Arithmetic Tuples for MPC

Pascal Reisert[1][0000−0003−1808−6140], Marc Rivinius[1][0000−0001−8005−8365], Toomas Krips[2], and Ralf Küsters[1][0000−0002−9071−9312]

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

Abstract. Some of the most efficient protocols for Multi-Party Computation (MPC) use a two-phase approach where correlated randomness, in particular Beaver triples, is generated in the offline phase and then used to speed up the online phase. Recently, more complex correlations have been introduced to optimize certain operations even further, such as matrix triples for matrix multiplications. In this paper, our goal is to speed up the evaluation of multivariate polynomials and therewith of whole arithmetic circuits in the online phase. To this end, we introduce a new form of correlated randomness: *arithmetic tuples*. Arithmetic tuples can be fine tuned in various ways to the constraints of application at hand, in terms of round complexity, bandwidth, and tuple size. We show that for many real-world setups an arithmetic tuples based online phase outperforms state-of-the-art protocols based on Beaver triples.

A major extension of this technical report has appeared in ASIACRYPT 2024 and as [64].

1 Introduction

Multi-Party Computation (MPC) allows several parties to compute an arithmetic circuit on private inputs without revealing information about the inputs apart from the result. Modern two-phase protocols, like SPDZ [29,30] and related protocols [4,48,49], consist of an offline phase, where (structured) random data, classically in the form of Beaver triples [7], is precomputed, and an online phase, where the precomputed data is used to compute the desired output from the private inputs. This general design principle allows parties to speed-up the online phase considerably. A reasonably less-efficient offline phase is usually considered acceptable since preprocessing can start well before the input data becomes available. Efficiency in these types of two-phase protocols and generally in MPC-protocols heavily depends on the number of communication rounds needed and the bandwidth, i.e. the amount of data that has to be sent. Local computation times are often considered less relevant for real-world applications as long as hardware requirements, e.g. memory requirements, do not get out of hand.

Table 1: Comparison for the evaluation a degree $d = \sum_{j=0}^{m-1} d_j$ monomial f in m variables with Beaver triples, binomial tuples, and arithmetic tuples.

Approach	Rounds	Bandwidth	Tuple Size
Beaver Triples	$\lceil \log d \rceil$	$2(d-1)$	$3(d-1)$
Binomial Tuples	1	m	$\prod_{j=0}^{m-1}(d_j+1)-1$
Example Intermediate Arithmetic Tuple	1	$\mathcal{O}(m\log(m))$	$\mathcal{O}(d\log(m)^2)$

In actively secure MPC protocols like SPDZ (and improvements thereof) classical multiplication with Beaver triples [7] is still the most prevalent technique to evaluate arithmetic circuits efficiently. The approach usually comes with logarithmically many rounds, communicating linearly many elements, with a linear amount of preprocessed data in the depth of the arithmetic circuit. Recently, a natural extension of the classical Beaver technique emerged that allows to reduce the round complexity[3] down to 1 and has only around half the bandwidth of Beaver triple based protocols [21,27,61,63]. However, the underlying form of correlated randomness, which we call *binomial tuples*, will in general have exponential size in the number of inputs. Therefore, this approach has only been used for products of a few variables or exponentiation of a single variable [27,29].[4]

In this paper, we propose a new online protocol Π_{online} based on a new form of correlated random tuple, which generalizes both Beaver's technique and binomial tuples. The new online phase then allows to evaluate any arithmetic (sub-)circuits and therewith any multivariate polynomial in a minimal number of rounds (just as binomial tuples) but with a much more moderate tuple sizes. Security-wise, our technique still provides, just like SPDZ and related-protocols, active security as long as one party remains honest.

We want to shortly describe the high-level idea of our approach. A SPDZ-like online phase has the following characteristics: at the begining parties possess (among others) shares of the input variables, they perform a series of local computations and communication to produce (masked) intermediate results and possibly output at some point a final result.[5] In current protocols this process is usually implemented using the standard properties of additive sharing, i.e. to produce an intermediate or final value y, each party P_i first locally computes an additive share $[y]_i$ thereof, which is then published and summed up: $f'([y]_1, \ldots, [y_n]) := \sum_{i=1}^{n} [y]_i = y$. This is however, not necessary. One can replace f' by any circuit on the set P of information available to the parties. Given P, the parties can then *locally* compute the intermediate/final result. It is often advantageous to use the usually stronger local computation power by shifting more of the overall computation into a then more complex circuit f'. Naturally,

[3] We refer to Section 3 for more details on how to count communication rounds.

[4] Note that an increased tuple size not only slows down the offline phase that has to generate these huge tuples, also the online runtime increases as the data contained in the tuples has to be processed.

[5] Other outputs, e.g. a share for each party, are also common and compatible with our online protocol (cf. Protocol 1.2).

certain limitations apply to f' and P, for example the input size should remain within practical range for two reasons: (i) for a very large input size (e.g. exponential) the local evaluation of f' might still become the bottleneck of the overall Multi-Party computation; (ii) all information in P has to be created either by the preprocessing or by transmissions from the other parties and will therefore either increase the bandwidth or slow down the preprocessing phase (similar to the case of binomial tuples). To satisfy these constraints, we will build for each polynomial f in m input variables $x_0, \ldots, x_{m-1}$ a sufficiently small set of published values $B_f \subset P$ and a circuit f' such that $f(x_0, \ldots, x_m) = f'(B_f)$, i.e. the parties can compute $f(x_0, \ldots, x_m)$ locally on B_f using f'. We call elements of B_f *building blocks*. In order to construct the building blocks in B_f we will use a new form of correlated randomness called *arithmetic tuples*. As mentioned before the size of the correlated random data, i.e. the arithmetic tuple size, also affects performance and has to be kept reasonably low.[6]

Another natural requirement in our MPC setup is that B_f is *privacy-preserving*, i.e. apart from the final result no information should be leaked. As it turns out this makes the construction of f' and B_f technically challenging. The reason is that more complex circuits f' that contain many multiplication gates are not easily compatible with the information-theoretically secure additive one-time pad masks we use.[7] One of the main contributions of this paper is therefore the proof that it is possible to build any polyomial f from a reasonably small set of building blocks B_f such that the elements of B_f can still be constructed with an *arithmetic tuple* of practical size. In fact, we will see in Section 5.2 that we can find a suitable B_f that consists only of polynomials of small degree ≤ 3 such that each polynomial can be constructed by an arithmetic tuple of size at most 8.

In summary, a protocol run will roughly look as follows: To compute $f(x_0, \ldots, x_{m-1})$ the parties precompute in an actively secure offline phase suitably formed shared arithmetic tuples which, among others, contain masks $a_0, \ldots, a_{m-1}$ for each input variable. The parties open the masked values $x_j - a_j$ for $0 \leq j < m$. Subsequently, the parties use the masked values $x_j - a_j$ together with shares of the remaining arithmetic tuple entries to locally compute shares of privacy-preserving building blocks. In principle, a building block can now be any shared data that the parties can locally compute given $x_j - a_j, 0 \leq j < m$ and the shared tuple. The parties open all building blocks at once. By construction this allows them to locally compute $f(x_0, \ldots, x_{m-1})$ as $f'(B_f)$.

The choice of the circuit f' and the resulting number and shape of building blocks and arithmetic tuples strongly influence various aspects of the online phase for the parties. For example, a more shallow circuit f' and/or high degree building blocks reduce the overall number of building blocks. Since building blocks are opened this decreases the bandwidth. The tradeoff are larger tu-

[6] The tuple size naturally affects the offline phase. It might also affect the online phase, e.g. in the case of an exponential tuple size, since the data contained in the tuples has to be processed like in the case of binomial tuples.

[7] We refer to Section 5.1 for further details and examples.

ple sizes (Section 5.2 for the explicit formulas for tuple size and bandwidth). Hence, the choice of f' makes our approach very flexible because it allows parties to evaluate polynomials in many different ways. That is, the evaluation can be tailored and optimized w.r.t. the needs of parties. Table 1 shows one specific kind of arithmetic tuple. This tuple lies between the linear size for Beaver multiplication and the exponential size for binomial tuples, has minimal round complexity, and a higher bandwidth cost than the other approaches. Almost all other trade-offs are however possible. For example, we can also construct an arithmetic tuple that keeps the round complexity at 1 and achieves a bandwidth in $\mathcal{O}(m \log\log(m))$ (or basically anything between the bandwidth m for binomial tuples and $\mathcal{O}(m \log(m))$)—the tuple will then however have a tuple size somewhere between $\mathcal{O}(d \log(m)^2)$ and $\prod_{j=0}^{m-1}(d_j + 1) - 1$. The exact relation will be explained in Section 5.2.

Since our protocol is also composable we can split up an arithmetic circuit into subcircuits and apply the arithmetic tuples approach to evaluate each of them. This feature adds additional flexibility since it allows us to trade round complexity and bandwidth/tuple size; Figure 2 illustrates that adding just one round can already make a big difference.

In summary, we can construct local arithmetic circuits f', privacy-preserving building blocks and suitable arithmetic tuples such that we get bandwidth, tuple size, and/or round complexity (almost) anywhere between Beaver triples and binomial tuples. So, arithmetic tuples will improve the online performance of (almost) any SPDZ-like MPC protocol and can be tuned for optimal performance in the concrete setting where the protocol is deployed.[8] 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 arithmetic tuples with lower bandwidth. If the runtime of the offline phase, local memory and/or computation time are important, striving for small tuple sizes is recommended. Our first experiments show that strategic deployment of arithmetic tuples can significantly speed-up the performance of the online phase.

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

- We introduce arithmetic tuples, which generalize the concept of Beaver triples and binomial tuples. These tuples allow parties to evaluate a multivariate polynomial or an arithmetic (sub-)circuit in just one round of online communication plus one opening round. Our tuple size is significantly lower than for existing single-round approaches.
- We compute the tuple size and bandwidth needed in the online phase for all types of arithmetic tuples and discuss the multi-round use of our online protocol. Our tuple size is significantly lower than for existing single-round approaches and also multi-round computations yield improvements (e.g. lower

[8] The sole exception is a setup with no or a very small latency and products of exactly 2^k input variables for some $k \in \mathbb{N}$. In this special case our technique will coincide with classical Beaver multiplication and will have the same performance as e.g. SPDZ.

bandwidth and round complexity than Beaver multiplication). More generally, our resulting arithmetic tuples based online protocol Π_{online} allows to optimize efficiency w.r.t. the number of rounds, bandwidth, and tuple size. Π_{online} guarantees active security as long as one party is honest.
- We present different protocols for the generation of arithmetic tuples as well as an new extended sacrificing technique compatible with arithmetic tuples (cf. Section 6 and Appendix C.1).
- We evaluate the performance of our approach for sample applications (evaluation of polynomials, comparisons of secret-shared values, simple machine learning algorithms) in Section 8 which shows that arithmetic tuples speed-up these computations compared to Beaver multiplication and binomial tuples.

Structure of the Paper. We start by recalling some basic notation and concepts in Section 3 and present *binomial tuples* in Section 4. We then introduce *arithmetic tuples* in Section 5, study their size and bandwidth requirements, and integrate them in our (online) MPC protocol. Section 6 focuses on the tuple production, i.e. the offline phase. Example applications and benchmarks are presented in Sections 7 and 8, respectively. Section 2 covers related work. We conclude in Section 9. More details can be found in the appendix, which is submitted as part of the supplementary material, which also includes our implementation.

2 Related Work

We see our work as an improvement to the common online phase of SPDZ [30] and related Protocols [48,49,4]. We therefore concentrate our discussion on recent progress applicable to SPDZ-like papers, rather than classical theoretical results like [3,7,44,23].

A first small optimization of the Beaver triple-based online phase in SPDZ already appeared in [29] where square pairs are used to improve the squaring of secret shared values. This idea has been picked up by Morton Dahl who describes in [27] a variety of generalized tuples that can improve the online phase. These include power tuples for the computation of a monomial x^d for a secret-shared value x, which are binomial tuples (cf. Section 4) for a single variable. Dahl [27] also presents matrix triples and convolution triples which have also been discussed in [57] in the passively secure domain. Matrix (and convolution) triples have since then seen further attention and are by now available as part of an actively secure protocol [20]. The multivariate version of binomial tuples appears in the passively secure protocol of [21] with additional trust assumptions on the dealing server, whereas the authenticated *binomial tuples* in this paper provide active security. Ohata and Nuida [61] 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 [3] and again in [28]. The more recent minor extension presented in [54] uses random multiplicative masks r_j to hide the inputs x_j. The parties then use Beaver triples to compute and open $y_j = x_j r_j$. A share of the product

$\prod_{j=1}^m x_j$ is given by the product of the y_j times a share of the product of the inverses r_j^{-1}, i.e. $\left[\prod_{j=1}^m x_j\right] = \prod_{j=1}^m y_j \left[\prod_{j=1}^m r_j^{-1}\right]$. A standard technique from [28] can ensure that a possible 0 value is not detectable and the protocol becomes passively secure. The combined passively-secure protocols needs $4+1+2$ rounds of (online) communication (cf. [18]).[9] The extension in [28] to an actively secure protocol uses a cut-and-choose technique, which is generally not compatible with the MAC-authenticated approach of SPDZ-like protocols. We remark that [54] also presents a protocol to compute powers of matrices (building on [3])—the papers do not discuss the case of matrices of non-full rank; the case of non-full rank matrices leaks information just as the zero case does for field elements. The general idea to use a multiplicative structure in the underlying primitives, e.g. a multiplicative secret sharing as in [10,38], is quite tempting. However, these multiplicative sharings can generally not 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 these approaches actively secure (if possible) comes with a considerable overhead.

Futhermore, there are many papers optimizing the use of maskings/tuples. Boura et al. [13] make use of the exponential identity ($\exp(x+y) = \exp(x) \cdot \exp(y)$) to employ an additive sharing for a multiplicative computation—a Fourier series expansion further allows one to use the functional properties of the exponential function to compute approximations of arbitrary smooth functions, too. How the necessary sharings of the exponential function can be created without a trusted third party in an offline phase (passively or even actively secure) is not discussed.

