Institute of Information Security

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Applications for Arithmetic Tuples

Mika Schieber

Course of Study:	Informatik
Examiner:	Prof. Dr. Ralf Küsters
Supervisor:	Marc Rivinius, M.Sc.
Commenced:	June 22, 2022
Completed:	December 22, 2022

Abstract

Secure Multi-Party Computation (SMPC) is a subfield of cryptography that allows multiple parties to compute a function without disclosing the inputs. Different types of specialised computation of specific (sub)functions are used to make SMPC computations more efficient. A recently published paper introduced Arithmetic Tuples, a new approach for evaluating multivariate polynomials and thereby, arithmetic circuits, in a minimal number of rounds and with practicable precomputation. In this thesis, we demonstrate the practicality of the new approach by applying it to a variety of real-world applications in which it has the potential to be particularly effective. These applications are multiplexers, permutations, demultiplexers and prefix products, which include functions with several outputs. We analyze each application and compare Arithmetic Tuples to the existing approaches Beaver Triples and Binomial Tuples. Comparison criteria are the number of rounds, the number of elements to be precalculated and the number of elements to be communicated.

Abstract in German (Kurzfassung)

Secure Multi-Party Computation (SMPC) ist ein Teilgebiet der Kryptographie, das es mehreren Parteien ermöglicht, eine Funktion zu berechnen, ohne die Eingaben offenzulegen. Um SMPC-Berechnungen effizienter zu machen, werden u.a. verschiedene Arten der spezialisierten Berechnung bestimmter (Teil-)Funktionen verwendet. In einer kürzlich veröffentlichten Arbeit wurden arithmetische Tupel vorgestellt, ein neuer Ansatz zur Berechnung multivariater Polynome und damit arithmetischer Schaltungen in einer minimalen Anzahl von Runden und mit praktikabler Vorberechnung. In dieser Arbeit, wenden wir den neuen Ansatz auf praxisrelevante Anwendungen an, für die der neue Ansatz vielversprechend erscheint. Diese Anwendungen sind Multiplexer, Permutationen, Demultiplexer und Präfixprodukte, die Funktionen mit mehreren Ausgängen beinhalten. Jede Anwendung wird analysiert und Arithmetic Tuples werden mit den bestehenden Ansätzen Beaver Triples und Binomial Tuples verglichen. Vergleichskriterien sind die Anzahl der Runden, die Anzahl der vorzuberechnenden Elemente und die Anzahl der zu übermittelnden Elemente.

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

DEMUX	Demultiplexer. 7
LSB	Least Significant Bit. 33
MAC	Message Authentication Code. 20
MSB	Most Significant Bit. 33
MUX	Multiplexer. 7
ORAM	Oblivious RAM. 17
RAM	Random Access Memory. 43
SMPC	Secure Multi-Party Computation. 15

1 Introduction

Secure Multi-Party Computation (SMPC) is a subfield of cryptography that allows multiple parties to compute a function on their inputs while not disclosing any information about the inputs other than that can be derived from the final output. For this, the participating parties agree on an arithmetic or Boolean circuit describing the function and deploy techniques to obscure inputs and intermediate results.

For arithmetic circuits, this can be achieved by additive secret sharing, which allows simple linear operations to be performed on the inputs while keeping them private. However, multiplications of secret values require further correlated randomness independent of the inputs. For this, in SMPC protocols like SPDZ [3], these values are preprocessed in an offline phase, whereas the actual computation takes place in the online phase. One widely used form of correlated randomness is Beaver Triples [1], with which it is possible to multiply two values with one round of communication and three random values a, b, ab. By this, it is then possible to evaluate any arithmetic circuit consisting of AND and MULT gates and, therefore, any multivariate polynomial. Unfortunately, Binomial Tuples lead to logarithmic many rounds for a product of variables which slows the computation, especially in networks with high latency.

An extension of Beaver Triples that allows multiplication of any number of values in one round, called Binomial Tuples [9, 11] leads to an exponentially growing number of correlated randomness needed. Products of many variables can cause high memory requirements to store the random values, and also generating them can become too time-consuming. Addressing this problem Reisert et al. [11] recently introduced a generalization of Beaver Triples and Binomial Tuples that enable products to be computed with minimal rounds and compared to Binomial Tuples, a practicable number of correlated random values also for larger products.

In this work, we want to explore several possible applications for Arithmetic Tuples to aid in the choice between Beaver Triples, Binomial Tuples and Arithmetic Tuples. One application is multiplexers that have many use cases in SMPC. They can be used to implement array accesses, branches or functions with several outputs like permutations of variables. Other applications we analyze are demultiplexers, the counterpart to multiplexers, and prefix products which have been addressed in [11]. In most cases, there are different approaches to realize these applications which we also compare with each other.

1.1 Contribution

- We provide algorithms to determine the tuple size and bandwidth to compute a multilinear polynomial using Beaver Triples, Binomial or Arithmetic Tuples. These algorithms can be applied to any multilinear polynomial to determine which multiplication approach is best in a specific scenario. Each algorithm can also be modified to suit different applications better and can be extended to general polynomials.
- We present several possible applications for Arithmetic Tuples and give methods to determine the tuple size, bandwidth and round complexity for each application with Beaver Triples, Binomial and Arithmetic Tuples. For each application, we compare several approaches, like multi-round evaluation or variations of the polynomials, that lead to different metrics. One of these applications is multiplexers, which are an important component in many use cases like branching.
- In the results, we compare how Beaver Triples, Binomial and Arithmetic Tuples perform for each application. For this, we use the previously presented methods to compute each approach's tuple size, bandwidth and round complexity.

1.2 Structure

In the central part of this work, first Chapter 3 introduces Additive Secret-Sharing and the methods to perform multiplications using Beaver Triples, Binomial- and Arithmetic Tuples. Furthermore, the metrics tuple size, bandwidth and round complexity we use to compare these methods are presented. Next Chapter 4 first introduces algorithms to measure these metrics for a given multilinear polynomial using Beaver Triples, Binomial- and Arithmetic Tuples and afterwards presents possible use cases for Arithmetic Tuples. These are multiplexers, permutations, demultiplexers and prefix products. We define multilinear polynomials for each of these functions to apply the previously introduced algorithms or find closed formulas for the metrics. Finally, Chapter 5, compares the resulting metrics for each use case to give directions on what method to use for a scenario.

2 Related Work

In this work, we look at applications for Arithmetic Tuples. To our knowledge, there is no further research on applications for Arithmetic Tuples yet, except for what is presented in the original paper on Arithmetic Tuples [11].

Here the protocol for an equality test and less-then comparison is shown. The equality check of two private values is based on a bitwise equality check of a private value with a public constant. To check if two private values x, y are equal $c = x - y + r$ for a random shared value, r is opened, and the bitwise equality check of c and $[r]$ is computed. When c and r are decomposed to k bits, this equality check requires the multiplication of k shares, for which Arithmetic Tuples can be used. In the less-than-protocol, a prefix-OR is used, which can be expressed as a prefix product. Computing prefix products with Arithmetic Tuples has therefore also been addressed in the paper. The paper also benchmarks the evaluation of multivariate polynomials, establishing a ranking of inputs, and evaluating neural networks.

For other related work, on the one hand, we have further alternatives or extensions to Binomial Tuples that can be used to evaluate any multivariate polynomial and therefore also the polynomials implementing our applications [2, 3, 8]. Regarding alternatives to Binomial Tuples in the two-party case, Yao's Garbled Circuit Protocol provides a constant round complexity to evaluate any function represented as a Boolean circuit[4].

On the other hand, there is research on the specialized implementation of the here presented applications of Arithmetic Tuples. One use case of multiplexers is conditional branching to select the correct branch. Each branch has to be evaluated so that its result can be input into a multiplexer. For this reason, the communication complexity depends on the whole circuit and not only on the single active branch, which can slow the computation. In the work of Goel et al. [6], an alternative approach is presented, for which the communication complexity depends only on the size of the largest branch. It ensures that the information on which branch was selected is not leaked and can be used to evaluate an arithmetic circuit over any field with a non-constant round complexity.

Another use case of multiplexers is accessing arrays. For this, every value in the array has to be touched since it is an input to the arithmetic circuit. When working with a large amount of data, reading it in its entirety becomes impractical. This can be avoided by using Oblivious RAM (ORAM), which obscures access patterns without the need for accessing each entry [5, 10].

3 Background and Fundamentals / Preliminaries

3.1 Additive Secret-Sharing

Secret Sharing is a useful tool in SMPC. It allows sharing a secret value among a number of parties so that it can only be reconstructed when a sufficiently large number of parties collaborate. One prominent example is Shamir Secret Sharing [13] based on polynomial interpolation. It functions by using the fact that a polynomial f with degree $k-1$ can be unambiguously reconstructed with at least k points of f, then allowing to evaluate $f(0) = D$ to learn the secret D.

Here, however, the much simpler additive secret sharing is used. First of all, all computations take place in a finite field like $\mathbb{Z}_{2^{32}}$ or $\mathbb{Z}_p$ for a prime number p. With additive secret sharing a secret x is shared among N parities $P_i, 0 \leq i < N$ so that $x = \sum_{i=0}^{N-1} [x]_i$. Where $[x]_i$ denotes the share of x held by party P_i.

Additive secret sharing ensures that the value of x can only be revealed if all N parties collaborate, and even $N-1$ dishonest parties can not learn anything about x when collaborating. It can therefore be used in a dishonest majority setting, where at most $N-1$ parties are corrupted. For example, if Bob wants to share a secret value $x = 42$ with Alice and Eve, he could give Alice the share $[x]_A = 17$, Eve the share $[x]_E = 5$ and only remember his own share $[x]_B = 42 - [x]_A - [x]_E = 20$. Then $[x]_A + [x]_B + [x]_E = x$, but even if Alice and Eve collaborate and reveal their shares to each other, they can not learn Bob's secret since $[x]_B$ and therefore also x could be any value in the finite field.

To reveal (open) a shared secret x, all N parties need to send their share of x to all other parties and then compute the sum of these shares.

