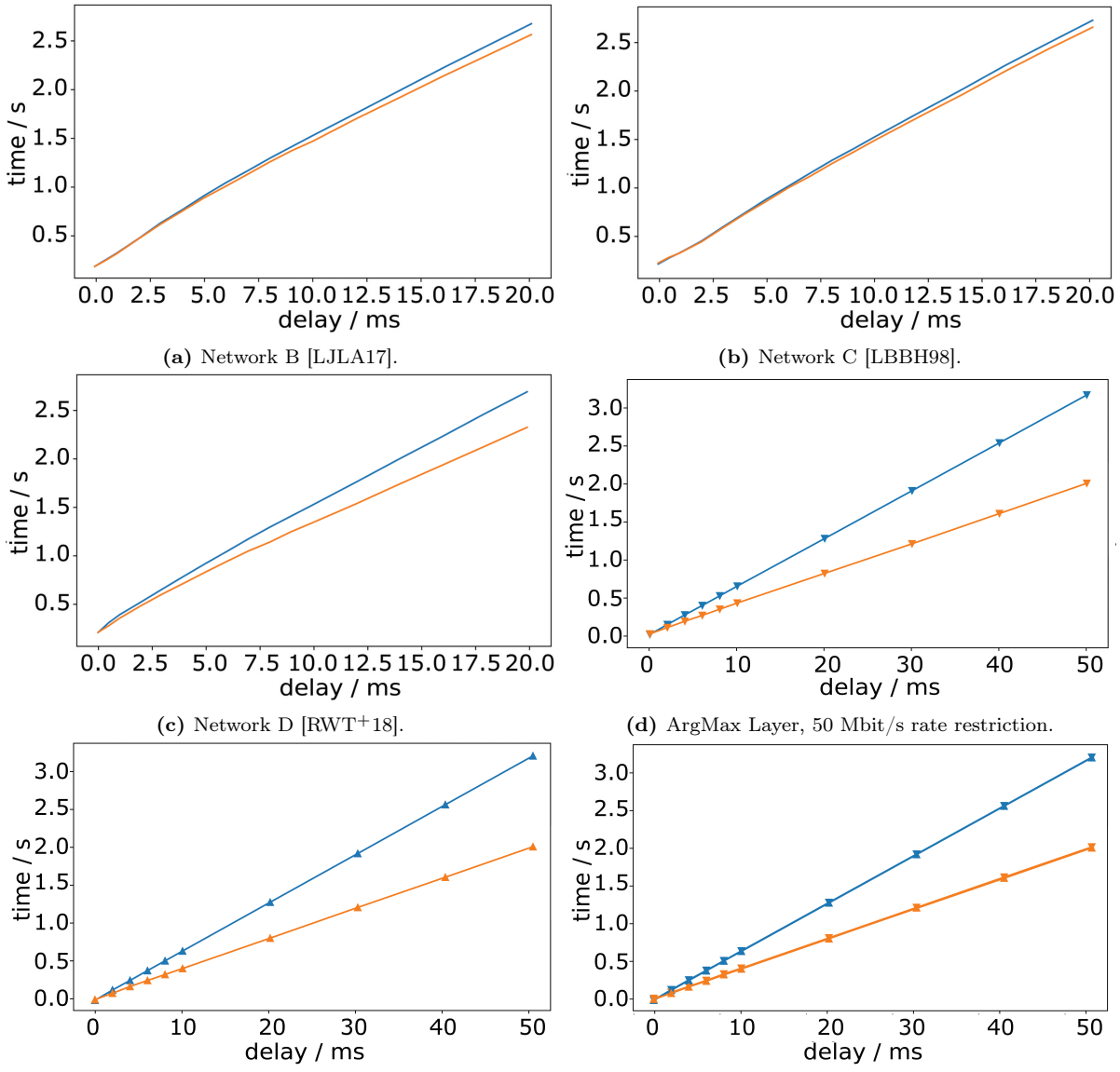
Benchmarks for Different Numbers of Parties Please recall from Section 3 that we assume (similar to [DPSZ12]) that parties broadcast their shares to all other parties (to open a value). Hence our benchmarks are expected to scale linearly in the number of parties n. Note, that the final MAC check is not linear in n, but it has to be done only once and is circuit-independent. However, just like in MP-SPDZ the MAC checks in our implementation are done more regularly to simplify the code. In particular, the non-linear contribution of the MAC check then becomes circuit-dependent. We illustrate the behavior for different numbers of parties in Fig. 11. We remark that the slight circuit-dependence of the MAC check is usually considered acceptable.