Boura et al. [13] reuse their masks for certain input variables for different multiplication gates. Dahl [27] extends the idea to reuse already constructed randomness to be applicable to neural networks. Function-dependent preprocessing can decrease the required tuple size and bandwidth in the online phase [8,63]. Also note that with a pseudo-random generator, as, e.g., in [14], structured randomness can be produced without further communication. Special solutions also exist for more complex structured random data like the matrix triples mentioned before ([20,57]).

3 Preliminaries

For our theoretical considerations in Section 5.2 we are working on a commutative base ring R. For all other parts we choose R a finite field as in [30]. We call a computation local if the parties can perform it without interaction.

[9] 4 rounds to check if x_j is zero, 1 round to replace x_j with a non-zero x_j' if x_j was zero, and 2 rounds to compute and open y_j; an additional round is necessary to set the product to zero if any of the x_j was zero; if wrong results (due to the x_j') are acceptable in this case, this does not add an additional round.

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 1.9)—to account for the different structures of an opening round we will count opening rounds separately, usually indicated by a "+1" in the round 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 hold independently of the used secret-sharing scheme.

In SPDZ and related protocols, shares are additionally authenticated to verify the outputs of the protocol using a MAC key [29,30]. The MAC key $\alpha \in R$ is shared in the preprocessing phase. Secret shared values (including inputs and structured randomness like Beaver triples or arithmetic tuples) are authenticated in the offline phase—we use $[\![x]\!] \coloneqq ([x], [\alpha x])$ to denote authenticated shares of x and $[\![X]\!] = ([\![x_1]\!], \dots, [\![x_k]\!])$ for a tuple $X = (x_1, \dots, x_k)$. Linear operations on authenticated shares are a trivial extension of linear operations on shares with the exception of $[\![x+c]\!]_i \coloneqq ([x+c]_i, [\alpha x]_i + c \cdot [\alpha]_i)$. A MAC check enables parties to verify the integrity of previously opened shares (cf. Protocol 1.9 or [29,30]). The soundness of the MAC check is proportional to $\frac{1}{|R|}$, can be aggregated over many opened linear combinations, and does not reveal the MAC key [29].

4 Binomial Tuples: Composable, One Round, Small Bandwidth, Large Tuples

In this section, we discuss a natural generalization of Beaver triples based on binomial expansion. Our goal is to compute a polynomial f in m variables $x_0, \ldots, x_{m-1} \in R$ of total degree $d = \sum_{j=0}^{m-1} d_j$ with one round of communication plus one opening round. A passively secure version of this extension was used in [21]. Recall the binomial expansion for one variable x: $x^d = (x - a + a)^d = \sum_{e=0}^{d} \binom{d}{e} a^e (x-a)^{d-e}$. Then $\left[x^d\right]_i = \sum_{e=0}^{d} \binom{d}{e} \left[a^e\right]_i (x-a)^{d-e}$ is an additive sharing of x^d based on sharings of a^e and the publicly known/previously opened $x - a$. One can generalize this construction to several variables in the straightforward way. Namely, let $a_j \in R$ be a shared value for each $0 \leq j < m$. Define $a^e := \prod_{j=0}^{m-1} a_j^{e_j}$ for $e = (e_0, \ldots, e_{m-1}) \in \bigtimes_{j=0}^{m-1} \{0, \ldots, d_j\} =: E$. Note that $a_0 = a^{(1,0,\ldots,0)}, \ldots, a_{m-1} = a^{(0,\ldots,0,1)}$ are already included in the a^e. Furthermore, $a^{(0,\ldots,0)} = 1$ is independent of the a_j and therefore a publicly known constant. Hence, we have a tuple size of $\prod_{j=0}^{m-1}(d_j + 1) - 1$.

Now we can construct a sharing of $f(x_0, \ldots, x_{m-1}) = \prod_{j=0}^{m-1} x_j^{d_j}$ using a shared tuple $[(a^e)_{e \in E}]$ by $[f(x_0, \ldots, x_{m-1})]_i = \sum_{e \in E} [a^e]_i \prod_{k=0}^{m-1} \binom{d_k}{e_k} (x_k - a_k)^{d_k - e_k}$. This is in fact a sharing of $f(x_0, \ldots, x_{m-1})$ since

$$\begin{aligned} \sum_{i=1}^{n} [f(x_0, \ldots, x_{m-1})]_i &= \sum_{e \in E} a^e \prod_{k=0}^{m-1} \binom{d_k}{e_k} (x_k - a_k)^{d_k - e_k} \\ &= \prod_{k=0}^{m-1} \sum_{e_k=0}^{d_k} \binom{d_k}{e_k} a_k^{e_k} (x_k - a_k)^{d_k - e_k} = \prod_{k=0}^{m-1} x_k^{d_k}. \end{aligned} \tag{1}$$

We call tuples of the form $[(a^e)_{e \in E}]$ *binomial tuples*. To be used in a maliciously secure online phase one extends binomial tuples in the usual way by adding MACs. We get *authenticated binomial tuples* $[\![(a^e)_{e \in E}]\!] := (([a^e], [\alpha a^e]))_{e \in E}$ and the correctness of the resulting online phase follows from Equation (1).

Remark 1. The construction linearly extends to polynomials of the form $g(x_0, \ldots, x_{m-1}) = \sum_{e \in E} g_e \prod_{j=0}^{m-1} x_j^{e_j}$ with $g_e \in R$ coefficients, where each a^e has to occurs at most once in the binomial tuple. For example, for $x_0^3 x_1 + x_0 x_1^3$ we need an 11-tuple $[\![(a^{(1,0)}, a^{(2,0)}, a^{(3,0)}, a^{(0,1)}, a^{(1,1)}, a^{(2,1)}, a^{(3,1)}, a^{(0,2)}, a^{(0,3)}, a^{(1,2)}, a^{(1,3)})]\!]$.

At (a naive) first glance binomial tuples seem to have a clear advantage compared to a classical Beaver multiplication based online phase like in [30] if we want to secretly evaluate a polynomial on shared data: the round complexity is minimal and also the data sent in the online phase is small. E.g. for a degree $d = \sum_{j=0}^{m-1} d_j$ monomial in m variables, classical Beaver multiplication (without authentication or circuit-dependent preprocessing as in [8]) uses $\lceil \log(d) \rceil + 1$ rounds and has to send at least $2d - 1$ elements per party to open the result. Binomial tuples on the other hand allow one to compute the same monomial in just

two rounds of communication (including the opening round) with a bandwidth of $m+1$ field elements per player. The effect is even stronger for monomials in one variable or the simultaneous computation of several of these powers. However, for $\prod_{j=0}^{m-1} x_j^{d_j}$ the binomial tuple is of size $\prod_{j=0}^{m-1}(d_j+1)-1$. In particular, for $d_j = 1$ for all $0 \leq j < m$, the tuple size becomes $2^m - 1 = 2^d - 1$. For large d, a maliciously secure offline phase might not be able to produce a sufficient number of binomial tuples in reasonable time. We also expect a decrease of performance of the online phase for large m since the local iteration through the large tuple might become relevant. As already mentioned in the introduction, we overcome these shortcomings with arithmetic tuples, which still guarantee minimal round complexity but with a smaller overall tuple size.

5 Arithmetic Tuples: Composable One Round Protocols With Moderate Tuple Size

We now present our main technical results on the use of arithmetic tuples, which generalize both the concept Beaver triples and binomial tuples and come with the advantages already sketched in the introduction. We first provide some intuition in Section 5.1 and then present the formal construction of arithmetic tuples along with our main theorems in Sections 5.2 and 5.3; the tuple generation in the offline phase is discussed in Section 6.

5.1 Examples of Arithmetic Tuples and Highly-Level Construction Idea

Recall from the introduction that in our protocol parties receive structured randomness from the offline phase, which contains more structure than plain Beaver triples, and use this to locally compute shares of so-called *building blocks*, where a building block is roughly any polynomial in the shared inputs, computed from the publicly available data and the shared tuple entries. Once (locally) computed building blocks will be opened so that they can publicly be used by all parties. Hence, they must be constructed in such a way that they do not reveal any information on party inputs or intermediate results. Now, with the opened building blocks every party can construct the final result locally.

In the following we discuss two examples of increasing complexity to illustrate the general design principle for arithmetic tuples and their corresponding building blocks. A formal description is contained in the next subsection.

We start by looking at arithmetic tuples for computing the product of four shared inputs $[\![x_0]\!]\cdot[\![x_1]\!]\cdot[\![x_2]\!]\cdot[\![x_3]\!]$. Here, each party receives a structured 13-tuple of the form

$$\begin{aligned}([\![a_0]\!],[\![a_1]\!],[\![a_2]\!],[\![a_3]\!],[\![a_0a_1]\!],[\![a_{01}]\!],[\![a_{01}a_2]\!],[\![a_{01}a_3]\!],[\![a_2a_3]\!],&\\ [\![a_{23}]\!],[\![a_{23}a_0]\!],[\![a_{23}a_1]\!],[\![a_{01}a_2a_3+a_{23}a_0a_1-a_{01}a_{23}]\!])& \qquad (2)\end{aligned}$$

from the offline phase. The parties then proceed in the online phase according to Protocol 1.1. In this simple example the building blocks are y_{01}, y_{23}, y_{0123} and

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

Protocol 1.1: Protocol to compute the product $x_0 \cdots x_3$.

the subsequent local computation is to combine these blocks as $y_{01}y_{23} + y_{0123}$. 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 building block y_{0123}. Note that the security of this approach follows analogously to Beaver multiplication. We will present a security proof of our general online protocol later in Theorem 2 (in Section 5.3).

Now let us extend the example to multiply eight variables, i.e., to compute $x_0 \cdots x_7$. For this purpose, we replace y_{0123} by $y_{0123} - a_{0123}$ with a new mask a_{0123}. We can do this by simply adding the share $[\![a_{0123}]\!]$ locally before we open the block. Hence we can compute the masked value $x_0 \cdots x_3 - a_{0123} = y_{01}y_{23} + y_{0123} - a_{0123}$. In parallel, and analogously, we construct $x_4 \cdots x_7 - a_{4567}$ using another tuple as in Eq. (2) with the obvious notational changes in the indices. Note that we still have the same number of rounds but we can now compute

$$
\begin{aligned}
&(x_0 \cdots x_3 - a_{0123})(x_4 \cdots x_7 - a_{4567}) \\
&= x_0 \cdots x_7 - a_{4567}x_0 \cdots x_3 + a_{0123}x_4 \cdots x_7 - a_{0123}a_{4567}.
\end{aligned}
$$

In order to get the product $\prod_{j=0}^{7} x_j$ we need further building blocks to cancel out the intermediate terms like $a_{4567} \prod_{j=0}^{3} x_j$. However, we do not need to build them from scratch since we can use, e.g., y_{23} and one additional block $[\![a_{4567}x_0x_1 - a_{4567,01}]\!] = [\![a_{4567}]\!](x_0 - a_0)(x_1 - a_1) + [\![a_0a_{4567}]\!](x_1 - a_1) + [\![a_1a_{4567}]\!](x_0 - a_0) + [\![a_0a_1a_{4567} - a_{4567,01}]\!]$ for new randomness $a_{4567,01}$. The parties can locally multiply y_{23} and this new block. This again leads to more terms like $a_{4567,01}x_2x_3$ which we also need to compensate with further building blocks. Still, with a sufficient number of suitably formed building blocks we can finally compute $\prod_{j=0}^{7} x_j$. Note that we never increase the number of communication rounds since all building blocks are constructed and opened in parallel.

Of course this construction method can only be efficient if we keep the number of building blocks reasonably low. Just as in this example, we therefore use already constructed building blocks over and over again to compensate for different mixed terms. Not only do we use the masked products like $x_0x_1 - a_{01}$,

but also the new blocks of the form $a_{4567}x_0x_1 - a_{4567,01}$ with one secret shared prefactor or even products with 2 prefactors, e.g. terms like $abx_0x_1 - c$ for new randomness a, b, c. Our main technical result tells us that building blocks of these three shapes, i.e. monomials in the input variables with up to two prefactors and an additive mask, are enough to construct any product of input variables. The fact that we can use one building block *multiple* times is the reason why we can reduce the bandwidth to $\mathcal{O}(m\log(m))$ and consequentially the tuple size down to $\mathcal{O}(m\log(m)^2)$ for a product of m input variables, while keeping the communication low. In comparison, a binomial tuple has exponential tuple size for the same product. Similar results hold for arbitrary monomials and general polynomials.

Also note that this construction works in the case when we compute a product in more than two factors (e.g. $x_0\cdots x_5$ as $(x_0x_1 - a_{01})(x_2x_3 - a_{23})(x_4x_5 - a_{45})$ plus mixed terms). Allowing arbitrary partitions in our general construction gives us the flexibility to tune arithmetic tuples for smaller size or lower bandwidth. Also, the approach can be generalized to polynomials in m variables instead of just monomials.

5.2 The General Construction

We now present our formal mathematical results. Since we our results might be applicable in different setups (e.g. for computations on ciphertexts) this subsection is kept in a very formal mathematical language. To better understand the high level idea, we refer to the introduction, the previous subsection as well as Figure 1 or Example 3.