Secret sharing alone has many applications, like distributing an encryption key among several parties so that compromising a single party does not cause any damage. In this work, nevertheless, we are interested in performing computations on these shares so that nothing more than the final result is unveiled. For this, additive secret sharing has the advantage of being linear, allowing one to perform some operations on shares of values so that after opening, the same result is obtained as if these operations had been performed on the values themselves.

For one, the share of the sum of two values x, y is the same as the sum of the shares of x, y ($[x+y]_i = [x]_i + [y]_i$). The following equation shows this:

$$\sum_{i=0}^{N-1} [x+y]_i = x + y = \sum_{i=0}^{N-1} [x]_i + \sum_{i=0}^{N-1} [y]_i = \sum_{i=0}^{N-1} ([x]_i + [y]_i) \tag{3.1}$$

Furthermore it holds, that $[k \cdot x]_i = k \cdot [x]_i$ (see Equation (3.2)) and, that $[x+k]_i = [x]_i + \delta_{1,i} \cdot k$ (see Equation (3.3)) for a publicly known constant k and the Kronecker delta $\delta_{1,i} = 1$ if $i = 0, 0$ if $i \neq 0$.

$$\sum_{i=0}^{N-1} [k \cdot x]_i = k \cdot x = k \cdot \sum_{i=0}^{N-1} [x]_i = \sum_{i=0}^{N-1} k \cdot [x]_i \tag{3.2}$$

$$\sum_{i=0}^{N-1} [x + k]_i = x + k = \sum_{i=0}^{N-1} [x]_i + k = ([x]_0 + k) + \sum_{i=1}^{N-1} [x]_i \tag{3.3}$$

None of these techniques requires communication between the parties and can thus be performed locally. Difficulties arise with the multiplication of two secret values since $[x \cdot y]_i \neq [x]_i \cdot [y]_i$ in most cases. To this, the so-called Beaver Triples by Beaver [1] are a possible solution. A Beaver Triple is a three-tuple $([a]_i, [b]_i, [ab]_i)$ containing correlated randomness that is generated before the actual computation (online phase) in the offline phase. Using these tuple entries then allows $[x \cdot y]_i$ to be calculated by first opening the masked values $(x - a)$ and $(y - b)$. This does not reveal any information about x and y since a, b are random shared values. Because of that, we will call tuple entries like a, b sometimes mask. By opening, $(x - a), (y - b)$ became publicly known constant and each party P_i can calculate

$$(x - a)(y - b) \cdot \delta_{0,i} + [a]_i(y - b) + [b]_i(x - a) + [ab]_i \tag{3.4}$$

to obtain a share of $x \cdot y$. From now on, When adding a public constant k, we will only write $\cdots + k$ instead of $\cdots + k \cdot \delta_{0,i}$ even though it still applies that k is only added to the share of P_0.

To ensure that the outputs of a SMPC are correct (to provide active security where parties could deviate from the protocol), in SPDZ, the shares are authenticated using a Message Authentication Code (MAC) key. For simplicity, we will disregard authentication in the following sections. However, note that all operations are compatible with the MAC authentication.

3.2 Metrics

SMPC has additional parameters influencing the performance compared to a typical local computation. Mainly, that data has to be transmitted between different parties. Depending on the properties of the network connecting participating parties, the time or resources needed to perform a SMPC vary.

For one, if the latency in the network is high, a small round complexity is desirable. Typically, in the here used SMPC protocols, after performing computations locally in parallel, the parties need to communicate to share intermediate or final results and continue the calculation. The number of times this communication occurs is called round complexity. Yet, the initial opening round is not counted because its structure can vary depending on the specific protocol. For example, recall the computation of a product $x \cdot y$ using beaver triples. Here we first had to open the masked values $(x - a)$ and $(y - b)$ (opening round) and then the locally computed shares of $(x - a)(y - b) + [a](y - b) + [b](x - a) + [ab]$ (second round). So, in that case, the round complexity is one.

Next, data rate restrictions can be a problem regarding SMPC. Therefore, with the bandwidth, we also measure the number of field elements transmitted between the parties. Here, when we have a look at the computation of a product $x \cdot y$ using beaver triples. Because of the three ring elements $(x - a)$, $(y - b)$ and $(x - a)(y - b) + [a](y - b) + [b](x - a) + [ab]$ we get a bandwidth of three.

Finally, independent of the network parameters, the number of additional field elements (tuple entries) required in the computation slows down the generation of these elements in the offline phase and requires sufficient memory resources. The tuple size measures this factor. Again, for the same example as above, the tuple size is $|([a], [b], [ab])| = 3$.

3.3 Binomial Tuples

Binomial Tuples make it possible to compute arbitrary polynomials $f(x_0, \dots, x_{n-1})$ in only one round of communication in addition to the opening round. This similarity to Arithmetic Tuples makes a comparison between the two approaches meaningful. Additionally, Binomial Tuples are used in Arithmetic Tuples to build elementary building blocks (see Section 3.4). For these reasons, this section gives a short introduction to Binomial Tuples. The complete proof can be found in [11].

The construction of Binomial Tupels is based on the Binomial Theorem, which states that a Polynomial $(x + y)^n$ can be expanded to $\sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k$. This can be used to calculate shares of x^n using a previously opened masked value $(x - a)$ and shares $[a^e]_i$ for $1 \le e \le n$:

$$(3.5)\quad [x^n]_i = [((x - a) + a)^n]_i = \sum_{e=0}^{n} \binom{n}{e} (x - a)^{n-e} \, [a^e]_i$$

However, to compute shares of a multivariate monomial of the form $f(x_0, \dots, x_{n-1}) = \prod_{k=0}^{n-1} x_k^{d_k}$ this is generalised using a tuple $[(a^{(e_0,\dots,e_{n-1})})_{(e_0,\dots,e_{n-1}) \in E}]_i$ for $E = \bigtimes_{k=0}^{n-1} \{0, \dots, d_k\}$ and $a^{(e_0,\dots,e_{n-1})} = \prod_{j=0}^{n-1} a_j^{e_j}$. The following equation then gives a sharing of f:

$$(3.6)\quad [f(x_0, \dots, x_{n-1})] = \sum_{e \in E} \left([a^e]_i \prod_{k=0}^{n-1} \binom{d_k}{e_k} (x_k - a_k)^{d_k - e_k} \right)$$

For example to compute $x_0^2 x_1$ we need the tuple $\left(a^{(0,0)}, a^{(0,1)}, a^{(1,0)}, a^{(1,1)}, a^{(2,0)}, a^{(2,1)}\right) = \left(1, a_1, a_0, a_0 a_1, a_0^2, a_0^2 a_1\right)$. Using equation 3.6 we then obtain:

$$\begin{aligned}
[x_0^2 x_1]_i &= 1 \cdot \binom{2}{0}(x_0 - a_0)^2 \cdot \binom{1}{0}(x_1 - a_1)^1 + [a_1]_i \cdot \binom{2}{0}(x_0 - a_0)^2 \cdot \binom{1}{1}(x_1 - a_1)^0 \\
&\quad + [a_0]_i \cdot \binom{2}{1}(x_0 - a_0)^1 \cdot \binom{1}{0}(x_1 - a_1)^1 + [a_0 a_1]_i \cdot \binom{2}{1}(x_0 - a_0)^1 \cdot \binom{1}{1}(x_1 - a_1)^0 \\
&\quad + [a_0^2]_i \cdot \binom{2}{2}(x_0 - a_0)^0 \cdot \binom{1}{0}(x_1 - a_1)^1 + [a_0^2 a_1]_i \cdot \binom{2}{2}(x_0 - a_0)^0 \cdot \binom{1}{1}(x_1 - a_1)^0 \\
&= (x_0 - a_0)^2(x_1 - a_1) + [a_1]_i(x_0 - a_0)^2 + 2 \cdot [a_0]_i \cdot (x_0 - a_0)(x_1 - a_1) \\
&\quad + 2 \cdot [a_0 a_1]_i(x_0 - a_0) + [a_0^2]_i(x_1 - a_1) + [a_0^2 a_1]_i
\end{aligned}$$

The required tuple-size of a monomial $\prod_{k=0}^{n-1} x_k^{d_k}$ equals $|E| - 1 = \prod_{k=0}^{n-1}(d_k + 1) - 1$ since for each $e \in E$ we get a tuple entry a^e except for $e = (0, \ldots, 0)$. $a^{(0,\ldots,0)} = 1$ and therefor is a publicly known constant. In terms of bandwidth, with binomial tuples, the n masked values $(x_i - a_i)$ and the share of f have to be published, which leads to a bandwidth of $n + 1$.

Finally, to build a polynomial of several monomials, one can combine the tuples required for each monomial so that each entry occurs at most once. For example, the tuple for $x_0^2 x_1 + x_0 x_1$ is the same as the tuple for $x_0^2 x_1$ because each entry for $x_0 x_1$ is already included in the tuple for $x_0^2 x_1$.

In this work, we will have a closer look at multilinear polynomials, of which each variable is of a degree of zero or one. Thus each monomial is of the form $\prod_{k=0}^{n-1} x_k^{d_k}$ with $d_k \in \{0, 1\}$. In this case, the size of the tuple is $2^m - 1$ with $m = |\{d_k \mid d_k = 1, 0 \leq k < n\}|$. Furthermore if we have a polynomial of the form $f(x_0, \ldots, x_{n-1}) = x_0 \cdot \ldots \cdot x_{n-1} + \sum_{j=0}^{m} x_0^{d_0^{(j)}} \cdot \ldots \cdot x_{m-1}^{d_{m-1}^{(j)}}, d_l^{(j)} \in \{0, 1\}$, the tuple size for f is given by $2^n - 1$. This is because each tuple entry required for a monomial $x_0^{d_0^{(j)}} \cdot \ldots \cdot x_{m-1}^{d_{m-1}^{(j)}}$ is already included in the tuple for $x_0 \cdot \ldots \cdot x_{n-1}$, which has the size $2^n - 1$.