H Further Related Literature

In this appendix we extend our exposition of related work in Section 2.

Since polynomial evaluation is one of the most fundamental arithmetic tasks, several solutions outside of SPDZ-like protocols or even MPC have been suggested over the last 30 years. To mention only a few different ideas: [MF06] uses shared polynomials, [FM10, DMRY11] use homomorphic encryption in the online phase, [GMRW13] also uses homomorphic encryption but in a single centralized server setup. Of course any fully homomorphic encryption scheme like the

[25] Note that edabits are an improvement in Fig. 6b only for very low latency.
[26] Except operations with binomial tuples; these are implemented in one round.



(a) Network B [LJLA17].

(b) Network C [LBBH98].

(c) Network D [RWT+18].

(d) ArgMax Layer, 50 Mbit/s rate restriction.

(e) ArgMax, 1 Gbit/s rate restriction.

(f) Combination of ArgMax.

Fig. 10: Figures (a), (b), (c) contain the benchmarks for the evaluation of the neural networks B, C, D included in MP-SPDZ [Kel20] (cf. [RWT+18]; blue: default MP-SPDZ implementation, orange: ours). (d), (e) contain benchmarks for the ArgMax layer with bandwidth restrictions. (f) places (d), (e) and the ArgMax layer in Fig. 10 in one diagram to show that the bandwidth restriction has no visible effect on the runtime.

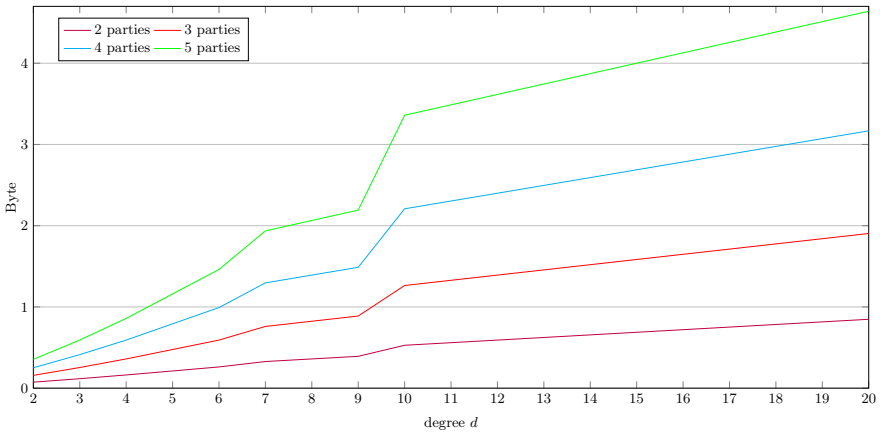


Fig. 11: Online bandwidth for the computation of $x_0 \cdots x_{d-1}$ for a different number of parties.

original protocol by Gentry [Gen09] can also be used to evaluate a polynomial. Another idea is to use oblivious transfer-based techniques like in [NP99] or [GM09, TJB13] where one party holds the polynomial f and the other party holds the input variables $x_0, \ldots, x_{m-1}$. Yet another recent idea is to compute multivariate polynomials of time-series data utilizing private stream aggregation (PSA) and trusted execution environments (TEEs) as in [KTM+21].

In [BDG+17] or [SA19], public verifiability of a polynomial evaluation is studied. We remark that our protocols can be extended to support (public) verifiability or (publicly) identifiable abort similarly to known extensions of [DPSZ12], e.g. [BDO14, BOS16, CFY17].

Since our paper aims at minimizing communication, we also want to shortly point to a more detailed discussion on the importance of communication rounds in MPC, e.g. in [AKP20, BNTW12] or [FM19].

Finally, there is also the recent research direction of non-interactive MPC (cf. [EOYN21, HHPV21, HIJ+17, HIJ+16, KBTJ19]) where parties send data online once and reconstruct the result locally without an opening round. However, these protocols are either vulnerable to residual function attacks or use trusted hardware (e.g., TEEs).