Let R be our commutative base ring, X and A disjoint finite sets of indeterminates—we will assume that all indeterminates commute to later evaluate at elements of our commutative ring. Let $P \subset R[X, A]$ be a finite set of polynomials and $L(X, A, P) = \{\sum_{r\in A\cup X\cup\{1\}, g\in R[P]} rg\}$. We call $B_{A,P,f} \subset L(X, A, P)$ a set of building blocks for a polynomial $f \in R[X]$ given A and P if $f \in R[P \cup B_{A,P,f}]$. We call a building block b additive if f is R-linear in b.[10] X will model the set of our input variables, A the set of arithmetic tuple entries—both usually only available in shared form. P contains public information available from previous rounds, e.g. $P = \{x_j - a_j : 0 \le j < m\}, x_j \in X, a_j \in A$ after an initial masking round. $L(X, A, P)$ contains the linear combinations of the X, A which can be computed locally, the public information from P can also be multiplied locally. To simplify notation we will sometimes drop the dependence on A, P if it is clear from context, i.e. simply write B_f. Note that a basis is not unique. B is called minimal if $f \notin R[P \cup (B \setminus \{b\})]$ for each $b \in B$. Since all building blocks are sent in our MPC protocol round we are mainly interested in minimal sets. Of course inclusion does not establish a total order for sets of building blocks, e.g. in the previous example Protocol 1.1 both $B_{x_0\cdots x_3} = \{x_0\cdots x_3\}$ and $B'_{x_0\cdots x_3} = \{y_{01}, y_{23}, y_{0123}\}$ are minimal—$B_{x_0\cdots x_3}$ can be constructed using

[10] The additivitiy of b depends on the choice of $g \in R[P \cup B_{A,P,f}]$ with $f(X) = g(P \cup B_{A,P,f})$. In our setup this choice will be clear from context.

a binomial tuple as in Section 4; $B'_{x_0\cdots x_3}$ is spanned by the y_{01}, y_{23}, y_{0123} from example Equation (2). We call a set of building blocks *privacy-preserving* if f can be combined in way that no information about the inputs or any intermediate results is leaked that cannot be deduced from the final result. Usually we ensure building blocks to be privacy-preserving by adding fresh randomness as a one-time pad. In particular, $B_{x_0\cdots x_3}$ and $B'_{x_0\cdots x_3}$ are clearly privacy-preserving. Obviously, $B_f \cup B_g$ is a set of building blocks for $f+g \in R[X]$ and $(f,g) \in R[X]^2$; B_f is a set of building blocks for cf for any constant $c \in R$. In particular, we can first concentrate on the building blocks for a symmetric monomial $x_0 \cdots x_{m-1}$, and later use the results to describe general monomials $x_0^{d_0} \cdots x_{m-1}^{d_{m-1}}$ or a general multivariate polynomial f.

To simplify notation we set $x_M = \prod_{i\in M} x_i$ for any $M \subset \mathbb{N}$, i.e. want to compute $x_{\{0,\dots,m-1\}}$. We further set $y_M = x_M - a_M$ for some random element a_M.[11] Our goal is to write $x_{\{0,\dots,m-1\}}$ in terms of the $y_{S_{l-1,j}}$ for a partition $\{0,\dots,m-1\} =: S_{l,0} = \dot{\bigcup}_{j\in\mathbb{Z}_{r_{l-1}}} S_{l-1,j}, r_{l-1} \in \mathbb{N}_+$ with some non-empty sets $S_{l-1,j}$, i.e. $x_M = Q(y_{S_{l-1,j}} : j \in \mathbb{Z}_{r_{l-1}})$ for some polynomial Q. In the example from Section 5.1 we had $\{0,1,2,3\} = \{0,1\} \,\dot\cup\, \{2,3\}$ and $x_{0123} = y_{01}y_{23} + a_{01}y_{23} + a_{23}y_{01} + a_{01}a_{23}$. If we can construct all summands in Q locally from building blocks then we are done. In the example, we can locally build $a_{01}y_{23}, a_{23}y_{01}, a_{01}a_{23}$.[12]

If we cannot build the summands in Q, then we have to construct each summand from smaller blocks. In general, we consider a series of refinements $\{0,\dots,m-1\} = \dot{\bigcup}_{j\in\mathbb{Z}_{r_k}} S_{k,j}$ of disjoint unions of non-empty sets for each $0 \le k \le l, 1 = r_l < r_{l-1} < \cdots < r_0 \le m$, i.e. $\forall 0 \le k < l \; \forall j \in \mathbb{Z}_{r_k} \; \exists j_0 \in \mathbb{Z}_{r_{k+1}} : S_{k,j} \subseteq S_{k+1,j_0}$. For later use we also define $I_{k,j} := \{i \in \mathbb{Z}_{r_{k-1}} : S_{k-1,i} \subset S_{k,j}\}$ for $k > 0$.

We will usually start at the ground level $\dot{\bigcup}_{j\in\mathbb{Z}_{r_k}} S_{k,j}$ where we can construct the building blocks locally and then we build higher degree products level-by-level until we get to $\prod_{j=0}^{m-1} x_j$. Depending on the size of a specific $S_{0,j}$, a building block $y_{S_{0,j}}$ (or a multiple thereof) will have degree $|S_{0,j}|$ in the x_j—the degree can vary over the different j. Recall that in principle we can build any degree $|S_{0,j}|$-term locally using binomial tuples once we know the masked inputs $x_j - a_j$ (cf. Section 4). We will use this fact later, but first let us see how many building blocks are needed to go one level up. This is the content of Lemma 1 below.

Without loss of generality we will consider $S_{k-1,i} = \{i\}$ and $S_{k,j} = \{0,\dots,r-1\}$ and hence $|I_{k,j}| = r$ for the next three lemmas. This simplifies the notation significantly. Furthermore, given a $\emptyset \neq J = \{j_0,\dots j_s\} \subset \mathbb{Z}_r$ with representatives $0 \le j_0 < \cdots j_s < r, j_{s+1} = j_0$ and a set of functions $\{f_{ij}, (i,j) \in \mathbb{Z}_r^2\}$, we define

[11] We will sometimes drop the set brackets $\{\dots\}$ in the index to simplify notation, e.g. x_{0123} instead of $x_{\{0,1,2,3\}}$.

[12] As we have seen, we only need one building block for all three terms—this optimization is discussed later.

the product $f_J \coloneqq \prod_{t=0}^{s} f_{j_t,j_{t+1}-1}$.[13] We give a specific example of this notation in the proof of Lemma 1.

Lemma 1. *Let $A_1 = \{a_{i+1,\ldots,j} : i,j \in \mathbb{Z}_r, i \neq j\} \subset A$ be a set of commuting variables. For $i,j \in \mathbb{Z}_r$ set $f_{ij} = x_i a_{i+1,\ldots,j} - a_{i,\ldots,j}$ if $i \neq j$ and $f_{ii} = x_i - a_i$.[14] The term $x_{\mathbb{Z}_r} - Q(f_{ij} : (i,j) \in \mathbb{Z}_r^2))$ is constant for $Q(f_{ij} : (i,j) \in \mathbb{Z}_r^2) \coloneqq \sum_{\emptyset \neq J \subset \mathbb{Z}_r} f_J$.*

Proof. Consider $\emptyset \neq J = \{j_0, \ldots j_s\} \subset \mathbb{Z}_r$ as above with representatives $0 \leq j_0 < \cdots j_s < r, j_{s+1} = j_0$. Then $f_J = \prod_{t=0}^{s}(x_{j_t} a_{j_t+1,\ldots,j_{t+1}-1} - a_{j_t,\ldots,j_{t+1}-1})$ by definition. E.g. the set $J = \mathbb{Z}_5$ leads to $f_J = \prod_{i=0}^{4}(x_i - a_i)$ and $J = \{2,4,5\} \subset \mathbb{Z}_6$ to $(x_2 a_3 - a_{23})(x_4 - a_4)(x_5 a_{01} - a_{501})$. We note that there are exactly r^2 different factors in products associated to non-empty sets J, since a factor is defined by its start j_t and end index j_{t+1}, i.e. 2 ordered samples from $\mathbb{Z}_r$. We show that $\prod_{j \in \mathbb{Z}_r} x_j - \sum_{\emptyset \neq J \subset \mathbb{Z}_r} f_J$ is constant.[15] Note that apart from $\prod_{j \in \mathbb{Z}_r} x_j$ each non-constant summand is of the form $x_j a_{j+1,\ldots,k-1} g$ for some specific term g and some j,k.[16] 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 $f_{j_l,k-1} g = (x_{j_l} a_{j_l+1,\ldots,k-1} - a_{j_l,\ldots,k-1})g$.[17] 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. So does $\prod_{j \in \mathbb{Z}_r} x_j$ with the degree r term in $f_{\mathbb{Z}_r}$. Thus, the remaining summands are constant. □

Remark 2. Note that only the r terms f_{ii} have a leading coefficient 1, one for each variable x_i. Another r terms are not used in products with other terms, namely the $f_{i,i-1}$. We can use this fact to combine these $f_{i,i-1}$ into a single private building block $\sum_{i=0}^{s} x_i a_{i+1,\ldots,i-1} - a_{\text{const}}$ for an $a_{\text{const}} \in A$—this fact was used to create y_{0123} in the example from Section 5.1. In particular, we can replace A_1 by $\{a_{i+1,\ldots,j} : i,j \in \mathbb{Z}_r, i \neq j, j+1\} \cup \{a_{\text{const}}\}$. Furthermore, if A contains a suitably formed $a_{\text{const}} \in A$, we can build exactly $x_{S_{k,j}}$ or $y_{S_{k,j}}$. Finally, for each $i \in \mathbb{Z}_r$ there are $r-1$ terms linear in x_i with a variable prefactor (this includes the $f_{i,i-1}$).

We want to shortly return to the general picture. Lemma 1 tells us that we can combine $x_{S_{k,j}}$ from secret linear terms in $x_{S_{k-1,i}}, i \in I_{k,j}$ (up to a constant). It also implies (cf. Remark 2) that $|I_{k,j}|$ of these smaller degree terms are again of the same shape $x_{S_{k-1,i}}$ (up to a constant)—for these terms we apply Lemma 1 again to lower the degree further. Lemma 1 applied for $S_{k-1,i} = \dot{\bigcup}_{\mu \in I_{k-1,i}} S_{k-2,\mu}$

[13] We use indices in $\mathbb{Z}_r$ because they wrap around nicely. To be more formal, let $\bar{i}$ be the unique representative of $i \in \mathbb{Z}_r$ in $\{0, \ldots, r-1\}$.

[14] We chose the notation $a_{i+1,\ldots,j}$—$a_{i,j}$ is a valid alternative.

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

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

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

results in $|I_{k-1,i}|^2$ terms linear in $x_{S_{k-2,\mu}}, \mu \in I_{k-1,j}$. We can reuse some of these terms of Lemma 1 and Lemma 2 is used to account for the remaining terms linear in $x_{S_{k-1,i}}$ but with non-trivial prefactor $a \in A$. Lemma 2 uses the same notation as Lemma 1.

Lemma 2. *Let A_1, f_{ij}, Q be as in Lemma 1. Let $\mu \in \mathbb{Z}_r$ be a fixed index and $a \in A$ some variable. Define $T_\mu := \{(i,j) \in \mathbb{Z}_r^2 : \overline{j-\mu} \leq \overline{i-\mu-1}\}$ and $S_\mu = \mathbb{Z}_r \setminus T_\mu$. Let $A_2^\mu = \{b_{ij}^\mu : (i,j) \in T_\mu\} \subset A \setminus A_1$. Define $g_{ij}^{a,\mu} = f_{ij}$ for $(i,j) \in S_\mu$, $g_{\mu\mu}^{a,\mu} = ax_\mu - b_\mu^\mu$, $g_{\mu j}^{a,\mu} = ax_\mu a_{\mu+1,\dots,j} - b_{\mu,\dots,j}^\mu$ for $j \neq \mu$ and $g_{ij}^{a,\mu} = x_i b_{i+1,\dots,j}^\mu - b_{i,\dots,j}^\mu$ for $T_\mu \setminus (\{\mu\} \times \mathbb{Z}_r)$. Then $ax_{\mathbb{Z}_r} - Q(g_{kl}^{a,\mu} : (k,l) \in \mathbb{Z}_r^2)$ is constant.*

Proof. To simplify notation set $b_{i,\dots,j}^\mu := a_{i,\dots,j}$ for $(i,j) \in S_\mu$. Then we can simply copy the proof of Lemma 1 for the variables $(ax_\mu, x_j, j \neq \mu)$ and cofficients b_* instead of a_*. Hence, $a \prod_{j \in \mathbb{Z}_r} x_j - Q(g_{ij}^{a,\mu} : (i,j) \in \mathbb{Z}_r^2)$ is constant. □

Remark 3. Observe that $r-1$ of the leading coefficients are products of two elements[18] of A, i.e. the $ax_\mu a_{\mu+1,\dots,j}$ in $g_{\mu j}^{a,\mu}$ for $j \neq \mu$. Moreover, note that r of the new $r(r+1)/2$ terms are not used in products with others, namely the $g_{i,i-1}^{a,\mu}$—in particular, $(i,i-1) \in T_\mu$. As before, these can be combined into a single building block $\sum_{i=0}^s x_i b_{i+1,\dots,i-1}^\mu - b_{\text{const}}^\mu$ for $b_{\text{const}}^\mu \in A$. One can then modify A_2 as in Remark 2.

Similar to the discussion after Remark 2, Lemma 2 comes with linear terms that have two secret prefactors and Lemma 3 below shows how we can build these terms from lower degree terms.

Lemma 3. *Let $\mu, \nu \in \mathbb{Z}_r$ be two fixed indices with $\mu \neq \nu$ and $a, b \in A$. Let A_1, A_2^μ, A_2^ν, f_{ij}, $g_{ij}^{a,\mu}$, $g_{ij}^{b,\nu}$, Q, S_μ, S_ν, T_μ, T_ν be as in Lemmas 1 and 2. Let $A_3 = \{c_{ij} : (i,j) \in T_\mu \cap T_\nu\} \subset A \setminus (A_1 \cup A_2^\mu \cup A_2^\nu)$. Let $h_{ij} = f_{ij}$ for $(i,j) \in S_\mu \cap S_\nu$, $h_{ij} = g_{ij}^\mu$ for $(i,j) \in T_\mu \setminus T_\nu$, $h_{ij} = g_{ij}^\nu$ for $(i,j) \in T_\nu \setminus T_\mu$. For $(i,j) \in T_\mu \cap T_\nu$ set: $h_{ij} = x_i c_{i+1,\dots,j} - c_{i,\dots,j}$ for $\mu \neq i \neq \nu$, $h_{\mu j} = ax_\mu b_{\mu+1,\dots,j}^\mu - c_{\mu,\dots,j}$ for $j \neq \mu$, $h_{\nu j} = bx_\nu b_{\nu+1,\dots,j}^\nu - c_{\nu,\dots,j}$ for $j \neq \nu$, and $h_{\mu\mu} = ax_\mu - c_\mu$, $h_{\nu\nu} = bx_\nu - c_\nu$. Then $abx_{\mathbb{Z}_r} - Q(h_{ij} : (i,j) \in \mathbb{Z}_r^2)$ is constant.*

Proof. Note that we can set consistently $c_{i,\dots,j} = a_{i,\dots,j}$ 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 $c_{i,\dots,j} = b_{i,\dots,j}^\mu$ 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 $c_{i,\dots,j} = b_{i,\dots,j}^\nu$ 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 $i \neq \mu \neq \nu$. The claim now follows as in Lemma 1 with variables $(ax_\mu, bx_\nu, x_j : \mu \neq j \neq \nu)$. □

Remark 4. Again r of the new terms are not used in products, i.e. $h_{i,i-1}$ and $(i,i-1) \in T_\mu \cap T_\nu$, and the usual reductions apply also to A_3. Furthermore, the number of new terms with two variable prefactors is also r, i.e. the terms $h_{\mu j}$ for $\overline{j-\nu} \leq \overline{\mu-\nu-1}$ and $h_{\nu j}$ for $\overline{j-\mu} \leq \overline{\nu-\mu-1}$.