3.4 Arithmetic Tuples

The previously introduced Binomial Tuples achieve a minimal number of rounds, but the tuple size grows exponentially with the number of variables in a monomial. However, keeping the tuple-size minimal is essential since the offline phase has to generate the tuple in a reasonable amount of time not to become the bottleneck of a SMPC and enough memory has to be available. For instance, $x_0 \cdot \ldots \cdot x_{15}$ requires a Binomial Tuple of size $2^{16} - 1 = 65535$ whereas an example arithmetic tuple has only 157 entries.

The main idea of Arithmetic Tuples is to first build elementary building blocks from the initially published masked values and the correlated randomness included in the tuple using the Binomial Tuple approach. After publishing, these elementary building blocks can then be locally combined to receive the desired polynomial while not revealing any information except the final result.

Figure 3.1 exemplary shows this method for a monomial $x_0x_1x_2x_3$. Here x_0x_1 and x_2x_3 are built using binomial tuples in order to be multiplied to obtain $x_0x_1 \cdot x_2x_3$. However, publishing x_0x_1 and x_2x_3 would reveal more information than could be learned from $x_0x_1x_2x_3$. For which reason x_0x_1 and x_2x_3 are masked by new randomness a_{01} and a_{23}. Multiplying $(x_0x_1 - a_{01}) \cdot (x_2x_3 - a_{23})$ then yields the unwanted terms $-a_{23}x_0x_1, -a_{01}x_2x_3, a_{01}a_{23}$. To remove these, a third elementary building block $a_{23}x_0x_1 + a_{01}x_2x_3 - a_{01}a_{23}$ is added. In the dashed boxes are the equations to compute these three building blocks using binomial tuples where the 15 tuple entries used to compute $x_0x_1x_2x_3$ appear. In theory, also $a_{23}x_0x_1$ and $a_{01}x_2x_3$ are masked, but since the additive building blocks are combined, these terms cancel out immediately.

Note that $[a_{23}a_0a_1], [a_{01}a_2a_3]$ and $[a_{01}a_{23}]$ are added to the same term and can therefore be combined to a single tuple entry $[a_{23}a_0a_1 + a_{01}a_2a_3 - a_{01}a_{23}]$ In general, tuple entries that are added to the same term (and not used elsewhere) can be combined and are called additive tuple entries. This lowers the tuple size, like in this example, from 15 to 13. It is also possible to obtain a share $[x_0x_1x_2x_3]$ by adding $[a_{23}a_0a_1 + a_{01}a_2a_3 - a_{01}a_{23}]$ only after opening the building blocks.

In general, all building blocks of the form $y = a \cdot b \cdot \sum_{i=0}^{n-1} x_i - c$ can be either built directly using binomial tuples or recursively by building

$$
\begin{aligned}
y_0 &= a \cdot \sum_{i=0}^{\lceil \frac{n}{2} \rceil - 1} x_i - c_0, \\
y_1 &= b \cdot \sum_{i=\lceil \frac{n}{2} \rceil}^{n-1} x_i - c_1, \\
y_{01} &= c_1 \cdot a \cdot \sum_{i=0}^{\lceil \frac{n}{2} \rceil - 1} x_i - c_2 + c_0 \cdot b \cdot \sum_{i=\lceil \frac{n}{2} \rceil}^{n-1} x_i - c_3 - c_0 c_1 - c + c_2 + c_3
\end{aligned}
$$

and then calculating $y = y' \cdot y'' + y'''$ ($a, b, c_0, \ldots, c_3$ are tuple entries (masks) or if not present $a, b = 1, c = 0$. e.g. $x_0 \cdot \ldots \cdot x_7 = y$ for $n = 8, a = b = 1, c = 0$). y, y' are again of the same form as y and y''' consists of three terms, of which two are of the same form as y, and the third are masks that can be simply added to the building block. Here it can also be seen that the masks c_2 and c_3 cancel out if all additive building blocks are combined and therefore are unnecessary. To compute a sum of monomials, the set of building blocks and, consequently also the tuples for each of the monomials can be unified.

The general definition of Arithmetic Tuples, however, allows many other variations on how to build y, like splitting y into more than two parts or at what degree binomial tuples are used to build y. Moreover, it is possible to add additional rounds to publish intermediate results, which are then again combined using Arithmetic tuples. Also, monomials with variables x_i^d of degree $d > 1$ can be built by replacing x_i by x_i^d in the elementary building blocks. Overall, it is important to note that Arithmetic Tuples are not definite, and many variations can result in entirely different metrics. There is often a trade-off between round complexity, tuple size and bandwidth. In this work, whenever we write about *the* Arithmetic Tuple for some polynomial, it is, in fact, only one possible example Arithmetic Tuple which we defined. In most cases, it will be optimized to achieve a minimal round complexity of one. Strictly speaking, every Binomial Tuple and Beaver Triple is also an Arithmetic Tuple since Arithmetic Tuples generalize these concepts.

To better understand Arithmetic Tuples, in Figure 3.2 on page 25, it is shown how $x_0 \cdot \ldots \cdot x_3$ and $x_4 \cdot \ldots \cdot x_7$ are combined to gain $x_0 \cdot \ldots \cdot x_7$ and in Section 4.1.3 an algorithmic approach to build a monomial using arithmetic tuples is described. Furthermore, refer to [11] for the formal definition.

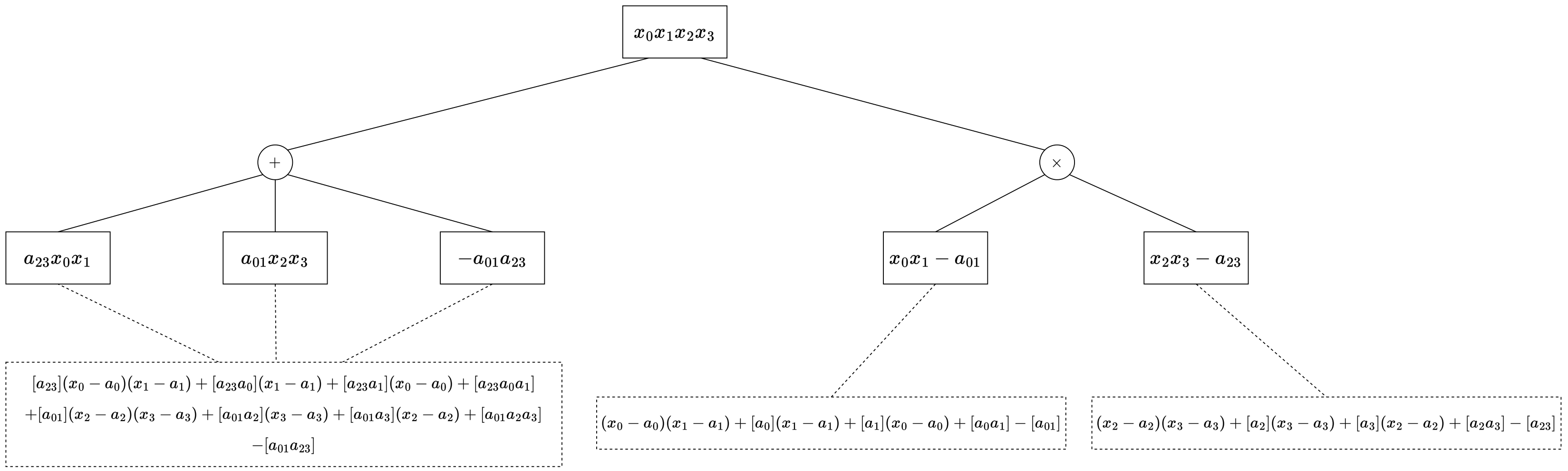

$x_0x_1x_2x_3$
$+$
$\times$
$a_{23}x_0x_1$
$a_{01}x_2x_3$
$-a_{01}a_{23}$
$x_0x_1 - a_{01}$
$x_2x_3 - a_{23}$
$[a_{23}](x_0 - a_0)(x_1 - a_1) + [a_{23}a_0](x_1 - a_1) + [a_{23}a_1](x_0 - a_0) + [a_{23}a_0a_1]$
$+[a_{01}](x_2 - a_2)(x_3 - a_3) + [a_{01}a_2](x_3 - a_3) + [a_{01}a_3](x_2 - a_2) + [a_{01}a_2a_3]$
$-[a_{01}a_{23}]$
$(x_0 - a_0)(x_1 - a_1) + [a_0](x_1 - a_1) + [a_1](x_0 - a_0) + [a_0a_1] - [a_{01}]$
$(x_2 - a_2)(x_3 - a_3) + [a_2](x_3 - a_3) + [a_3](x_2 - a_2) + [a_2a_3] - [a_{23}]$

Figure 3.1: This figure illustrates how $x_0 \cdot ... \cdot x_3$ can be computed using Arithmetic Tuples. The leaves of this tree (dashed boxes) show how the elementary building blocks are computed using the tuple entries and the previously opened masked values $x_i - a_i$ for $0 \leq i < 4$. Afterwards, these elementary building blocks are opened and combined according to the tree's structure.