Finally in Lemma 3 no new types of linear terms come up, i.e. we still only have prefactors 1, a or ab for some $a, b \in A$. Hence, we can iteratively use the

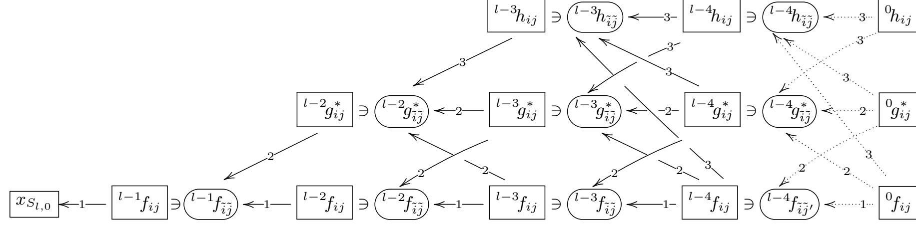


Fig. 1: The diagram illustrates which types of lower-degree polynomials are used to build higher degree terms. We added a left-upper index to denote the level, e.g. ${}^k g^*_{ij}$ denotes a linear term in $x_{S_{k,i}}$ as used in Lemma 2. Boxes, e.g. around ${}^k f_{ij}$, denote sets over $i, j \in I_{k+1,\tilde{i}}$ with $\tilde{i}$ the specific index one level higher, e.g. the index of ${}^{k+1} f_{\tilde{i}\tilde{j}}$; round framings denote single elements. The labels of the arrows refer to the numbers of the Lemmas 1 to 3. Note that in some special cases, e.g. f_{ii} not all arrows are necessary, e.g. only Lemma 1. Furthermore, the g^*_{ij} already contain some f_{ij} which will not be constructed multiple times. Analogously for h_{ij}.

three previous lemmas to reduce to any basis level $S_{0,i}$ we choose. The general construction is illustrated in Figure 1. Observe that one only needs terms of the form $f_{ij}, g^*_{ij}, h^*_{ij}$ which are by definition of the expected form with $0, 1$ or 2 prefactors.

We can now compute the number of elementary building blocks needed to compute $x_0 \cdots x_{m-1}$. Let $N^0_{S_{k,j}}$ be the number of elementary privacy-preserving building blocks linear in $x_{S_{0,i}}$ needed to compute $x_{S_{k,j}} - a_{S_{k,j}}$ for some $a_{S_{k,j}} \in A$. Let $N^1_{S_{k,j}}$ be the number of additional elementary privacy-preserving building blocks linear in $x_{S_{0,i}}$ needed to compute also $a x_{S_{k,j}} - b_{S_{k,j}}$ for some $a, b_{S_{k,j}} \in A$. Let $N^2_{S_{k,j}}$ be the number of further additional elementary privacy-preserving building blocks linear in $x_{S_{0,i}}$ needed to also compute $abx_{S_{k,j}} - c_{S_{k,j}}$ for some $a, b, c_{S_{k,j}} \in A$. From Lemmas 1 to 3 and Remarks 2 to 4 we get

$$N^0_{S_{k,j}} = \sum_{i \in I_{k,j}} N^0_{S_{k-1,i}} + (|I_{k,j}| - 1) \sum_{i \in I_{k,j}} N^1_{S_{k-1,i}} - |I_{k,j}| + 1 \tag{3}$$

$$N^1_{S_{k,j}} = \sum_{i \in I_{k,j} \setminus \{\mu\}} N^1_{S_{k-1,i}} |T_\mu \cap M_i| + (|I_{k,j}| - 1) N^2_{S_{k-1,\mu}} + N^1_{S_{k-1,\mu}} - |I_{k,j}| + 1 \tag{4}$$

$$N^2_{S_{k,j}} = \sum_{i \in I_{k,j}} N^{1+|\{i\} \cap \{\mu,\nu\}|}_{S_{k-1,i}} |M_i \cap T_\mu \cap T_\nu| - |I_{k,j}| + 1 \tag{5}$$

where $M_\iota = \{\iota\} \times I_{k,j}$ and the T_μ, T_ν are defined as in Lemma 2 using our identification $I_{k,j} \to \mathbb{Z}_r$.[19] Recall that we can remove the $|I_{k,j}|$ linear summands in each mixed term by a single building block. Hence, we get the final summand $-|I_{j,k}| + 1$ in each of the expressions.

[18] Not necessarily different.

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

Application in MPC Protocols and Asymptotic Behavior. Now, we will explain how we can use the previously discussed purely theoretical results in an MPC online phase. We remark however, that the technique might be usable in other setups to transform interactively evaluated arithmetic circuits in to circuits that can be evaluated locally.

Lemmas 1 to 3 show how to locally evaluate a term $x_{S_{l,0}}$ for some finite set $S_{l,0}$ from publicly available privacy-preserving building blocks linear in $x_{S_{0,j}}$ with $S_{l,0} = \dot{\bigcup}_{j\in\mathbb{Z}_{r_0}} S_{0,j}$. Equations (3) to (5) describe how many of these building blocks we need if we set $N^{\gamma}_{S_{0,j}}$ for all $\gamma = 0, 1, 2$ and all $j \in \mathbb{Z}_{r_0}$. In our MPC protocol we have to send the resulting $N^0_{S_{l,0}}$ building blocks plus the inital $|S_{l,0}|$ masked values send first. The Equations (3) to (5) also decribe the number of variables used in these building blocks, namely the $|A_1|, |A^{\mu}_2|, |A_3|$ from Lemmas 1 to 3 (with the size modifications discussed in the corresponding remarks). We will use *binomial tuples* to construct these building blocks. Recall from Section 4 that a term y_S can be computed with a $2^{|S|} - 1$ tuple for any finite set S; a term $dx_S - b_S$, as well as a term $dd'x_S + c_S$ for some $d, d', b_S, c_S \in A$, each need a tuple of size $2^{|S|}$ compensating for the additional prefactor(s), i.e. in the notation of Section 4 a tuple $(da^e)_{e\in E}$ or $(dd'a^e)_{e\in E}$. Hence if we replace the $N^{\gamma}_{S_{k,j}}, \gamma = 0, 1, 2$ in Equations (3) to (5) by the corresponding tuple sizes $T^{\gamma}_{S_{k,j}}$ and set $T^0_{S_{0,j}} + 1 = T^1_{S_{0,j}} = T^2_{S_{0,j}} = 2^{|S_{0,j}|}$, then $T^{\gamma}_{S_{l,0}}$ will be the tuple size needed to compute $x_{S_{l,0}}$.

We will call these tuples *arithmetic tuples* because they are used to construct the inputs of an arithmetic circuits. This arithmetic circuit is definied by Lemmas 1 to 3, e.g. in Lemma 1 we compute for each $\emptyset \neq J \subset I_{k,j}$ a multiplication gate $f_J = \prod_{t=0}^{s} f_{j_t, j_{t+1}-1}$ and then use an addition gate to add over all the J to get $x_{S_{k,j}}$. Note that the maximal number of inputs into f_J is just $|I_{k,j}|$, i.e. the number of factors in the product $\prod_{i\in I_{k,j}} y_{S_{k-1,i}}$.

Arithmetic tuples are an obvious generalization to Beaver triples or binomial tuples, which are usually only used to build a single building block. The size of an arithmetic tuple as well as the number of building blocks strongly depends on the locally evaluated circuit as the following Theorem 1 on the asymptotic complexity shows. Before we do so, we give an initiating example:

Example 1. As an example we want to compute the product of $x_0, \ldots, x_{15}$ using the partitions by $S_{k,j} = \{2^{k+1} \cdot j + i : 0 \leq i < 2^{k+1}\}$. In particular, we will get elementary building blocks 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 $y_{S_{1,j}}$ by a tuple of size $3 + 3 + (2 - 1)(4 + 4) - 2 + 1 = 13$. Moreover, for $\mu = 2j + 1$ we have $T_\mu = \{(\iota, \kappa) \in \{2j, 2j + 1\}^2 : \kappa \leq \iota\}$ and $T^1_{S_{1,j}} = 4 \cdot 1 + 1 \cdot 4 + 4 - 2 + 1 = 11$. Thus $T^0_{S_{2,j}} = 13 + 13 + (2 - 1) \cdot 22 - 2 + 1 = 47$. Next, we compute again the mixed terms with $\mu = 2j + 1, \nu = 2j$ and hence $T_\mu \cap T_\nu = \{(\mu, \mu), (\nu, \nu)\}$: $T^2_{S_{1,j}} = T^2_{S_{0,j}} + T^2_{S_{0,j}} - 2 + 1 = 7$ and $T^1_{S_{2,j}} = 11 \cdot 1 + (2 - 1) \cdot 7 + 11 - 2 + 1 = 28$. Thus, $T^0_{S_{2,j}} = 47 + 47 + (2 - 1) \cdot 56 - 2 + 1 = 149$, 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 $x_{\{0,\dots,15\}}$ has size $2^{16}-1$.

To better understand how the asymptotic behavior of the bandwidth and tuple size depends on the number of factors $|I_{k,j}|$ locally multiplied in one multiplication gate, we extend the previous example to arbitrary products of $m=\lambda b^n$ variables for some base $b \geq 1$ and $S_{k,j}=\{\lambda b^k \cdot j + i : 0 \leq i < \lambda b^k\}, 0 \leq j < b^{n-k}, 0 \leq k \leq n$, i.e. each degree b^k term splits into b building blocks of degree b^{k-1} until we reach a level of elementary building blocks of degree $\lambda \geq 1$. In Example 1 we had $b=2, n=3, \lambda=2$. 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 an arithmetic tuple 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 5. If we fix λ small, e.g. $\lambda \leq 3$, the case $b=2$ leads to a bandwith 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 4 in Appendix A). Furthermore, we remark that for a mixed number of factors going into a multiplication gate as in Equations (3) to (5) the complexity will be dominated by the largest number of factors that occurs in a significant fraction of gates. In particular, by choosing all multiplication gates to have two factors, we get for any m: bandwidth in $\mathcal{O}(m\log(m))$ and tuple size in $\mathcal{O}(m\log(m)^2)$. Finally, the complexity analysis also covers the case of a binomial tuple for $b=1$.

Polynomials in Several Variables. Up to this point we mainly discussed the computation of symmetric monomials of the form $x_0 \cdots x_{m-1}$. However, the previous results directly transfer to general monomials $x^d = x_0^{d_0} \cdots x_{m-1}^{d_{m-1}}, d=(d_0,\dots,d_{m-1})$ simply by replacing the variables x_i in the building blocks by $x_i^{d_i}$. A building block will then be linear in $\prod_{s\in S_{0,j}} x_s^{d_s}$ and this building block can still be constructed using a binomial tuple. From Section 4 we know that $T^0_{S_{0,j}} = T^1_{S_{0,j}}-1 = T^2_{S_{0,j}}-1 = \prod_{s\in S_{0,j}}(d_s+1)-1$. For the special case where $|S_{0,j}|=1$, e.g. $S_{0,j}=\{j\}$, 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 (3) to (5). If $d_j=d'$ the tuple size to compute $x_0^{d_0}\cdots x_{m-1}^{d_{m-1}}$ for $m=d'2^n$ becomes $2^{n-2}((d'+1)n^2+(3d'+7)n+4d')+1$. For details we refer to the proof of Theorem 1 in Appendix A which contains the formulas (and proof thereof) whenever $T^1_{S_{0,j}} = T^2_{S_{0,j}}$. The result shows that in the total degree $d=\sum_{j=0}^{m-1} d_j = md'$ 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 (4) and (5) and therewith the building blocks with additional prefactors from those base cases where $T^1_{S_{0,j}} = T^2_{S_{0,j}}$ is minimal, i.e. from the cases with $d_j \leq d/m$. Please note that the monomial $x_0^{d_0} \cdots x_{m-1}^{d_{m-1}}$ was also discussed in the introduction in Table 1.

Next, we recall from the beginning of this section that given sets of building blocks B_f and B_g for functions $f, g, f+g$ can be computed using the set $B_f \cup B_g$. Hence a general multivariate polynomial $f(x_0, \ldots, x_{m-1}) = \sum_{e \in E} f_e x^e$ can be constructed using $\bigcup_{e \in E: f_e \neq 0} B_{x^e}$, i.e. the building blocks B_{x^e} for its monomials. Analogously, one combines the corresponding arithmetic tuples to a tuple that allows to construct B_f. As in Remark 2 all additive building blocks from the different B_{x^e} with $f_e \neq 0$ can be combined into a single building block which further reduces the number of blocks and the corresponding to tuple. Overall we find for any multivariate polynomial f an arithmetic tuple and a set of building blocks B_f to evaluate $f(x_0, \ldots, x_{m-1})$.

5.3 Security and Composability

From Section 5.2 we know how to evaluate a polynomial $f(x_0, \ldots, x_{m-1})$ in a single round using arithmetic tuples. With our MPC protocol Π_{online} presented in Protocol 1.2, 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 $f(f_0(X_0), \ldots, f_{m-1}(X_{m-1}))$. The parties construct suitable arithmetic tuples $[\![A_j]\!], 0 \leq j < m$ (for each f_j) and $[\![A]\!]$ (for f) in the preprocessing phase and receive inputs $[\![X_j]\!]$ in the input phase. They run $\Pi_{\text{online}}.\text{Arithmetic}(X_j, f_j, \mathsf{continuation} := (f, j))$ 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 building blocks and adjust an additive building block of B_{f_j} by $[\![a_j]\!] \in [\![A]\!]$ such that after the next broadcast every party can locally compute the public values $z_j := f_j(X_j) - a_j$.

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

Remark 6. Note that our protocol is compatible with techniques used in Turbospeedz [8] and ABY2.0 [63] 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 building blocks in our protocol as all maskings $x_j - a_j$ are already accounted for.[20]

In Section 5.2 we have seen that by suitably choosing the arithmetic tuples, the corresponding building blocks and the local circuit evaluated on the building blocks, we can tradeoff bandwidth and tuple size while keeping the round complexity minimal. The multi-round feature adds additional flexibility to our online protocols Π_{online}. In particular, it allows to increase the round complexity slightly to prevent possible performance bottlenecks in bandwidth and tuple size.

Example 2. The 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 size 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. Arithmetic tuples 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 arithmetic tuple 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 $4 \cdot 7 + 9 = 37$.

Figure 2 further illustrates this tradeoff between round complexity, bandwidth and tuple size. 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 (3) to determine the best performing arithmetic tuple solution before the actual computation starts. Furthermore, ideal solutions for classical and regularly used setups can be hard-coded.

Security. Our Protocol 1.2 Π_{online} is secure and composable in the sense of *universal composability* (UC) [17], i.e. it can be combined with other MPC protocols and itself, while still giving the same guarantees as an idealized protocol

[20] Using only Beaver multiplication (or binomial tuples), this would exactly correspond to the complexity of ABY2.0 or Turbospeedz, where we open one element per multiplication instead of two elements as in classical protocols.

[21] Please see Example 3 for a vizualization of the arithmetic tuple construction in the case $m = 12$.

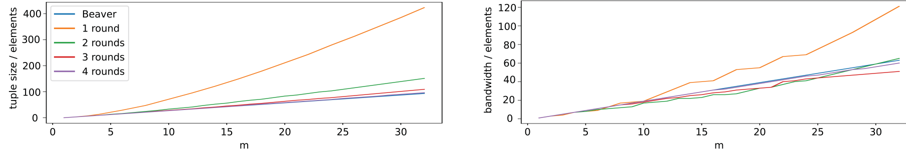


Fig. 2: Multi-round example to evaluate a product of m factors with arithmetic tuples with optimal tuple size.

(a so-called functionality; $\mathcal{F}_{\text{online}}$ in our case). For the sake of completeness we include this and other ideal functionalities in Appendix B. However, apart from the Arithmetic subprotocol there are no significant changes to [29,49].

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 building block are masked with a new random element from $[\![A]\!]$, i.e. they are encrypted with a one-time pad and hence information-theoretically secure. The final additive building block contains the result minus a public constant (constructed from the other uniformly random looking building blocks). 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 1.9; recall that we now consider R to be a finite field). In particular, Π_{CheckMAC} is chosen identical to the classical MAC check in [29]. Formally we have the following security result:

Π_{online}

Initialize. The parties call $\mathcal{F}_{[\![\cdot]\!]}$ to get a sufficient number of (shared) random data (including tuples, MAC key shares).
Input. On input a finit set X of inputs of party P_i, the parties invoke $\mathcal{F}_{[\![\cdot]\!]}$.Input. They receive $[\![x]\!]$ for each $x \in X$.
Arithmetic. On input $((z_1, \dots, z_m), f, \texttt{continuation})$, the parties do the following:

1. Get a tuple $[\![(a_1, \dots, a_k)]\!]$ from $\mathcal{F}_{[\![\cdot]\!]}$.Tuple. If `continuation` is another polynomial g and an index j (i.e. the result of f is used as the jth input to g), $\mathcal{F}_{[\![\cdot]\!]}$.Tuple also returns $[\![b_j]\!]$—the jth entry of the tuple for g.
2. Each z_j is either a share $[\![x_j]\!]$ or a previously opened value $y_j = x_j - a_j$; the parties compute and open $[\![y_j]\!] = [\![x_j]\!] - [\![a_j]\!]$ for each of the shares.
3. The parties compute the building blocks $[\![B_f]\!]$. If `continuation` = share, open all but the additive building block; if `continuation` = open, open all building blocks; otherwise subtract $[\![b_j]\!]$ from the additive building block and open all building block.
4. Compute $[\![f(x_1, \dots, x_m)]\!]$, $f(x_1, \dots, x_m)$, or $f(x_1, \dots, x_m) - b_j$ from the building blocks (depending on the value of `continuation`).

Check. Call Π_{CheckMAC} for all values opened up until now.

Protocol 1.2: Online Protocol.

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 [29,30]. Both construct a suitable simulator, e.g. [29, Fig. 22]. The only difference for a simulator in our protocol is in arithmetic operations that will be opened (i.e. calls to Π_{online}.Arithmetic with $\mathsf{continuation} = \text{open}$). Recall that the simulator works on *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 final additive building block b_{add} of one (simulated) honest party P_i by $\Delta = y - z$, i.e. $[b_{\text{add}}]_i \to [b_{\text{add}}]_i + \Delta$. Since the simulator also knows the MAC key α, it can change $[\alpha b_{\text{add}}]_i \to [\alpha b_{\text{add}}]_i + \alpha\Delta$. 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. □

Actively-Secure Offline Phase. In order to build a complete actively-secure MPC protocol, the new correlated randomness, i.e. arithmetic tuples, has to be produced in an actively secure way. In Section 6 we therefore present different and partly new solutions for an actively secure tuple generation.

6 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 [29,30] utilizes BGV [15]), oblivious transfer (as used in MASCOT [48]), or linear homomorphic encryption (LHE; as used in Overdrive [49]). These mostly focus on generating Beaver triples. We present two ways to generate arithmetic tuples 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 arithmetic tuple generation in Appendix C.3.

6.1 Plugin Approach

We can use the structured randomness, i.e. Beaver triples, generated by existing protocols to construct arithmetic tuples in an actively secure offline phase. Each entry of an arithmetic tuple is a share of some polynomial in random variables, i.e. the additive masks for the building blocks. 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 [1] or [47] are available and that further improvements of these offline phases will be directly available to the production of arithmetic tuples as well.

Remark 7. Please also note that an arithmetic tuple does not have to contain complex correlated randomness of high degree. In fact, as we have seen in Section 5.2, for specific arithmetic tuples we only need building blocks with up to two prefactors linear in a monomial of some degree d. But we only need randomness of degree $d+2$ for these building blocks of degree d, e.g., $d = 2$ to compute building blocks with 2 prefactors such as abx_1x_2—in this example the tuple contains $[\![aba_1]\!], [\![aba_2]\!], [\![aba_1a_2]\!]$.

6.2 Linear Homomorphic Encryption

We now propose a new multi-round offline protocol for generating arithmetic tuples based on linear homomorphic encryption. We construct a protocol similar to Overdrive's multiplication protocol [49] but which extends it to multiple rounds (to compute higher order randomness). In contrast to Section 6.1, 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 $\mathrm{Enc}_{\mathrm{pk}_j}([a]_j)$ to P_i and then receives back a term $\mathrm{Enc}_{\mathrm{pk}_j}([a]_j)\,[b]_i + \mathrm{Enc}'_{\mathrm{pk}_j}(r_{ji})$ which they decrypt to $[a_j]\,[b]_i + r_{ji}$. Here, Enc' has larger noise than Enc (cf. [49] for further details).

Our adaption removes the second step. Instead of returning $\mathrm{Enc}_{\mathrm{pk}_j}([a]_j\,[b]_i + r_{ij})$ 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 1.3, 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 [49]. 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 $\text{Enc}_{\text{pk}_j}([c]_j)$ for each $1 \le j \le n$, i.e. we can iterate protocol $\Pi_{\text{LHE-rd}}$, as seen in Protocol 1.4. 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 $\text{Enc}_{\text{pk}_j}([a]_j)$ for each $1 \le j \le n$, $[b]_i$. Each P_i does:

1. For each $j \neq i$ sample r_{ij}. Set $r_{ii} := -\sum_{j \neq i} r_{ij}$.
2. Broadcast $\hat{d}_{ji} := \text{Enc}_{\text{pk}_j}([a]_j)\,[b]_i - \text{Enc}'_{\text{pk}_j}(r_{ij})$ 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 $\text{Enc}_{\text{pk}_j}([c]_j) = \sum_{k=1}^{n} \hat{d}_{jk}$ for all $1 \le j \le n$.

Protocol 1.3: Multiplication using an LHE scheme.

We have included several variations of our technique in Appendix C.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 7, 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.

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 $0 \le j < m$ 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 $0 \le j < m$ and proof that it is well-formed with the zero-knowledge proof $\mathcal{F}^{S}_{\text{ZKPoP}}$ from [49].
2. Compute $[f(x_0, \ldots, x_{m-1})]_i$ in $\lceil \log d \rceil$ rounds of $\Pi_{\text{LHE-rd}}$.

Protocol 1.4: Triple production with multi-round LHE.

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 C.1.

Remark 8. Our MP-SPDZ implementation [65] 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 Section 6). As our focus in on applications where the offline phase is not time-critical, we leave benchmarking (and optimizing) the offline phase to future work.

7 Applications

Our arithmetic tuple approach is clearly relevant for applications where arithmetic circuits, polynomials and 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 arithmetic tuples can be used and in Section 8 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 [30] and similar protocols) to compute polynomials in $\mathbb{F}_p$. Most applications that perform operations on integer-valued data can benefit from arithmetic tuples 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. Arithmetic tuples of small polynomial degree as discussed in Theorem 1 and Remark 7 could be a good fit for these applications. A detailed discussion on polynomial evaluations over $\mathbb{R}$ with arithmetic tuples is, however, left to future work. As we demonstrate next, there are applications where our arithmetic tuples 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. [18,19,60]). They often use (not maliciously secure) techniques as in [3,28,54] 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 arithmetic tuples to compute prefix products can be found in Appendix D.

To give a concrete example, we briefly look at a standard approach for equality and less-than tests [18,60], 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 1.5 and 1.6).

Π_{EQ}

1. Let $[\![r_j]\!]_i$ and c_j be the inputs (bit-decomposed; index $0 \le j < k$ 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 $0 \le j < k$.
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 1.5: Bit-wise equality test protocol [60].

Π_{LT}

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

Protocol 1.6: Bit-wise less-than protocol [28].

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 1.5). We see that this protocol involves two operations with communication: (i) a masked opening (not pictured in Protocol 1.5) 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 $[\![x \le y]\!]_i$) can be done as in [18]. 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 1.6). The version shown here is based on the classical less-than protocol in [28] and turns out to be more efficient than the ones of [18,66] (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 1.6, 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.

Results for sample applications (auctions, e-voting, neural networks) are discussed next.

8 Implementation and Evaluation

To illustrate the practicality of our approach, we have implemented the online phase in the MP-SPDZ framework [47] and run several benchmarks. Our implementation is available at [65]. These first benchmarks show that we can outperform the standard Beaver triple-based approach 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) and simulated different network settings for $n = 2$ parties with standard Linux tools (see Appendix E 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 [61]). The trends in all benchmarks show that our approach will perform even better in such a setting.

Our MP-SPDZ implementation 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 Section 6). As our focus in on applications where the offline phase is not time-critical, we leave benchmarking (and optimizing) the offline phase to future work.

We added elementary operations for powers and products to MP-SPDZ. The former are based on binomial tuples, as a special case of arithmetic tuples, and the latter on arithmetic tuples that minimize the tuple size (see Equation (3) and the case $b = 2$ in Theorem 1). Both operations are also available as a prefix variant. All operations support MP-SPDZ's parallelism model: arbitrarily many operations of the same type can be combined and are executed in one step (reducing the number of communication rounds).

Note that in most of our benchmarks, one could not expect binomial tuples to work on their own, as some examples contain products of 32 or 64 elements—the resulting tuple sizes, 2^{32} and 2^{64}, and local computation times are beyond

practical. Therefore, we do not compare our approach to binomial tuples but to classical approaches (Beaver multiplication).

Next, we describe our test applications and discuss the results of our benchmarks.

Polynomial Evaluation. As an example for 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 arithmetic tuples. We compare this to the same computation with standard (Beaver triple-based) tools included in MP-SPDZ.

Figure 3 shows 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).

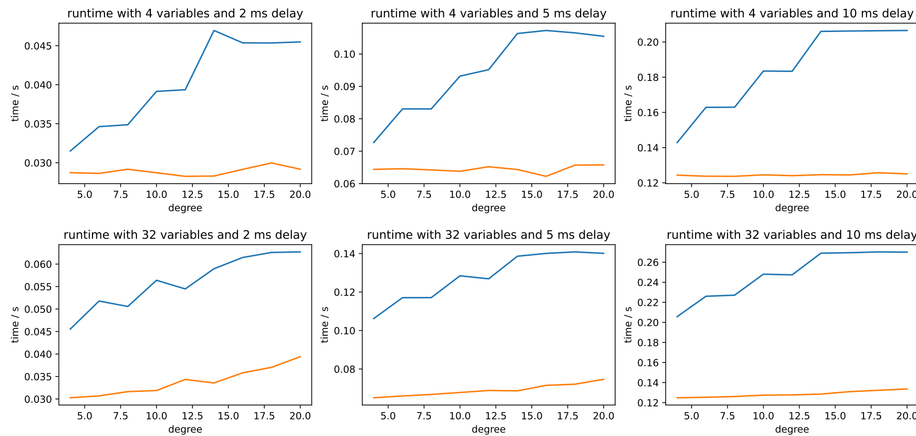
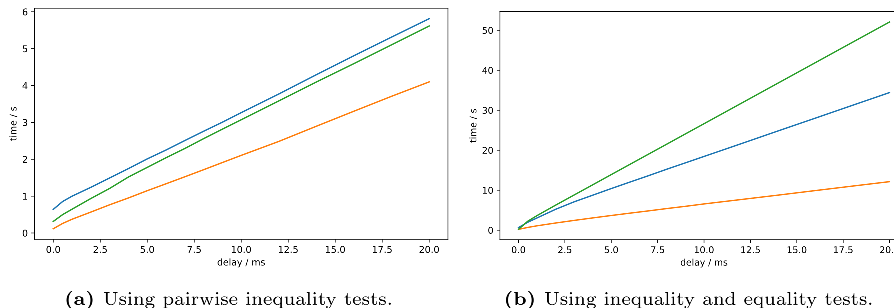


Fig. 3: Benchmark for polynomial evaluation (blue: default MP-SPDZ implementation, orange: ours).