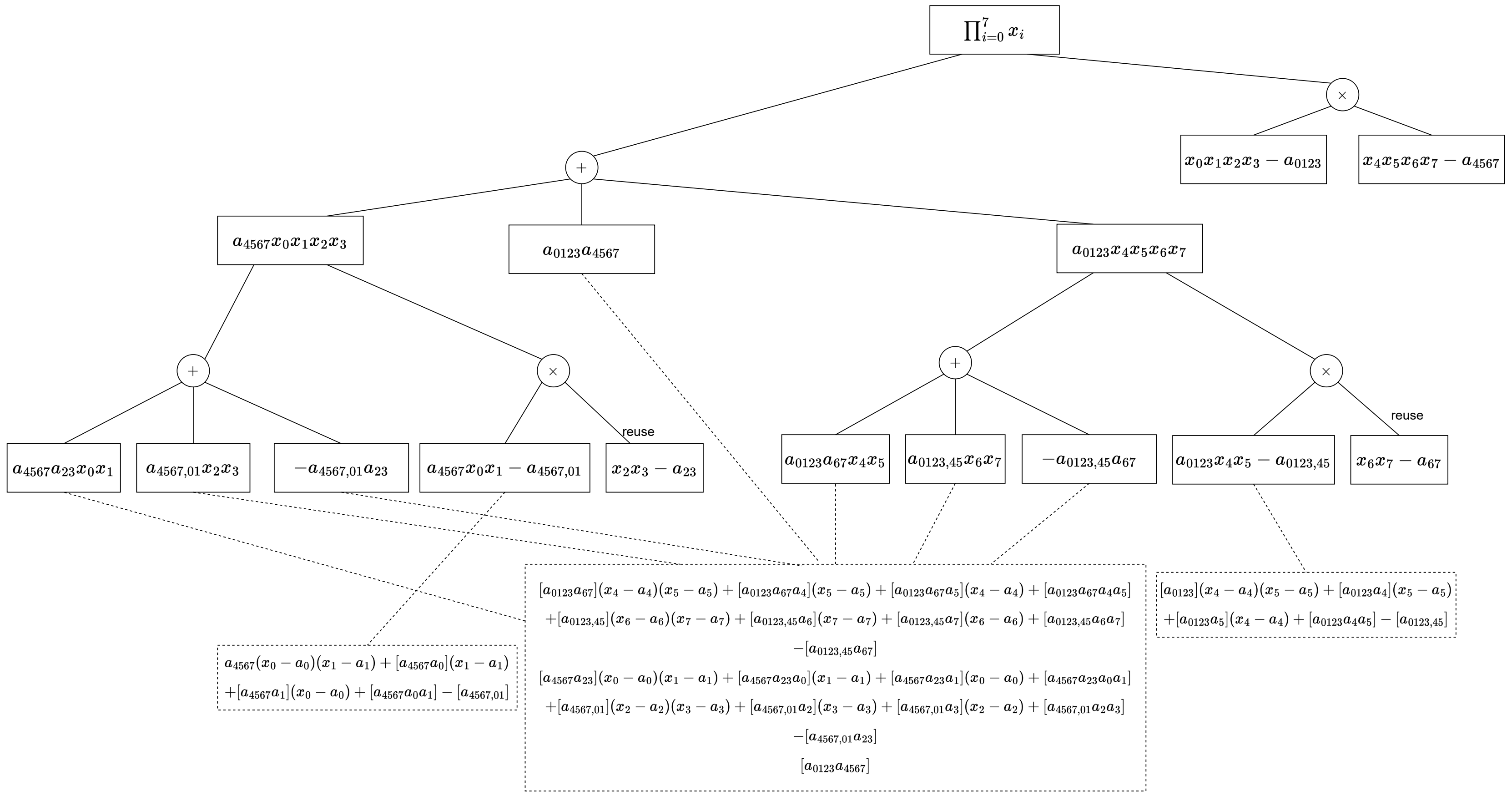

$\prod_{i=0}^{7} x_i$
$+$
$\times$
$x_0x_1x_2x_3 - a_{0123}$
$x_4x_5x_6x_7 - a_{4567}$
$a_{4567}x_0x_1x_2x_3$
$a_{0123}a_{4567}$
$a_{0123}x_4x_5x_6x_7$
$+$
$\times$
$+$
$\times$
reuse
reuse
$a_{4567}a_{23}x_0x_1$
$a_{4567,01}x_2x_3$
$-a_{4567,01}a_{23}$
$a_{4567}x_0x_1 - a_{4567,01}$
$x_2x_3 - a_{23}$
$a_{0123}a_{67}x_4x_5$
$a_{0123,45}x_6x_7$
$-a_{0123,45}a_{67}$
$a_{0123}x_4x_5 - a_{0123,45}$
$x_6x_7 - a_{67}$
$a_{4567}(x_0 - a_0)(x_1 - a_1) + [a_{4567}a_0](x_1 - a_1)$
$+[a_{4567}a_1](x_0 - a_0) + [a_{4567}a_0a_1] - [a_{4567,01}]$
$[a_{0123}a_{67}](x_4 - a_4)(x_5 - a_5) + [a_{0123}a_{67}a_4](x_5 - a_5) + [a_{0123}a_{67}a_5](x_4 - a_4) + [a_{0123}a_{67}a_4a_5]$
$+[a_{0123,45}](x_6 - a_6)(x_7 - a_7) + [a_{0123,45}a_6](x_7 - a_7) + [a_{0123,45}a_7](x_6 - a_6) + [a_{0123,45}a_6a_7]$
$-[a_{0123,45}a_{67}]$
$[a_{4567}a_{23}](x_0 - a_0)(x_1 - a_1) + [a_{4567}a_{23}a_0](x_1 - a_1) + [a_{4567}a_{23}a_1](x_0 - a_0) + [a_{4567}a_{23}a_0a_1]$
$+[a_{4567,01}](x_2 - a_2)(x_3 - a_3) + [a_{4567,01}a_2](x_3 - a_3) + [a_{4567,01}a_3](x_2 - a_2) + [a_{4567,01}a_2a_3]$
$-[a_{4567,01}a_{23}]$
$[a_{0123}a_{4567}]$
$[a_{0123}](x_4 - a_4)(x_5 - a_5) + [a_{0123}a_4](x_5 - a_5)$
$+[a_{0123}a_5](x_4 - a_4) + [a_{0123}a_4a_5] - [a_{0123,45}]$

Figure 3.2: This figure illustrates how $x_0 \cdot \dots \cdot x_{15}$ can be computed using Arithmetic Tuples. Only the additive part is shown in detail. The tree for the two multiplicative parts can be seen in Figure 3.1 with the exception of the missing masks a_{0123} and a_{4567}. Note that additive tuples are not combined in this illustration even though it is possible.

4 Implementation and Methodology

4.1 Computing the tuple size and bandwidth for a multilinear polynomial

4.1.1 Beaver Triples

With Beaver Triples, the metrics depend on the circuit used to compute the polynomial. For example, $x_0x_1x_2x_3$ can be computed in two rounds by first computing x_0x_1 and x_2x_3 and then combining them in the second round. Nevertheless, if in the first round x_0 and x_1, in the second round x_0x_1 and x_2 and in the third round $x_0x_1x_2$ and x_3 are combined, three rounds are necessary. Also, for polynomials consisting of several monomials, the splitting strategy can affect the number of three tuples needed because it can influence which blocks can be reused.

For this reason, we will first consider a general approach to determine a tuple size and bandwidth for a multilinear polynomial with the aim to minimize round complexity and, in some cases, specialized approaches for specific use cases.

The general approach is to go through each monomial in a polynomial and add each required tuple entry and building block to a set. Algorithm 4.1 therefore calls on each monomial the recursive procedure buildMonomial, which first splits monomials of degree two or more into two monomials $left, right$. $left, right$ are then used to add the tuple entries required two build $left \cdot right$ to the tuple set. Here Mask(m) means the unique mask for a monomial m, e.g. a_{01} for x_0x_1, depending on the concrete implementation. Finally, $left, right$ are added to the set of monomials that have to be opened (as a masked value), and buildMonomial is recursively called on $left$,$right$.

Storing these values in a set ensures that duplicates are not counted if we use $|Tuple|$ and $|OpenedBlocks| + 1$ to determine the tuple size and bandwidth.

To find the minimal number of rounds, the algorithm is not necessary since a monomial m_i of degree d_i requires $r(m_i) = \lceil \log d_i \rceil$ rounds to compute using beaver triples and then for a polynomial $\sum_{i=0}^{n} m_i$ $\max\{r(m_i) \mid 0 \leq i \leq n\}$ rounds are required.

4.1.2 Binomial Tuples

When considering Binomial Tuples, it is often possible to find an analytical solution to the tuple size for a family of polynomials as already mentioned in Section 3.3. In general, an upper bound to the tuple size for a multilinear polynomial $f(x_0, \ldots, s_{n-1})$ in n variables is given by $2^n - 1$, since $\{(a_0^{e_0}, \ldots, a_{n-1}^{e_{n-1}}) \mid (e_0, \ldots e_{n-1}) \in \{0,1\}^n \setminus \{(0, \ldots 0)\}\}$ includes all necessary tuple entries for all multilinear monomials in $x_0, \ldots x_{n-1}$.

Algorithm 4.1 Algorithm to compute an upper bound to the number of tuple entries for a given multilinear polynomial when using beaver triples.

```
procedure computeBeaverTupleSize($polynomial$)
    $Tuple \leftarrow \emptyset$
    $BuldingBlocks \leftarrow \emptyset$
    for all $monomials$ $m$in $polynomial$ do
        buildMonomial($m$)
    end for
end procedure
procedure buildMonomial($monomial$)
    if $monomial.degree < 2$ then
        return
    end if
    $left, right \leftarrow$ splitMonomial($monomial$)
    $Tuple \leftarrow Tuple \cup \{\text{Mask}(left), \text{Mask}(right), \text{Mask}(left) \cdot \text{Mask}(right)\}$
    $OpenedBlocks \leftarrow OpenedBlocks \cup \{left, right\}$
    buildMonomial($left$)
    buildMonomial($right$)
end procedure
function splitMonomial($monomial$)
    Input $monomial = \prod_{i=0}^{k} x_i$
    return $\prod_{i=0}^{\lfloor k/2 \rfloor} x_i, \quad \prod_{j=\lfloor k/2 \rfloor+1}^{k} x_j$
end function
```

Algorithm 4.2 Algorithm to compute the Binomial Tuple size for a given multilinear polynomial.

```
procedure computeBinomialTupleSize($polynomial$)
    $Tuple \leftarrow \emptyset$
    for all $monomials$ $m = \prod_{i=0}^{n-1} x_i^{d_i}, d_i \in \{0,1\}$ in $polynomial$ do
        $Tuple \leftarrow Tuple \cup \{(a_0^{e_0}, \dots, a_{n-1}^{e_{n-1}}) \mid e_i \in \{0,1\} \text{ if } d_i = 1, e_i = 0 \text{ else }, 0 \le i < n\}$
    end for
    Output $|Tuple|$
end procedure
```

Finding the exact tuple size for an arbitrary multilinear polynomial can become more time-consuming when its structure becomes more complex. In that case, we can use a simple algorithm (Algorithm 4.2) to determine the tuple size for a given polynomial.