Rankings. For auctions (or e-voting), one can 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 [51]). Note that e-voting (or auctions) might require additional security properties (e.g. public verifiability or identifiable abort) that is not directly covered by our protocol. However, this can be achieved with extension to SPDZ

that have these properties [5,6,25]. 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 [51]. 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 $(x_i = x_j)_{0<i<j}$. These operations can be implemented with our tuples as described in Section 7. We tested both approaches and compare them to the respective default implementation in MP-SPDZ (based on the protocols with logarithmic complexity in [18]; with and without edabits [32] to speed up the comparison). We compute rankings of $m = 40$ items (bids or candidates). The benchmark results in Fig. 4 show that our new approach is faster than the others.



(a) Using pairwise inequality tests.

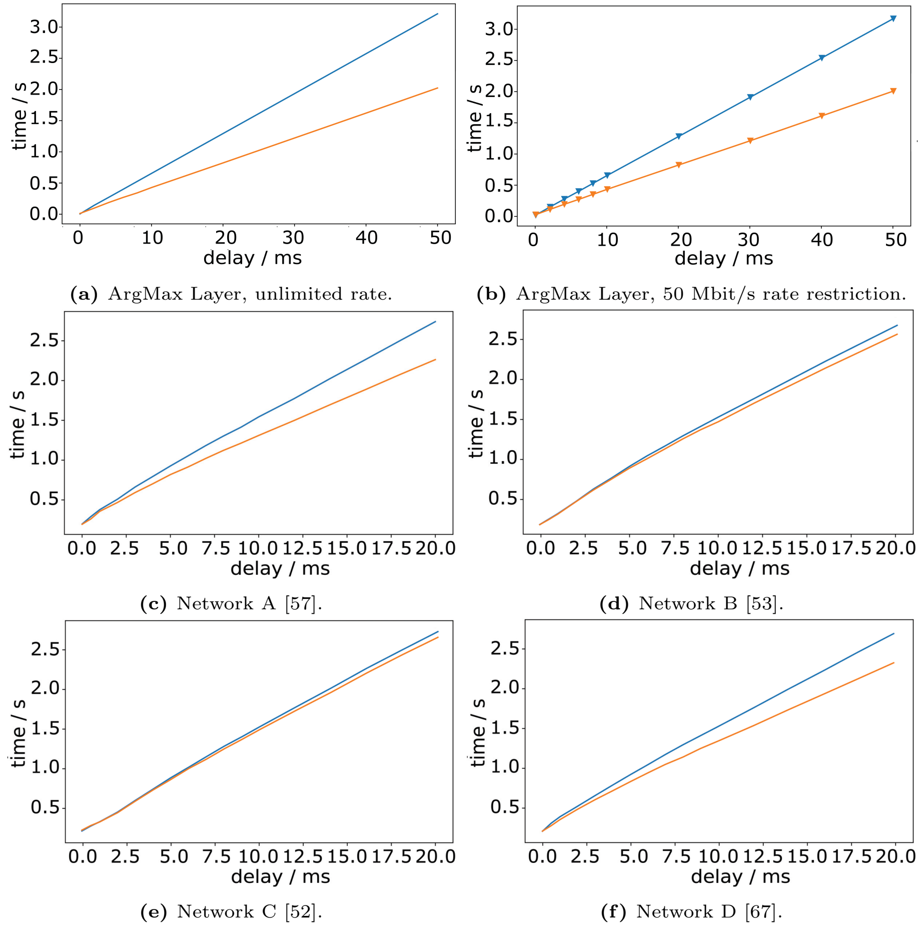
(b) Using inequality and equality tests.

Fig. 4: Benchmark for rankings (blue: default MP-SPDZ implementation, orange: ours, green: MP-SPDZ with edabits [32]).

Remark 9. SPDZ is an protocol originally designed for an arithmetic circuit evaluation and not for comparisions. In particular, there other MPC approach better suited for some types of comparisions. 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 comparisions also to SPDZ, although there are other competitive MPC protocols.

Neural Networks. Among others, MP-SPDZ [47] contains examples of deep neural networks. For our benchmarks, we ran the networks labeled A [57], B [53], C [52], and D [67] (as in [50,72]). Each of these networks has a final ArgMax layer. Replacing *only this single layer* with our arithmetic tuple-based comparison (see Section 7) can already have a noticeable impact on the overall runtime of the network, as can be seen in Fig. 5. We also remark that a bandwidth rate restriction does not affect the performance and hence the theoretical bandwidth

overhead of the arithmetic tuples approach is negligible in our example. Similiar results hold for the other evaluations.[22]



(a) ArgMax Layer, unlimited rate.

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

(c) Network A [57].

(d) Network B [53].

(e) Network C [52].

(f) Network D [67].

Fig. 5: Benchmarks for evaluating neural networks A–D included in MP-SPDZ [47] (cf. [67]; blue: default MP-SPDZ implementation, orange: ours).

9 Conclusion and Future Work

In summary, arithmetic tuples provide a new tool to evaluate polynomials (or arithmetic circuits) in a minimal number of communication rounds with moderate tuple size and bandwidth, i.e. we provide lower round complexity than

[22] Appendix E contains further evaluations for other bandwidth restrictions.

Beaver multiplication and better tuple size than with binomial tuples. Additionally, the framework is flexible and allows protocols to trade lower bandwidth for larger tuple sizes (and vice versa). It supports multi-round evaluations that can achieve lower bandwidth and smaller round complexity than Beaver multiplication for only a slightly larger tuple size. We show our performance advantage over SPDZ in the online phase for classical sample applications like the evaluation of multivariate polynomials or comparisons.

In Section 5.2, we describe in detail how the different performance measures for arithmetic tuples relate for arbitrary monomials in the input variables. This could be the basis for a new generation of MPC compilers that automatically transforms a function to be evaluated into an arithmetic circuit that is either evaluated as a whole with our technique or arithmetic tuples are used round-wise to evaluate the subcircuits.

Furthermore, our theoretical results from Section 5.2 have the potential to be applied outside of secret-sharing based MPC, e.g. to evaluate polynomials with encrypted inputs when we replace secret-sharing and openings in our protocol with ciphertexts and decryption. This could be used e.g. in an MPC offline phase to compute correlated randomness of a very high degree with linear/levelled homomorphic encryption in a small (constant) number of rounds.

While we generally work on finite fields or rings in this paper, our case study on comparisons also applies to applications with real-valued data. It would, however, be interesting future work to investigate challenges related the evaluation of real-valued polynomials with fixed-point arithmetic and to optimize arithmetic tuples for such applications.

References

1. Aly, A., Coenen, B., Cong, K., Koch, K., Keller, M., Rotaru, D., Scherer, O., Scholl, P., Smart, N.P., Tanguy, T., Wood, T.: SCALE-MAMBA MPC software. GitHub, `https://github.com/KULeuven-COSIC/SCALE-MAMBA`
2. Applebaum, B., Kachlon, E., Patra, A.: The round complexity of perfect MPC with active security and optimal resiliency. In: FOCS 2020. pp. 1277–1284. IEEE (2020)
3. Bar-Ilan, J., Beaver, D.: Non-cryptographic fault-tolerant computing in constant number of rounds of interaction. In: PODC 1989. pp. 201–209. ACM (1989)
4. Baum, C., Cozzo, D., Smart, N.P.: Using TopGear in Overdrive: A more efficient ZKPoK for SPDZ. In: SAC 2019. pp. 274–302. Springer (2020)
5. Baum, C., Damgård, I., Orlandi, C.: Publicly Auditable Secure Multi-Party Computation. In: SCN 2014. pp. 175–196. Springer (2014)
6. Baum, C., Orsini, E., Scholl, P.: Efficient Secure Multiparty Computation with Identifiable Abort. In: TCC 2016-B. pp. 461–490 (2016)
7. Beaver, D.: Efficient multiparty protocols using circuit randomization. In: CRYPTO '91. pp. 420–432. Springer (1992)
8. Ben-Efraim, A., Nielsen, M., Omri, E.: Turbospeedz: Double your online SPDZ! improving SPDZ using function dependent preprocessing. In: ACNS 2019. pp. 530–549. Springer (2019)

9. Bendlin, R., Damgård, I., Orlandi, C., Zakarias, S.: Semi-homomorphic Encryption and Multiparty Computation. In: EUROCRYPT 2011. pp. 169–188. Springer (2011)
10. Bitan, D., Dolev, S.: Optimal-Round Preprocessing-MPC via Polynomial Representation and Distributed Random Matrix (extended abstract). IACR Cryptol. ePrint Arch. **2019**, 1024 (2019)
11. Bogdanov, D., Niitsoo, M., Toft, T., Willemson, J.: High-performance secure multiparty computation for data mining applications. Int. J. Inf. Sec. **11**(6), 403–418 (2012)
12. Boneh, D., Boyle, E., Corrigan-Gibbs, H., Gilboa, N., Ishai, Y.: Zero-knowledge proofs on secret-shared data via fully linear pcps. In: CRYPTO 2019. pp. 67–97. Springer (2019)
13. Boura, C., Chillotti, I., Gama, N., Jetchev, D., Peceny, S., Petric, A.: High-precision privacy-preserving real-valued function evaluation. In: FC 2018. pp. 183–202. Springer (2018)
14. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Scholl, P.: Efficient Pseudorandom Correlation Generators: Silent OT Extension and More. In: CRYPTO 2019. pp. 489–518. Springer (2019)
15. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (leveled) fully homomorphic encryption without bootstrapping. In: ITCS 2012. pp. 309–325. ACM (2012)
16. Bultel, X., Das, M.L., Gajera, H., Gérault, D., Giraud, M., Lafourcade, P.: Verifiable Private Polynomial Evaluation. In: ProvSec 2017. pp. 487–506. Springer (2017)
17. Canetti, R.: Universally Composable Security: A New Paradigm for Cryptographic Protocols. In: FOCS 2001. pp. 136–145. IEEE Computer Society (2001)
18. Catrina, O., de Hoogh, S.: Improved Primitives for Secure Multiparty Integer Computation. In: SCN 2010. pp. 182–199. Springer (2010)
19. Catrina, O., Saxena, A.: Secure Computation with Fixed-Point Numbers. In: FC 2010. pp. 35–50. Springer (2010)
20. Chen, H., Kim, M., Razenshteyn, I.P., Rotaru, D., Song, Y., Wagh, S.: Maliciously Secure Matrix Multiplication with Applications to Private Deep Learning. In: ASIACRYPT 2020. pp. 31–59. Springer (2020)
21. Cho, H., Wu, D., Berger, B.: Secure genome-wide association analysis using multiparty computation. Nat. Biotechnol. **36**(6), 547–551 (2018)
22. Corrigan-Gibbs, H., Boneh, D.: Prio: Private, robust, and scalable computation of aggregate statistics. In: NSDI 2017. pp. 259–282. USENIX Association (2017)
23. Cramer, R., Damgård, I.: Secure Distributed Linear Algebra in a Constant Number of Rounds. In: CRYPTO 2001. pp. 119–136. Springer (2001)
24. Cui, H., Zhang, K., Chen, Y., Liu, Z., Yu, Y.: Mpc-in-multi-heads: A multi-prover zero-knowledge proof system - (or: How to jointly prove any NP statements in ZK). In: ESORICS. pp. 332–351. Springer (2021)
25. Cunningham, R.K., Fuller, B., Yakoubov, S.: Catching MPC Cheaters: Identification and Openability. In: ICITS 2017. pp. 110–134. Springer (2017)
26. Dachman-Soled, D., Malkin, T., Raykova, M., Yung, M.: Secure Efficient Multiparty Computing of Multivariate Polynomials and Applications. In: ACNS 2011. pp. 130–146 (2011)
27. Dahl, M.: Cryptography and machine learning (2017), `https://mortendahl.github.io`
28. Damgård, I., Fitzi, M., Kiltz, E., Nielsen, J.B., Toft, T.: Unconditionally Secure Constant-Rounds Multi-party Computation for Equality, Comparison, Bits and Exponentiation. In: TCC 2006. pp. 285–304. Springer (2006)

29. Damgård, I., Keller, M., Larraia, E., Pastro, V., Scholl, P., Smart, N.P.: Practical covertly secure MPC for dishonest majority – or: Breaking the SPDZ limits. In: ESORICS 2013. pp. 1–18. Springer (2013)
30. Damgård, I., Pastro, V., Smart, N., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: CRYPTO 2012. pp. 643–662. Springer (2012)
31. Eriguchi, R., Ohara, K., Yamada, S., Nuida, K.: Non-interactive Secure Multiparty Computation for Symmetric Functions, Revisited: More Efficient Constructions and Extensions. In: CRYPTO 2021. pp. 305–334. Springer (2021)
32. Escudero, D., Ghosh, S., Keller, M., Rachuri, R., Scholl, P.: Improved Primitives for MPC over Mixed Arithmetic-Binary Circuits. In: CRYPTO 2020. pp. 823–852. Springer (2020)
33. Falamas, D., Marton, K.: Performance impact analysis of rounds and amounts of communication in secure multiparty computation based on secret sharing. In: RoEduNet 2019. pp. 1–6. IEEE (2019)
34. Franklin, M.K., Mohassel, P.: Efficient and Secure Evaluation of Multivariate Polynomials and Applications. In: ACNS 2010. pp. 236–254 (2010)
35. Gavin, G., Minier, M.: Oblivious Multi-variate Polynomial Evaluation. In: INDOCRYPT 2009. pp. 430–442. Springer (2009)
36. Gentry, C.: A Fully Homomorphic Encryption Scheme. Ph.D. thesis, Stanford University (2009)
37. Gentry, C., Halevi, S., Smart, N.P.: Fully homomorphic encryption with polylog overhead. In: EUROCRYPT 2012. pp. 465–482. Springer (2012)
38. Ghodosi, H., Pieprzyk, J., Steinfeld, R.: Multi-party computation with conversion of secret sharing. Des. Codes Cryptogr. **62**(3), 259–272 (2012)
39. Gjøsteen, K., Haines, T., Müller, J., Rønne, P.B., Silde, T.: Verifiable Decryption in the Head. IACR Cryptol. ePrint Arch. p. 558 (2021)
40. Gordon, S.D., Malkin, T., Rosulek, M., Wee, H.: Multi-party Computation of Polynomials and Branching Programs without Simultaneous Interaction. In: EUROCRYPT 2013. pp. 575–591. Springer (2013)
41. Halevi, S., Hazay, C., Polychroniadou, A., Venkitasubramaniam, M.: Round-Optimal Secure Multi-party Computation. J. Cryptol. **34**(3), 19 (2021)
42. Halevi, S., Ishai, Y., Jain, A., Komargodski, I., Sahai, A., Yogev, E.: Non-Interactive Multiparty Computation Without Correlated Randomness. In: ASIACRYPT 2017. pp. 181–211. Springer (2017)
43. Halevi, S., Ishai, Y., Jain, A., Kushilevitz, E., Rabin, T.: Secure Multiparty Computation with General Interaction Patterns. In: ITCS 2016. pp. 157–168. ACM (2016)
44. Ishai, Y., Kushilevitz, E.: Randomizing Polynomials: A New Representation with Applications to Round-Efficient Secure Computation. In: FOCS 2000. pp. 294–304. IEEE Computer Society (2000)
45. Karl, R., Burchfield, T., Takeshita, J., Jung, T.: Non-Interactive MPC with Trusted Hardware Secure Against Residual Function Attacks. In: SecureComm 2019. pp. 425–439. Springer (2019)
46. Karl, R., Takeshita, J., Mohammed, A., Striegel, A., Jung, T.: Cryptonomial: A Framework for Private Time-Series Polynomial Calculations. In: SecureComm 2021. pp. 332–351. Springer (2021)
47. Keller, M.: MP-SPDZ: A Versatile Framework for Multi-Party Computation. In: CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020. pp. 1575–1590. ACM (2020)

48. Keller, M., Orsini, E., Scholl, P.: MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In: CCS 2016. pp. 830–842. ACM (2016)
49. Keller, M., Pastro, V., Rotaru, D.: Overdrive: Making SPDZ great again. In: EUROCRYPT 2018. pp. 158–189. Springer (2018)
50. Keller, M., Sun, K.: Secure Quantized Training for Deep Learning. CoRR **abs/2107.00501** (2021)
51. Küsters, R., Liedtke, J., Müller, J., Rausch, D., Vogt, A.: Ordinos: A Verifiable Tally-Hiding E-Voting System. In: EuroS&P 2020. pp. 216–235. IEEE (2020)
52. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE **86**(11), 2278–2324 (1998)
53. Liu, J., Juuti, M., Lu, Y., Asokan, N.: Oblivious Neural Network Predictions via MiniONN Transformations. In: CCS 2017. pp. 619–631. ACM (2017)
54. Lu, D., Yu, A., Kate, A., Maji, H.K.: Polymath: Low-Latency MPC via Secure Polynomial Evaluations and Its Applications. Proc. Priv. Enhancing Technol. **2022**(1), 396–416 (2022)
55. Lyubashevsky, V., Nguyen, N.K., Seiler, G.: Shorter Lattice-Based Zero-Knowledge Proofs via One-Time Commitments. In: PKC 2021. pp. 215–241. Springer (2021)
56. Mohassel, P., Franklin, M.K.: Efficient Polynomial Operations in the Shared-Coefficients Setting. In: PKC 2006. pp. 44–57. Springer (2006)
57. Mohassel, P., Zhang, Y.: SecureML: A System for Scalable Privacy-Preserving Machine Learning. In: SP 2017. pp. 19–38. IEEE Computer Society (2017)
58. Naehrig, M., Lauter, K., Vaikuntanathan, V.: Can homomorphic encryption be practical? In: CCSW 2011. pp. 113–124. ACM (2011)
59. Naor, M., Pinkas, B.: Oblivious Transfer and Polynomial Evaluation. In: STOC 1999. pp. 245–254. ACM (1999)
60. Nishide, T., Ohta, K.: Multiparty Computation for Interval, Equality, and Comparison Without Bit-Decomposition Protocol. In: PKC 2007. pp. 343–360. Springer (2007)
61. Ohata, S., Nuida, K.: Communication-Efficient (Client-Aided) Secure Two-Party Protocols and Its Application. In: FC 2020. pp. 369–385. Springer (2020)
62. Ore, Ø.: Über höhere kongruenzen. Norsk Mat. Forenings Skrifter **1**(7), 15 (1922)
63. Patra, A., Schneider, T., Suresh, A., Yalame, H.: ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation. In: USENIX Security 2021. pp. 2165–2182. USENIX Association (2021)
64. Reisert, P., Rivinius, M., Krips, T., Hasler, S., Küsters, R.: Actively Secure Polynomial Evaluation from Shared Polynomial Encodings (Full Version). Crypt. ePrint 2024/1435 (2024)
65. Reisert, P., Rivinius, M., Krips, T., Hasler, S., Küsters, R.: Implementation to *Actively Secure Polynomial Evaluation from Shared Polynomial Encodings* (2024), Website of the Insitute of Information Security Stuttgart
66. Reistad, T.I.: Multiparty Comparison - An Improved Multiparty Protocol for Comparison of Secret-shared Values. In: SECRYPT 2009. pp. 325–330. INSTICC Press (2009)
67. Riazi, M.S., Weinert, C., Tkachenko, O., Songhori, E.M., Schneider, T., Koushanfar, F.: Chameleon: A Hybrid Secure Computation Framework for Machine Learning Applications. In: AsiaCCS 2018. pp. 707–721. ACM (2018)
68. Sahraei, S., Avestimehr, A.S.: INTERPOL: Information Theoretically Verifiable Polynomial Evaluation. In: ISIT 2019. pp. 1112–1116. IEEE (2019)
69. Silde, T.: Verifiable Decryption for BGV. IACR Cryptol. ePrint Arch. p. 1693 (2021)

70. Tassa, T., Jarrous, A., Ben-Ya'akov, Y.: Oblivious evaluation of multivariate polynomials. J. Math. Cryptol. **7**(1), 1–29 (2013)
71. Vaikuntanathan, V.: Secure Computation and PPML: Progress and Challenges (2021), PPML 3rd Privacy-Preserving Machine Learning Workshop 2021
72. Wagh, S., Gupta, D., Chandran, N.: 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 5.2. We will also include tuple-size-optimized arithmetic tuples for the product of $m \leq 2$ elements in Table 2.

Proof of Theorem 1. We will only need the case $N^1_{S_0} = N^2_{S_0} = 2N^{1/2}_{S_0}$ to treat bandwidth and tuples size simultaneously. Hence, we slightly restrict our setup to this case from now on. In fact using generalized tuples we can construct the elementary building blocks with tuple sizes $T^0_{S_{0,j}} = 2^\lambda - 1, T^1_{S_{0,j}} = T^2_{S_{0,j}} = 2^\lambda$. For the number of building blocks we trivially have $T^0_{S_{0,j}} = T^1_{S_{0,j}} = T^2_{S_{0,j}} = 1$. Now choose $\mu = bj + b - 1$ to get $T_\mu = \{(\iota, \kappa) \in \{bj, \dots, bj + b - 1\}^2 : \kappa \leq \iota\}$ and $\nu = bj + \lfloor \frac{b-1}{2} \rfloor$ to get $T_\mu \cap T_\nu = \{(\iota, \kappa) \in \{bj, \dots, bj + b - 1\}^2 : (\kappa \leq \iota \leq \nu) \vee (\nu + 1 \leq \iota, \kappa \leq \iota - \nu - 1)\}$. In particular, $|T_\mu \cap M_{bj+i}| = i + 1$ and $|T_\mu \cap T_\nu \cap M_{bj+i}| = i + 1$ for $0 \leq i \leq \nu - bj$, $|T_\mu \cap T_\nu \cap M_{i+\nu+1}| = i + 1$ for $0 \leq i < \mu - \nu$. Hence we have

$$N^0_{S_k} = bN^0_{S_{k-1}} + (b-1)(bN^1_{S_{k-1}} - 1) \tag{6}$$

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

$$N^2_{S_k} = uN^1_{S_{k-1}} + b(N^2_{S_{k-1}} - 1) + 1 \tag{8}$$

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 $S_k := S_{k,j}$. 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

$$b(N^2_{S_k} - 1) = ((b+1)N^{1/2}_{S_0} - 1)\left(\frac{b^2+1}{2}\right)^k + \frac{(b^2-1)(N^{1/2}_{S_0} - 1)}{2}\left(\frac{b+1}{2}\right)^{k-1} \tag{9}$$

$$bN^1_{S_k} = 2\left((b+1)N^{1/2}_{S_0} - 1\right)\left(\frac{b^2+1}{2}\right)^k - 2(N^{1/2}_{S_0} - 1)\left(\frac{b+1}{2}\right)^k \tag{10}$$

$$(b-1)(N^0_{S_k} - 1) = 4b(N^2_{S_k} - 1) + ((b-1)(N^0_{S_0} - 1) - 4b(2N^{1/2}_{S_0} - 1))b^k \tag{11}$$

Note that it is enough to proof these formulas for $N^0_{S_0} = T^0_{S_0} = 2^\lambda - 1, N^1_{S_0} = T^1_{S_0} = N^2_{S_0} = T^2_{S_0} = 2^\lambda$ to get the estimates on the tuple size, and for $N^0_{S_0} = N^1_{S_0} = N^2_{S_0} = 1$ for the bandwidth estimate. Recall that the bandwidth is just $N^0_{S_n} + m$, which accounts for the building blocks 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 (9), (10), (11) by induction on k with $k = 0$:[23] $b(N^2_{S_0} - 1) = (b+1)N^{1/2}_{S_0} - 1 + (b-1)(N^{1/2}_{S_0} - 1)$, $bN^1_{S_0} = 2((b+1)N^{1/2}_{S_0} - 1) - 2(N^{1/2}_{S_0} - 1)$ and $(b-1)(N^0_{S_0}) = 4b(N^2_{S_0} - 1) + (b-1)(N^0_{S_0} - 1) - 4b(2N^{1/2}_{S_0} - 1)$. Hence, we get

$$
\begin{aligned}
&b(N^2_{S_{k+1}} - 1) \\
&= u \cdot 2\left((b+1)N^{1/2}_{S_0} - 1\right)\left(\frac{b^2+1}{2}\right)^k - u \cdot 2(N^{1/2}_{S_0} - 1)\left(\frac{b+1}{2}\right)^k \\
&\quad + b\left(((b+1)N^{1/2}_{S_0} - 1)\left(\frac{b^2+1}{2}\right)^k + \frac{(b^2-1)(N^{1/2}_{S_0} - 1)}{2}\left(\frac{b+1}{2}\right)^{k-1}\right) \\
&= ((b+1)N^{1/2}_{S_0} - 1)\left(\frac{b^2+1}{2}\right)^{k+1} + \frac{(b^2-1)(N^{1/2}_{S_0} - 1)}{2}\left(\frac{b+1}{2}\right)^k
\end{aligned}
$$

where we used $2u + b = \frac{b^2-2b+1+2b}{2} = \frac{b^2+1}{2}$ and $b(b^2-1)N^{1/2}_{S_0} - 1) - 2(N^{1/2}_{S_0} - 1)u(b+1) = \frac{b+1}{2}(N^{1/2}_{S_0} - 1)(2b(b-1) - (b-1)^2) = (b^2-1)(N^{1/2}_{S_0} - 1)\frac{b+1}{2}$. Analogously

$$
\begin{aligned}
&b(N^1_{S_{k+1}}) \\
&= \frac{(b-1)b+2}{2}\left(2\left((b+1)N^{1/2}_{S_0} - 1\right)\left(\frac{b^2+1}{2}\right)^k - 2(N^{1/2}_{S_0} - 1)\left(\frac{b+1}{2}\right)^k\right) \\
&\quad + \frac{b-1}{2}\left(2((b+1)N^{1/2}_{S_0} - 1)\left(\frac{b^2+1}{2}\right)^k + (b^2-1)(N^{1/2}_{S_0} - 1)\left(\frac{b+1}{2}\right)^{k-1}\right)
\end{aligned}
$$

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,

$$
\begin{aligned}
&(b-1)(N^0_{S_{k+1}} - 1) \\
&= b(b-1)(N^0_{S_k} - 1) + b(b-1)^2 N^1_{S_k} \\
&= 4b^2(N^2_{S_k} - 1) + 4ubN^1_{S_k} + ((b-1)(N^0_{S_0} - 1) - 4b(2N^{1/2}_{S_0} - 1))b^{k+1} \\
&= 4b(N^2_{S_{k+1}} - 1) + ((b-1)(N^0_{S_0} - 1) - 4b(2N^{1/2}_{S_0} - 1))b^{k+1}
\end{aligned}
$$

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.