However, calculating the bandwidth of a polynomial in n variables is simple since we only need to open the n masked variables and, finally, the sum of the computed shares of the monomials. This gives a bandwidth of $n + 1$ and the round complexity is always one.

4.1.3 Arithmetic Tuples

With Arithmetic Tuples, finding the tuple size and bandwidth analytically for an arbitrary polynomial becomes more difficult. Therefore we use Algorithm 4.3 that uses the Procedure BUILDTREE in Algorithm 4.4 to build a tree similar to Figure 3.1 for each monomial in the polynomial. Here each required tuple entry is added to the set $Tuple$ so that duplicates are not considered. We then get the tuple size with $|Tuple|$ and can analogically determine the bandwidth by counting the unique building blocks.

In detail, the procedure BUILDTREE 4.4 given a node and a monomial $a \cdot b \cdot \prod_{i=0}^{n-1} x_i$ with up to two prefactors a, b first checks (ll.2) if the degree of the monomial is less then four ($n \leq 3$). In that case, the termination condition is met, and the current node of the tree is considered a leaf. The monomial is then directly built using binomial tuples in the procedure ADDLEAF 4.5. This procedure adds all tuple entries to the set necessary to build the term $p \cdot \prod_{i=0}^{k} x_i - mask, k \in \{1, 2\}$ using binomial tuples. Otherwise, the monomial is split into the two monomials $leftMonomial = a \cdot \prod_{i=0}^{\lfloor k/2 \rfloor} x_i$ and $rightMonomial = b \cdot \prod_{j=\lfloor k/2 \rfloor+1}^{k} x_j$ (ll.6).

The current node then gets two children. One $AddNode$ indicating that its children should be summed up, and one $MultNode$ indicating that its children should be multiplied. This $MultNode$ has the two masked monomials $leftMonomial - \textsc{Mask}(leftMonomial)$ and $rightMonomial - \textsc{Mask}(rightMonomial)$ as children. $\textsc{Mask}(m)$ is a unique mask for a term m so that $m = m' \Leftrightarrow \textsc{Mask}(m) = \textsc{Mask}(m')$. The $AddNode$ has three children compensating for the unwanted terms that arise if the two masked monomials are multiplied. These are $leftMonomial \cdot \textsc{Mask}(rightMonomial)$, $rightMonomial \cdot \textsc{Mask}(leftMonomial)$ and $\textsc{Mask}(leftMonomial) \cdot \textsc{Mask}(rightMonomial)$. On each of these five nodes, the procedure is then called recursively except for $\textsc{Mask}(leftMonomial) \cdot \textsc{Mask}(rightMonomial)$ because it is only a product of tuple entries.

This product, together with the mask of the current node, should be added to the tuple. Since they are not used in multiplication, they can be combined. Therefore the function FINDHIGHESADDNODE starting from an $AddNode$ returns the highest $AddNode$ in the tree that can be reached without crossing a $MultNode$. All additive tuple entries underneath this $highestAddNode$ can be combined into a single tuple entry.

To achieve this, the tuple entries are not directly added to the tuple but to a tuple entry stored in the $highestAddNode$ (ll.17). This tuple entry is only added to the set $Tuple$ at the end of the procedure so that all children have already returned and added all terms to the entry (ll.22). The same strategy is applied to combine the to-be-opened building blocks. In Figure 3.2 on page 25 is an example of how building blocks can be combined. Here all but two leaves are combined into a single building block (big dashed box). All tuple entries that are only added or subtracted to this block can be combined into a single tuple entry.

When it comes to a polynomial of several monomials, it is possible to combine all additive building blocks of the highest $addNode$ (i.e. that are not used in multiplication) and also monomials that are directly built using Binomial Tuples (i.e. of degree ≤ 3) into a single building block. This consequently also applies to the there used additive tuple entries, further lowering the tuple size. Note that for simplicity, this optimization is not depicted in Algorithm 4.3 even though it is considered in the results.

Algorithm 4.3 Algorithm to determine the arithmetic tuple size and bandwidth for a given multilinear polynomial.

```
procedure computeArithmeticTupleSize($polynomial$)
    $Tuple \leftarrow \emptyset$
    $BuildingBlocks \leftarrow \{(x_i - \text{Mask}(x_i)) \mid x_i$ occuring in a multiplication in the polynomial$\}$
    for all $monomials\ m$ in $polynomial$ do
        buildTree(new Node($m, null$), $m$, 0)
    end for
    Output $|Tuple|, |BuildingBlocks|$
end procedure
```

This algorithm is optimal regarding tuple size for monomials of degree $\leq$ 12. However, for monomials of degree $>$ 12, there is a small optimization for the split routine and possibly further optimizations, which are not implemented. In some cases, the only prefactor a should be multiplied to the right half instead of the left so that this building block can be reused to build a monomial with two prefactors. For monomials of degree $\leq$ 16, this reduces the tuple size by at most eight and at most two in the bandwidth.

There are several other possibilities for modification of the algorithm. For one, depending on the concrete implementation and the size of the monomials, it could increase the performance to check whether buildTree has already been called on a similar node. This node can then be reused, like $x_3x_4 - a_{34}$ or $x_7x_8 - a_{78}$ in Figure 3.1.

Moreover some changes influence bandwidth and tuple size when optimizing for different properties. One possibility is to modify the termination condition so that, e.g. monomials of degree ≤ 4 are considered a leaf and thus directly built using Binomial Tuples. Another option is to change the splitting strategy in thesplitMonomial function so that the monomial is cut at a different position or into more parts. The current version is optimized for the tuple size of a single monomial. However, when working with a polynomial of several monomials, different splitting strategies could increase the reusability of building blocks and thus decrease the tuple size.

For instance, if we look at the polynomial $f(x_0, \ldots, x_4) = x_0x_1x_2x_3x_4 + x_0x_1x_2x_3$. By default, $x_0x_1x_2x_3x_4$ would be split into the elementary building blocks $x_0x_1x_2$ and x_3x_4 none of which can be reused to build $x_0x_1x_2x_3$. In that case, we get 21 tuple entries for $x_0x_1x_2x_3x_4$ and additional eight entries ($a_0, \ldots, a_3, a_{01}$ can be reused) for $x_0x_1x_2x_3$, which leads to a total tuple size of 29 (when not combining the highest additive tuple entries of each of the two monomials). In comparison, if $x_0x_1x_2x_3x_4$ is split into the building blocks $x_0x_1x_2x_3$ and x_4, the required tuple size for this monomial increases from 21 to 26. However, the total tuple size stays at 26 since we have already built $x_0x_1x_2x_3$, decreasing the total tuple size for f by three. This effect will also be considered later on when we look at the polynomials of multiplexers.

Algorithm 4.4 Implementation of the buildTree procedure used in Algorithm 4.3.

```
procedure buildTree($node, monomial, mask$)
    if $degree \leq 3$ then
        addLeaf($node, monomial, mask$)                          // see Algorithm 4.5
        return
    end if
    $leftMonomial, rightMonomial \leftarrow$ splitMonomial($monomial$)
    $leftMask \leftarrow$ new Mask($leftMonomial$)
    $rightMask \leftarrow$ new Mask($rightMonomial$)
    $multNode \leftarrow$ new MultNode($node$)
    $multNode.children[0] \leftarrow$ new Node($leftMonomial - leftMask, multNode$)
    $multNode.children[1] \leftarrow$ new Node($rightMonomial - rightMask, multNode$)
    buildTree($multNode.children[0], leftMonomial, leftMask$)
    buildTree($multNode.children[1], rightMonomial, rightMask$)
    $addNode \leftarrow$ new AddNode($node$)
    $addNode.children[0] \leftarrow$ new Node($-leftMask \cdot rightMask - mask, addNode$)
    $highestAN \leftarrow$ findHighestAddNode($addNode$)
    $highestAN.tuple \leftarrow highestAN.tuple - leftMask \cdot rightMask - mask$
    $highestAN.buildingBlock \leftarrow highestAN.buildingBlock - leftMask \cdot rightMask - mask$
    $addNode.children[1] \leftarrow$ new Node($leftMask \cdot rightMonomial, addNode$)
    $addNode.children[2] \leftarrow$ new Node($rightMask \cdot leftMonomial, addNode$)
    buildTree($addNode.children[1], leftMask \cdot rightMonomial, 0$)
    buildTree($addNode.children[2], rightMask \cdot leftMonomial, 0$)
    $Tuple \leftarrow Tuple \cup \{addNode.tuple\}$
    $BuildingBlocks \leftarrow BuildingBlocks \cup \{addNode.buildingBlock\}$
end procedure

function splitMonomial($monomial$)
    Input $monomial = a \cdot b \cdot \prod_{i=0}^{k} x_i$        // $b = 1$ fore one prefactor, $a, b = 1$ for no prefac.
    return $a \cdot \prod_{i=0}^{\lfloor k/2 \rfloor} x_i, \quad b \cdot \prod_{j=\lfloor k/2 \rfloor+1}^{k} x_j$
end function

function findHighestAddNode(node)
    $currentlyHighestAddNode \leftarrow node$
    while $node$ has a parent do
        $node \leftarrow node.parent$
        if $node$ is a MultNode then
            return $currentlyHighestAddNode$
        else if $node$ is an AddNode then
            $currentlyHighesAddNode \leftarrow node$
        end if
    end while
    return $currentlyHighesAddNode$
end function
```