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

The second part concerns the case $b = 2$. In particular, we have $u = 0$ in (8) and $N^1_{S_k}$ and $N^1_{S_k}$ decouple partly. Thus we get $N^2_{S_{k+1}} = (2N^{1/2}_{S_0} - 1) \cdot 2^{k+1} + 1 = 2 \cdot ((2N^{1/2}_{S_0} - 1) \cdot 2^k + 1) - 1 = 2(N^2_{S_k} - 1) + 1$ inductively with induction start $N^2_{S_0} = 2N^{1/2}_{S_0}$. Next, $N^1_{S_{k+1}} = 2^k((2N^{1/2}_{S_0} - 1)(k+1) + 4N^{1/2}_{S_0}) = 2 \cdot 2^{k-1}((2N^{1/2}_{S_0} - 1)k + 4N^{1/2}_{S_0}) + ((2N^{1/2}_{S_0} - 1) \cdot 2^k + 1) - 1 = 2N^1_{S_k} + N^0_{S_k} - 1$ follows inductively from induction start $N^1_{S_0} = 2^{-1}((2N^{1/2}_{S_0} - 1)(-1+1) + 4N^{1/2}_{S_0}) = 2N^{1/2}_{S_0}$. Altogether we get the tuple size

$$\begin{aligned}
N^0_{S_{k+1}} &= 2^{k-1}((2N^{1/2}_{S_0} - 1)(k+1)^2 + (6N^{1/2}_{S_0} + 1)(k+1) + 4(N^0_{S_0} - 1)) + 1 \\
&= 2^{k-1}((2N^{1/2}_{S_0} - 1)(k^2 + 2k) + (6N^{1/2}_{S_0} + 1)k + 8N^{1/2}_{S_0} + 4(N^0_{S_0} - 1)) + 1 \\
&= 2 \cdot (2^{k-2}((2N^{1/2}_{S_0} - 1)k^2 + (6N^{1/2}_{S_0} + 1)k + 4(N^0_{S_0} - 1)) + 1) \\
&\quad + 2 \cdot 2^{k-1}((2N^{1/2}_{S_0} - 1)k + 4N^{1/2}_{S_0}) - 1 \\
&= 2N^0_{S_k} + 2N^1_{S_k} - 1
\end{aligned}$$

where we started our induction with $N^0_{S_0} = \frac{1}{4} \cdot 4(N^0_{S_0} - 1) + 1$. □

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 number of building blocks and the tuple size:

Lemma 4. *Let $b > 2$ be even. Define*

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

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 $k = 0$. Inductively we get

$$\begin{aligned}
\tilde{N}^2_{S_{k+1}} &= \frac{1}{2}\left(\frac{b^2}{2}\right)^{k+1} + 1 = \frac{b(b-2) + 2b}{4}\left(\frac{b^2}{2}\right)^k + 1 = u\left(\frac{b^2}{2}\right)^k + \frac{b}{2}\left(\frac{b^2}{2}\right)^k + 1 \\
&= u\tilde{N}^1_{S_k} + b(\tilde{N}^2_{S_k} - 1) + 1 \leq uN^1_{S_k} + b(N^2_{S_k} - 1) + 1 = N^2_{S_{k+1}}
\end{aligned}$$

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

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

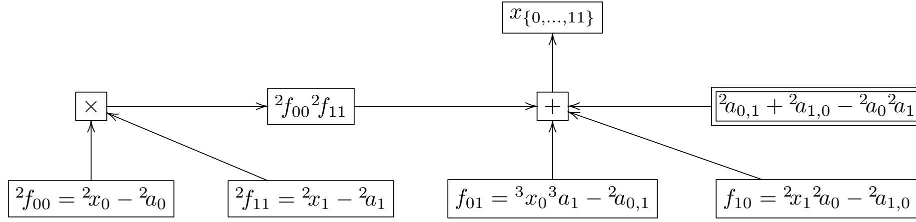
Finally,

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

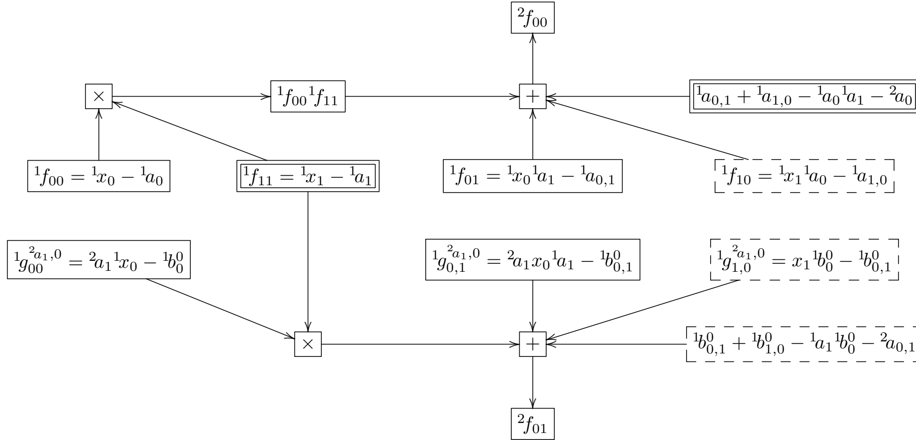
since $b - 1 \leq \left(\frac{b^2}{2}\right)^{k+1}$ for $k \geq 0, b \geq 4$. □

Example 3. In this example we visualize Example 2 by diagrams. We will use double frames for building blocks. To account for Remarks 2 to 4 we will only mark one part of a building block with the double frame and the other parts with a dashed frame to make counting easier.

Take $S_{3,0} = \{0,\ldots,11\}$, $S_{2,0} = \{0,\ldots,5\}, S_{2,1} = \{6,\ldots,11\}$ and set ${}^2x_0 = x_{S_{2,0}}, {}^2x_1 = x_{S_{2,1}}$



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



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

Observe that we got $1 + 2 \cdot 2 + 2 \cdot 6$ building blocks and hence with the intial 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.

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 [29].

Remark 10. To describe $\mathcal{F}_{\text{online}}$, we used a modification of the functionality $\mathcal{F}_{\text{AMPC}}$ from [30]. One can also use $\mathcal{F}_{\text{Online}}$ from [29] or similar functionalities.

Remark 11. The functionality $\mathcal{F}_{[\![\cdot]\!]}$ can be realized as in [49, Fig. 4].

Lemma 5. *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 [30]. □

Table 2: Arithmetic tuples to compute $x_0 \cdots x_{m-1}$ optimized for tuple size. Numbers indicate the degree of the building block, 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)))

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

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.

Arithmetic. On input $(\text{Arithmetic}, (\text{id}_{x_k})_{0 \le k < m}, f, \text{id}_z)$ for an arithmetic circuit 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 1.7: Ideal functionality for the online phase.

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

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 $(a_1, \ldots, a_l)$ and compute the tuple $(a_1, \ldots, a_k)$ for the respective building blocks.[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] $m \le l < k$ as also building blocks contain masks and we need additional tuple entries to compute building blocks.

Protocol 1.8: Preprocessing functionality.

C Further Results on the Offline Phase

This section contains results that can be used in our offline phase. It starts wit a subsection on sacrificing for binomial and arithmetic tuples which can be applied to most of the currently implemented actively secure offline phases like the once in [30] or [49]. In Appendix C.2 we offer options on how to realize the ZKP functionality used in Section 6.2. Finally we shortly discuss arithmetic tuple production with leveled homomorphic encryption.

C.1 Extended Sacrificing Technique

This appendix contains a new sacrificing technique for binomial and arithmetic tuples which can also be applied to most of the currently implemented actively secure offline phases like the once in [30] or [49].

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 1.9: CheckMAC

To make sure that the binomial tuples are indeed in the right form, we need to extend the well-know sacrificing technique from [30] in Protocol 1.10.[24]

$\Pi_{\text{sacrificing}}$

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 1.10: Verification for generalized tuples by sacrificing.

Inspecting Protocol 1.10, we see that it is obliviously correct. We use the simple identity

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

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 $\sum_{i=1}^n [\text{sac}]_i = 0$. On the other hand, if

[24] We will use the index notation from Section 4.

$\sum_{i=1}^{n} [\text{sac}]_i = 0$, 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 $\mathcal{F}_{\text{rand}}$). The Schwartz-Zippel Lemma for finite fields [62] guarantees that the probability that a random s is a zero of a total degree $d \geq 0$ polynomial f is smaller than $\frac{d}{p}$: $\Pr(f(s_0, \ldots, s_{m-1}) = 0 \mid r_s \in \mathbb{F}_p) \leq \frac{d}{p}$ 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 12. Recall from Section 5.2 that building blocks 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 arithmetic tuple of the same type for the same function, we can simply apply the sacrificing step for two binomial tuples from Protocol 1.10.

C.2 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.[25] Protocols 1.11, 1.13 and 1.15 depict the (exposition-only) functionalities used in Protocols 1.3, 1.12 and 1.14, respectively. The main reason for the ZKPs is that the parties need to know that the ciphertexts $\text{Enc}_{\text{pk}_j}([c]_j)$ 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 1.12) requires parties to prove correct decryption in zero-knowledge. This protocol leaves the responsibility for proving that $\text{Enc}_{\text{pk}_j}([c]_j)$ has small noise with P_j. Verifiable decryption can be achieved efficiently with recent protocols [39,55,69].

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

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

Send $(\text{Enc}_{\text{pk}_i}(a), P_i, \text{ok})$ 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 $(\text{Enc}_{\text{pk}_i}(a), P_i, \text{fail})$.

Protocol 1.11: Ideal functionality for verifiable decryption.

The second approach (Protocol 1.14) requires parties to prove correct decryption in zero-knowledge but it is done differently to the approach taken in

[25] 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 1.12: Multiplication using an LHE scheme with ZKP of decryption.

Protocol 1.12. Instead of proving 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 1.13.

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

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 1.13: 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 1.14: Multiplication using an LHE scheme with randomness extraction.

The third and maybe most straighforward approach (used in Protocol 1.3) requires parties to prove correct multiplication in zero-knowledge. Examples for protocols that provide verifiable multiplication can, for example, be found in the BDOZ [9] and below.

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

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 1.15: Ideal functionality for verifiable multiplication.

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 1.4. This can be done by slightly adapting existing zero-knowledge proofs, e.g. from [30], [29], [49] or [4]. As an example we present suitable modifications to [30], Fig. 9 with slight simplifications to the bounds similar to [49], 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 [30] for non-interactive variants.

Let B^{τ}_{plain} and B^{ρ}_{rand} be the ZKP bounds introduced in [30], i.e. $B^{\tau}_{\text{plain}} = 2^{\sec}\rho$ and $B^{\rho}_{\text{rand}} = 2^{\sec}\rho$. Let $V = 2\sec - 1$, $M_e \in \{0,1\}^{V \times \sec}$ the matrix associated to a challenge $e \in \{0,1\}^{\sec}$ with $(M_e)_{ij} = e_{i-j+1}$ for $1 \leq i - j + 1 \leq \sec$ and zero otherwise. Let $U, X \in R^{\sec}$ be a plaintext vector, $R \in R^{\sec \times 3}$ the encryption randomness. $\text{Enc}_{\text{pk}}(X, R) = (\text{Enc}_{\text{pk}}(X_i, R_i))_{1 \leq i \leq \sec}$ 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 [49] 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

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

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 $B^{\tau}_{\text{plain}}, B^{\xi}_{\text{plain}}, B^{\chi}_{\text{plain}}, B^{\rho}_{\text{rand}}$.

Proof. Completeness follows as in [30], 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 [30]. To account for the additional U, we note that $(M_e - M_{e'})U = Q$ obviously has a solution. Hence we get

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 $A = \mathrm{Enc}_{\mathrm{pk}}(b)W + \mathrm{Enc}_{\mathrm{pk}}(Y, S)$ to the verifier.
2. The verifier selects $e \in \{0,1\}^{\mathrm{sec}}$ 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 $D = \mathrm{Enc}_{\mathrm{pk}}(b)Q + \mathrm{Enc}_{\mathrm{pk}}(Z, T)$ 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 1.16: The Zero-Knowledge Protocol.

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 13. Please note that Enc(b) is constantly 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 [22] and [12] 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, [12] 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 [24] 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 14. To reduce the overhead introduced by noise flooding is an ongoing research task. There are however promising results as the ones announced in [71] that might be applied in our setup, too.

C.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 1.17:[26]

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. [30]).
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 1.17: Protocol for generation of $[a]^e$ for all $0 \le e_j \le d_j$ using leveled homomorphic encryption.

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 C.1.

As in Remark 7 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 arithmetic tuples 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. [58]) or using the natural action of the Galois group in case R is a underlying cyclotomic field extension (cf. [37]), can be used to improve the performance of the offline phase.

D Prefix Products with Arithmetic Tuples

Here, we describe how one can add on the approach presented in Section 5 to additionally get all the prefix products. For simplicity, we assume that we have to

[26] We use the index notation from Section 4.

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 arithmetic tuples approach gives us $x_0 - a_0, x_0x_1 - a_{01}, \dots, x_0 \cdots x_{m'-1} - a_{0,\dots,m'-1}$ for all $m' < m$ that are again powers of two. $x_0 - a_0$ is an initial masked value and the other terms are building blocks or are publicly computed from them. We get similarly structured terms (again as masked value, building block, 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 $x_l, \dots, x_h$ 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 arithmetic tuples approach of Section 5). Then, we can compute the shared prefix products of $x_0, \dots, x_{2m-1}$ as follows:

1. Compute shares of the prefix products for $x_0, \dots, x_{m-1}$ and $x_m, \dots, x_{2m-1}$.
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 5).

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 $p_{0,2^{n+1}-1,i}$.

E Further Specific of the Implementation and Evaluation

The results of Figs. 3 to 5 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). 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).

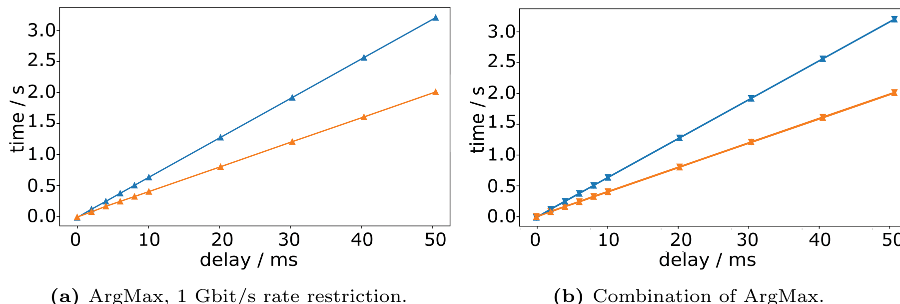
For the ranking benchmark, we chose to compare our implementation to MP-SPDZ with edabits [32] as it is MP-SPDZ's recommendation for our test program. We can see that this is indeed an improvement over the standard implementation,[27] however our new tuple-based approach clearly beats both existing approaches.

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, arithmetic tuples are created naively by simply splitting products in half recursively instead of finding local arithmetic circuits with optimal size and/or better bandwidth. Bandwidth and/or size optimal partitions could be implemented on top of our results from Section 5.2 instead. Another optimization opportunity is the one shown in Protocol 1.2 (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 arithmetic tuples and the input round of another computation. Currently, every operation based on arithmetic tuples[28] 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 5 and 6 show that there is no significant impact of the bandwidth overhead in this example. Similar results hold for all our evaluations.

[27] Note that edabits are an improvement in Fig. 4b only for very low latency.
[28] Except operations with binomial tuples; these are implemented in one round.



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

(b) Combination of ArgMax.

Fig. 6: The right diagram contains the ArgMax Evaluations from Figure 5 and the left side in one diagram for comparison.

F 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: [56] uses shared polynomials, [34,26] use homomorphic encryption in the online phase, [40] also uses homomorphic encryption but in a single centralized server setup. Of course any fully homomorphic encryption scheme like the original protocol by Gentry [36] can also be used to evaluate a polynomial. Another idea is to use oblivious transfer-based techniques like in [59] or [35,70] 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 [46].

In [16] or [68], 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 [30], e.g. [5,6,25].

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 [2,11] or [33].

Finally, there is also the recent research direction of non-interactive MPC (cf. [31,41,42,43,45]) 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).