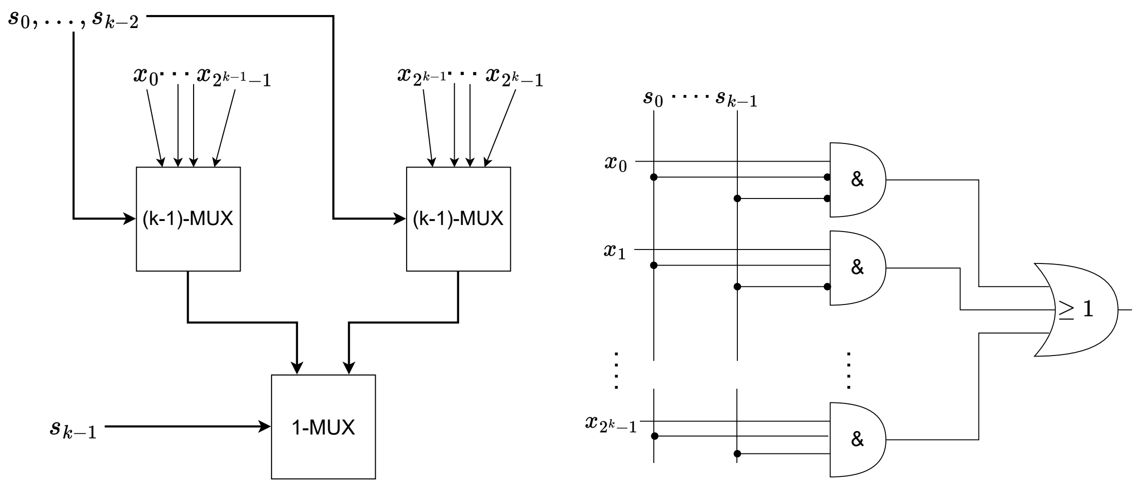
Algorithm 4.5 Implementation of the addLeaf procedure used in Algorithm 4.4.

```
procedure addLeaf($node, monomial, mask$)
    Input $monomial = p \cdot \prod_{i=0}^{k} x_i, k \in \{1,2\}$            // $p = 1$ in case of no prefactors
    $AddNodehighestAN$
    if $node.parent$ is a $AddNode$ then
        $highestAN \leftarrow$ findHighestAddNode($addNode$)
        $highestAN.buildingBlock \leftarrow highestAN.buildingBlock + monomial - mask$
    else
        $BuildingBlock \leftarrow BuildingBlock \cup \{(monomial - mask)\}$
    end if
    $a_i \leftarrow$ Mask($x_i$), $0 \le i \le k$
    if $k = 1$ then
        $Tuple \leftarrow Tuple \cup \{pa_0, pa_1\}$
        if $node.parent$ is a $AddNode$ then
            $highestAN.tuple \leftarrow highestAN.tuple + pa_0a_1$
        else
            $Tuple \leftarrow Tuple \cup \{pa_0a_1\}$
        end if
    else if $k = 2$ then
        $Tuple \leftarrow Tuple \cup \{pa_0, pa_1, pa_2, pa_0a_1, pa_0a_2, pa_1a_2\}$
        if $node.parent$ is a $AddNode$ then
            $highestAN.tuple \leftarrow highestAN.tuple + pa_0a_1a_2$
        else
            $Tuple \leftarrow Tuple \cup \{pa_0a_1a_2\}$
        end if
    end if
                                                // Add $mask, p$ to the tuple only if they are present.
    if $mask \neq 0$ then
        $Tuple \leftarrow Tuple \cup \{mask\}$
    end if
    if $p \neq 1$ then
        $Tuple \leftarrow Tuple \cup \{p\}$
    end if
end procedure
```

4.2 Multiplexer

MUXs are a powerful element in an arithmetic circuit. They can be used to implement array accesses or branches in a program to calculate functions of the form:

$$f(x) = \begin{cases} f_1(x) & \text{if condition}_1 \\ \vdots & \\ f_n(x) & \text{if condition}_n \end{cases} \tag{4.1}$$



(a) Recursive definition a k-MUX using two $(k-1)$-MUX and one 1-MUX.

(b) Circuit diagram of a k-MUX using AND and OR gates.

Figure 4.1: Visualisation of two approaches on how to find a polynomial for a k-MUX.

This alone is instrumental for branching programs that are, for example, derived from a state machine. Also, in privacy-preserving machine learning MUXs can be used. For example, a Maxpool function could be implemented by first applying a procedure that compares the inputs to determine which has the greatest value and then using a MUX to select that input. In Section 4.2.3, we will further review a specialised use case for multiplexers and its properties regarding SMPC.

4.2.1 Multiplexers in arithmetic circuits

In this work, we will define a k-MUX as multiplexer with k binary selection variables $s_0, \dots, s_{k-1}$ and thus 2^k data variables $x_0, \dots, x_{2^k-1}$. This gives the function

(4.2) $k\text{-MUX}(s_0, \dots, s_{k-1}, x_0, \dots, x_{2^k-1}) = x_i \text{ for } i = (s_{k-1} \dots s_1 s_0)_2$

Note that here s_0 is the Least Significant Bit (LSB) and s_{k-1} the Most Significant Bit (MSB). Therefore e.g. $2\text{-MUX}(s_0, s_1, x_0, \dots, x_3)$ with $s_0 = 0$ and $s_1 = 1$ gives x_2.

To compute the result of a k-MUX using additive secret sharing, it is necessary to find a polynomial f with $f(s_0, \dots, s_{k-1}, x_0, \dots, x_{2^k-1}) = k\text{-MUX}(s_0, \dots, s_{k-1}, x_0, \dots, x_{2^k-1})$. For which, a first approach is to find a polynomial for

(4.3) $$1 - \text{MUX}(s_0, x_0, x_1) = \begin{cases} x_0 & \text{if } s_0 = 0 \\ x_1 & \text{if } s_0 = 1 \end{cases}$$

leading to $1\text{-MUX}(s_0, x_0, x_1) = (1 - s_0)x_0 + s_0 x_1 = x_0 - s_0(x_0 - x_1)$. There are two different intuitions on extending this to a general k-MUX. On the one hand, we can combine two $(k-1)$-MUX and a 1-MUX to obtain a k-MUX as in Figure 4.1a. This gives the simple recursive definition:

$$\begin{aligned} &(4.4) \quad k\text{-MUX}(s_0, \dots, s_{k-1}, x_0, \dots, x_{2^k-1}) = 1\text{-MUX}(s_{k-1}, A, B) \\ &(4.5) \quad \phantom{k\text{-MUX}(s_0, \dots, s_{k-1}, x_0, \dots, x_{2^k-1})} = A - s_{k-1}(A - B) \end{aligned}$$

with

$$A = (k-1)\text{-MUX}(s_0, \ldots, s_{k-2}, x_0, \ldots, x_{2^{k-1}-1})$$

and

$$B = (k-1)\text{-MUX}(s_0, \ldots, s_{k-2}, x_{2^{k-1}}, \ldots, x_{2^k-1})$$

On the other hand, the basic circuit of a MUX 4.1b using AND and OR gates can also give an intuition on how to find a polynomial for a k-MUX. Here, for each data variable x_i there is an AND gate with inputs x_i and s_j or $1 - s_j$ for $0 \le j < k$ so that the output is x_i if $i = (s_{k-1} \ldots s_1 s_0)_2$ and 0 if $i \neq (s_{k-1} \ldots s_1 s_0)_2$. These AND gates can be expressed in a polynomial $g_i(s_0, \ldots, s_{k-1}, x_i)$ as

$$g_i(s_0, \ldots, s_{k-1}, x_i) = (S_0 \cdot S_1 \cdot \ldots \cdot S_{k-1} \cdot x_i) \tag{4.6}$$

with $S_j = s_j$ if $i_j = 1$ and $S_j = (1 - s_j)$ if $i_j = 0$ where $i = (i_{k-1} \ldots i_j \ldots i_0)_2$ (i_j is the j'th bit in the binary representation of i). Because for all $s = (s_{k-1} \ldots s_1 s_0)_2$ only g_s can be unequal to zero, we can form the sum over all g_i to express the final OR gate. Thus we get the definition of k-MUX as:

$$f_k = k\text{-MUX}(s_0, \ldots, s_{k-1}, x_0, \ldots, x_{2^k-1}) = \sum_{i=0}^{2^k-1} g_i \tag{4.7}$$

By this definition for example a polynomial of a 2-MUX is then: $2\text{-MUX}(s_0, s_1, x_0, x_1, x_2, x_3) = ((1 - s_0)(1 - s_1)x_0) + (s_0(1 - s_1)x_1) + ((1 - s_0)s_1 x_2) + (s_0 s_1 x_3)$

After simplifying both approaches result in the same polynomial, which can be used as an input to the Algorithms 4.3 and 4.1 to compute the required tuple sizes and bandwidths.

The second definition is more complex, but it is still helpful because some essential properties of these polynomials are easier to see with the second approach. Most importantly for all $0 \le i < 2^k$ in the product $g_i = S_0 \cdot S_1 \cdot \ldots \cdot S_{k-1} \cdot x_i$ each s_j $(0 \le j < k)$ additional to x_i is included exactly once. So the degree of each variable can be at most one, and thus $\sum_{i=0}^{2^k-1} g_i$ is indeed a multilinear polynomial. Furthermore, in each g_i, the monomial $s_0 \cdot \ldots \cdot s_{k-1} x_i$ appears and is included in the polynomial f_k.

We can use this fact to find the binomial tuple for f_k by defining the tuple for $\sum_{i=0}^{2^k-1} s_0 \cdot \ldots \cdot s_{k-1} x_i$ as:

$$A = \left(\left[a_i b_0^{e_0} \cdot \cdots \cdot b_{k-1}^{e_{k-1}}\right] \mid e_0, \ldots, e_{k-1} \in \{0,1\}, 0 \le i < 2^k\right) \tag{4.8}$$

$$\cup \left(\left[b_0^{e_0} \cdot \cdots \cdot b_{k-1}^{e_{k-1}}\right] \mid e_0, \ldots, e_{k-1} \in \{0,1\}, \neg e_0 = \cdots = e_{k-1} = 0\right) \tag{4.9}$$

Here we denote the masks for the x_i as a_i and the masks for the s_i as b_i. Now let h be any monomial in f_k. Then h is of the form $s_0^{j_0} \cdot \ldots \cdot s_{k-1}^{j_{k-1}} x_i$ for a $0 \le i < 2^k$ and $j_0, \ldots, j_{k-1} \in \{0,1\}$. To compute h using Binomial Tuples, each tuple entry is then of the form $b_0^{e_0} \cdot \cdots \cdot b_{k-1}^{e_{k-1}} a_i^{e_k}$ with $e_i = 0$ if $i < k, j_i = 0$ and $e_i \in \{0,1\}$ else. Therefore, it is also in A; consequently, A is the binomial tuple for f_i. We can then define the binomial tuple size as

$$|A| = 2^k \cdot 2^k + 2^k - 1 = (2^k + 1)2^k - 1 \tag{4.10}$$

The second definition based on Figure 4.1b additionally hints at a different version of a polynomial f'_k for a k-MUX. Instead of defining a polynomial $f_k(s_0, \cdots, s_{k-1}, x_0, \ldots x_{2^k-1})$ we use a polynomial $f'_k(s_0, \cdots, s_{k-1}, \bar{s}_0, \cdots, \bar{s}_{k-1}, x_0, \ldots x_{2^k-1})$ with $\bar{s}_i = (1-s_i), 0 \leq i < k$. This has the disadvantage, that the bandwidth increases by k, because additionally the masked values $(\bar{s}_i - c_i) = ((1-s_i) - c_i)$ have to be opened. With these values, however, it is then possible to compute

(4.11)
$$k\text{-MUX} = f'_k(s_0, \cdots, s_{k-1}, \bar{s}_0, \cdots, \bar{s}_{k-1}, x_0, \ldots x_{2^k-1})$$
(4.12)
$$= \bar{s}_0 \cdot \ldots \cdot \bar{s}_{k-1} x_0 + s_0 \bar{s}_1 \cdot \ldots \cdot \bar{s}_{k-1} x_1 + \cdots + s_0 \cdot \ldots \cdot s_{k-1} x_{2^k-1}$$

In contrast to Equation (4.7), f'_k consists of exactly 2^k monomials of degree $k+1$ without any additional lower degree terms. This reduction in the number of monomials is promising regarding arithmetic tuple size. As for binomial tuple size, similar to Equations (4.8) and (4.9) we get $2^k \cdot 2^k + 2^k - 1$ tuple entries $\left[a_0 c_0^{e_0} \cdot \cdots \cdot c_{k-1}^{e_{k-1}}\right], \ldots, \left[a_{2^k-1} b_0^{e_0} \cdot \cdots \cdot b_{k-1}^{e_{k-1}}\right]$ for $e_0, \ldots, e_{k-1} \in \{0,1\}$ and $\left[b_0^{e_0} \cdot \cdots \cdot b_{k-1}^{e_{k-1}}\right]$for $e_0, \ldots, e_{k-1} \in \{0,1\}, \neg e_0 = \cdots = e_{k-1} = 0$. Apart from these $2^k \cdot 2^k + 2^k - 1$ tuple entries, there are further tuple entries like $[c_0 \cdot \cdots \cdot c_{k-1}]$ so the binomial tuple size for f'_k is greater than the tuple size for f_k and it is better to use f_k (in case of Binomial Tuples).

As already mentioned depending on the used circuit, we get different results with beaver triples. Instead of using the general algorithm for any polynomial, it is also possible to use a circuit based on Figure 4.1a. Each 1-MUX in this circuit requires one multiplication to compute $x - s(x-y)$ on the inputs x, y and s. In detail for the SMPC setting, each 1-MUX gets a share of x, and the previously opened masked values $(x-y-a), (s-b)$ as inputs and computes $[x] - (s-b)(x-y-a) - [b](x-y-a) - [a](s-b) - [ab]$. For a k-MUX, we have k phases corresponding to the k levels in the circuit. On each level, all 1-MUXs have the same s_i as input but different x, y. This means that for each 1-MUX, the tuple entries $[a]$ and $[ab]$ are unique, whereas $[b]$ is the same within each level/phase. So for the $2^k - 1$ 1-MUXs and k levels the tuple size is $2 \cdot (2^k - 1) + k$ and the bandwidth is $\frac{2^k}{2} + k + 2^k - 1$ to publish the $\frac{2^k}{2} + k$ initial masked values plus the $2^k - 1$ results of the 1-MUXs.

Until now, we only considered MUXs with 2^k data variables for a $k \in \mathbb{N}$ because naturally k selection variables enable selecting between 2^k values. Despite this, it is possible to use a k-MUX on m data variables with $m < 2^k$ by setting $x_i = 0$ for $i \geq m$. This achieves that for $(s_{k-1} \ldots s_0)_2 \geq m$ $k\text{-MUX}(s_0, \ldots, s_{k-1}, x_0, \ldots, x_{m-1}, 0 \ldots, 0)$ is just zero. For example, if we want to select between nine data variables, a $\lceil \log 9 \rceil = 4$-MUX is required. Whereas the polynomial of a general 4-MUX contains 81 monomials, the polynomial $4\text{-MUX}(s_0, s_1, s_2, x_0, \ldots, x_8, 0, \ldots, 0)$ consists of only 62 monomials since each monomial in which a x_i with $i \geq m$ appears is eliminated.

4.2.2 Multi round evaluation

For MUXs with several thousand or even millions of inputs, the tuple size can become excessively large and slow down the computation. If it is necessary to handle such a high amount of inputs, it could be advisable to accept a slightly higher round complexity. To illustrate this, we will build a 16-MUX using Arithmetic Tuples and only one additional round. The procedure will be as follows:

In the initial opening round, publish the masked inputs. Then use these to locally compute the elementary building blocks for 2^{16-k} $(16-k)$-MUXs. Each of these MUXs has $s_0, \dots, s_{16-k-1}$ but different x_i as inputs, which still gives the potential for reusing building blocks. To ensure that no additional information is leaked, add a new mask d_j to one of the highest additive building blocks for each $(16-k)$-MUX before publishing them. Afterwards, each party can now build

$$M_j - d_j = (16-k)\text{-MUX}(s_0, \dots, s_{16-k-1}, x_{j2^{16-k}}, \dots, x_{(j+1)2^{16-k}-1}) - d_j$$

for $0 \le j < 2^{16-k}$.

Using these these it is then possible to build k-MUX$(s_{16-k}, \dots, s_{16-1}, M_0, \dots, M_{2^{16-k}-1})$ with Arithmetic Tuples which requires only one additional round.

To determine the tuple size, we can apply Algorithm 4.3 to each polynomial of the $(16-k)$-MUXs using the same tuple set and then add the tuple size for the k-MUX which already includes the masks d_j. The bandwidth can also be determined in the same way.

4.2.3 Permutations

Apart from the mentioned direct use cases of MUXs, they can also serve more specialized purposes. One area of functions that is relevant to examine is functions with more than one output of the form $f(x) = (f_0(x), \dots f_n(x))$. This allows reusing building elements among the different functions $f_i, 0 \le i \le n$. To determine the tuple size and bandwidth of a function f, with several outputs Algorithms 4.1 to 4.3 can be applied to each f_i separately but using the same set to store tuple entries and opened building blocks.

Especially when for each $0 \le i \le n$: the polynomial f_i is the polynomial of a multiplexer, because of the similar structure, the potential for reusing whole monomials seems promising.

Here we consider permutations (without repetition) over n variables $x_0, \dots x_{n-1}$. The goal is to define a permutation function p that gets the n variables and some selection variables as input so that it is possible to select between all $n!$ possible permutations of the n variables. Using binary selection variables s_i as used with MUXs $\lceil \log n! \rceil$ variables are necessary. Let $s = (s_{\lceil \log n! \rceil - 1} \dots s_0)_2$.

To make it unambiguous which s leads to which permutation, we will define the s'th permutation as the s'th permutation in lexicographic ordering. Note that there are more efficient methods to generate permutations that are not in lexicographic ordering, like Heap's Algorithm [7, 12], that only switch two elements per iteration.

Depending on s, each p_i should output a specific variable x_j, making it an obvious task for MUXs where the same variable occurs multiple times at an input. In the polynomial, this will lead to some monomials occurring multiple times, which either cancel out or can be summed up. These thereby arising coefficients only require multiplication with a constant and thus do not increase the tuple size. To illustrate the concrete construction, Table 4.1 shows the output of p for all six possible values of s in the case of permutation of three variables. In this example, the 3! potential values require three bits to encode s and a 3-MUX for each output. The inputs to each MUX can be taken from the three columns in the table giving:

$p_0(s_0, s_1, s_2, x_0, x_1, x_2) = 3\text{-MUX}(s_0, s_1, s_2, x_0, x_0, x_1, x_1, x_2, x_2, 0, 0) = x_0 - s_1x_0 + s_1x_1 - s_2x_0 + s_2x_2 + s_1s_2x_0 - s_1s_2x_1 - s_1s_2x_2$

s	$p_0(s,x)$	$p_1(s,x)$	$p_2(s,x)$
0	x_0	x_1	x_2
1	x_0	x_2	x_1
2	x_1	x_0	x_2
3	x_1	x_2	x_0
4	x_2	x_0	x_1
5	x_2	x_1	x_0

Table 4.1: Outputs of $p(s,x)$ so that p gives all permutations of three variables $x = (x_0, x_1, x_2)$ in lexicographic order for $0 \leq s < 3! = 6$.

k	$k!$	Used MUX	# unique monomials in p	# monomials general MUX
2	2	1-MUX	4	3
3	6	3-MUX	20	27
4	24	5-MUX	102	243
5	120	7-MUX	505	2187
6	720	10-MUX	5180	59049
7	5040	13-MUX	51943	1594323
8	40320	16-MUX	$*$	43046721

Table 4.2: Number of possible permutations $k!$, MUXs used to describe $p_0, \dots, p_{k!-1}$, number of unique monomials in p and number of monomials in one general MUX used in p for permutations of k variables. ($*$ takes too long to compute)

$p_1(s_0, s_1, s_2, x_0, x_1, x_2) = \text{3-MUX}(s_0, s_1, s_2, x_1, x_2, x_0, x_2, x_0, x_1, 0, 0) = x_1 - s_0x_1 + s_0x_2 + s_1x_0 - s_1x_1 + s_2x_0 - s_2x_1 - s_0s_1x_0 + s_0s_1x_1 - s_0s_2x_0 + 2 \cdot s_0s_2x_1 - s_0s_2x_2 - 2 \cdot s_1s_2x_0 + s_1s_2x_1 + 2 \cdot s_0s_1s_2x_0 - 2 \cdot s_0s_1s_2x_1$

$p_2(s_0, s_1, s_2, x_0, x_1, x_2) = \text{3-MUX}(s_0, s_1, s_2, x_2, x_1, x_2, x_0, x_1, x_0, 0, 0) = x_2 + s_0x_1 - s_0x_2 + s_2x_1 - s_2x_2 + s_0s_1x_0 - s_0s_1x_1 + s_0s_2x_0 - 2 \cdot s_0s_2x_1 + s_0s_2x_2 - s_1s_2x_1 - 2 \cdot s_0s_1s_2x_0 + 2 \cdot s_0s_1s_2x_1$

These polynomials $p_0, \dots, p_2$ are based on the definition of multiplexers without publishing $(1 - s_i)$ separately because this allows reusing more monomials and will lead to smaller tuple sizes. In total, p consists of only 20 unique monomials where two are of degree four, whereas a general 3-MUX consists of 27 where eight are of degree four. So even though three 3-MUXs are used, the number of monomials stays in comparison low. The same scheme can be applied to any number of variables to find polynomials for each output. Table 4.2 shows this comparison for permutations of up to eight variables.

The number of general permutations grows factorial, leading to zeros and variables occurring several times in the inputs of the MUXs. This results in a rapid increase in the multiplexer size. Permutations of only eight variables already require a 16-MUX for each output. To avoid this, it is possible to restrict the set of permutations to only rotating the position of the variables; then, the number grows linearly with the number of variables. Therefore for 2^k variables, no zeroes or duplicate variables are necessary to find a polynomial using MUXs. Also, many other variations are imaginable, like shifts, where zeroes are shifted in.

4.3 Demultiplexer

The counterpart of a MUX is a DEMUX which has one data input, multiple outputs and several selection bits deciding to which output the data is forwarded. It implements the function $f(s_0, \dots, s_{k-1}, x) = (f_0, \dots, f_{2^k-1})$ with

(4.13)

$$f_i(s_0, \dots, s_{k-1}, x) = \begin{cases} x & i = (s_{k-1}s_{k-2}\dots s_0)_2 \\ 0 & \text{else} \end{cases}, \quad 0 \le i < 2^k$$

A DEMUX's circuit diagram consists of one AND gate g_i for each output with inputs $x, s_0, \dots, s_{k-1}, \bar{s}_0, \dots \bar{s}_{k-1}$. The polynomials for f_i can then be defined as $f_i = g_i = S_0 \cdot \dots \cdot S_{k-1}$ with $S_j = s_j$ if $i_j = 1$ and $S_j = \bar{s}_j$ if $i_j = 0$ where $i = (i_{k-1} \dots i_j \dots i_0)_2$.

To compute these products in a SMPC setting, on the one hand, we can publish x, s_i and $\bar{s}_i = (1 - s_i)$ as masked values and thus get for each function exactly one unique monomial. Or on the other hand, it is possible to publish only x and s_i. Then $\bar{s}_i$ is replaced with $(1 - s_i)$ in the polynomial after simplifying, giving at most 2^k terms in each monomial. For two selection bits and four outputs, this gives the polynomials:

$$\begin{aligned} f_0 &= \bar{s}_0 \cdot \bar{s}_1 \cdot x = (1 - s_0)(1 - s_1)x = x - s_0x - s_1x + s_0s_1x \\ f_1 &= s_0 \cdot \bar{s}_1 \cdot x = s_0(1 - s_1)x = s_0x - s_0s_1x \\ f_2 &= \bar{s}_0 \cdot s_1 \cdot x = (1 - s_0)s_1x = s_1x - s_0s_1x \\ f_3 &= s_0s_1x \end{aligned}$$

At first, the first approach seems more promising since we only get one term in each polynomial. However, since $f_0 = (1 - s_0) \cdot \dots \cdot (1 - s_{k-1}) \cdot x$ already includes all monomials that appear in the other polynomials, the same number of monomials need to be computed, of which most are of lower degree.

The tuple size of Binomial Tuples for the second approach is the same as for one degree $k + 1$ monomial $s_0 \cdot \dots \cdot \cdot_{k-1} x$, which is $2^{k+1} - 1$. The first approach gives several degrees $k + 1$ monomials leading to a larger tuple size for binomial tuples.

4.4 Prefix Products

The prefix product of a monomial $x_0 \cdot \dots \cdot x_{n-1}$ gives the terms $x_0, x_0x_1, x_0x_1x_2, \dots, x_0 \cdot \dots \cdot x_{n-1}$. Therefore it is also a function with multiple outputs. Prefix Products are, for example, used in some protocols that compare two bit-decomposed values. In [11], such a protocol and also a method to compute shares of prefix products of a monomial $x_0 \cdot \dots \cdot x_{2^k-1}$ is discussed.

Algorithm 4.6 This algorithm outputs index pairs (i, j) of products $x_i \cdot \dots \cdot x_j$ that can be combined to obtain the prefix product $x_0 \cdot \dots \cdot x_{k-1}$ of a product $x_0 \cdot \dots \cdot x_{n-1}$ with $k \leq n$.

```
procedure DECOMPOSEPREFIX($k$)
    $i \leftarrow 0$
    $j \leftarrow 0$
    while $i < k$ do
        $j \leftarrow i + 2^{\lfloor \log(k-i) \rfloor} - 1$
        output$(i, j)$
        $i \leftarrow j + 1$
    end while
end procedure
```

For example, to compute the prefix products of $x_0 \cdot \dots \cdot x_7$ first $[x_0 \cdot \dots \cdot x_7]$ is computed using Arithmetic Tuples as described in Section 3.4. This also gives the masked values $(x_0 - a_0), \dots, (x_{2^k-1} - a_{2^k-1}), (x_0x_1 - a_{01}), \dots, (x_6x_7 - a_{67}), (x_0 \cdot \dots \cdot x_3 - a_{0123}), (x_4 \cdot \dots \cdot x_7 - a_{4567})$. These then can then be used to locally compute the shares of all prefix products of $x_0 \cdot \dots \cdot x_7$:

$$
\begin{aligned}
[x_0] &= [x_0] \\
[x_0x_1] &= (x_0x_1 - a_{01}) + [a_{01}] \\
[x_0x_1x_2] &= (x_0x_1 - a_{01})(x_2 - a_2) \\
[x_0 \cdot \dots \cdot x_3] &= (x_0 \cdot \dots \cdot x_3 - a_{0123}) + [a_{0123}] \\
[x_0 \cdot \dots \cdot x_4] &= (x_0 \cdot \dots \cdot x_3 - a_{0123})(x_4 - a_4) \\
[x_0 \cdot \dots \cdot x_5] &= (x_0 \cdot \dots \cdot x_3 - a_{0123})(x_4x_5 - a_{45}) \\
[x_0 \cdot \dots \cdot x_6] &= (x_0 \cdot \dots \cdot x_3 - a_{0123})(x_4x_5 - a_{45})(x_6 - a_6) \\
[x_0 \cdot \dots \cdot x_7] &= [x_0 \cdot \dots \cdot x_7]
\end{aligned}
$$

The products of masked values can be computed with Binomial Tuples if the necessary tuple entries have been added to the tuple. As has been proven in [11], such a product is at most of the degree k as can be seen in the example for $k = 3$.

To determine the tuple size for an arbitrary k, we can first compute the tuple set for $x_0 \cdot \dots \cdot x_{2^k-1}$ using Algorithm 4.4 and then add the missing binomial tuple entries. To add the missing binomial tuple entries, it is necessary to find out which masked values from the previous step are to be combined to acquire a specific prefix. For this, Algorithm 4.6 can be used. The bandwidth is the same as the bandwidth to compute $x_0 \cdot \dots \cdot x_{2^k-1}$ since no further field elements are sent.

We want to compare this method to two more naive approaches: First, it is possible to compute the shares directly with Binomial Tuples, for which we need the tuple $\left(a_0^{d_0} \cdot \dots \cdot a_{2^k-1}^{d_{2^k-1}} \mid (d_0, \dots, d_{2^k-1}) \in \{0,1\}^{2^k} \setminus \{(0, \dots, 0)\}\right)$ of size $2^{(2^k)} - 1$. For this, the bandwidth is 2^k to open the initial masked values. Secondly, we can apply Algorithm 4.4 to each prefix product of $x_0 \cdot \dots \cdot x_{2^k-1}$ reusing the same tuple set.

5 Results

Now we will analyze how the applications for arithmetic tuples perform regarding the metrics tuple size, bandwidth and round complexity. For this, the methods presented in Chapter 4 are applied to measure these metrics for beaver triples, binomial tuples and arithmetic tuples. The presented algorithms were implemented in Java. We also need to compare different approaches in some cases, like the different function definitions for multiplexers.

5.1 Multilinear Monomials

First, we will give an overview of the metrics for computing the value (not only a share) of a single multilinear monomial $m_n = x_0 \cdot \dots \cdot x_{n-1}$ of degree n. For Beaver Triples, when using the same splitting strategy as in Algorithm 4.1 $n-1$ multiplications are necessary, and thus the tuple size is $3(n-1)$ the bandwidth $2(n-1)+1$ and the round complexity $\lceil \log n \rceil$. Following the results in Section 3.3, we need a binomial tuple of size $2^n - 1$, get a bandwidth of $n+1$ and a round complexity of one. With our example Arithmetic Tuple, only one round in addition to the opening round is necessary. To determine tuple size and bandwidth, we can use Algorithm 4.3 on the input m_n. The results are shown in Table 5.1.

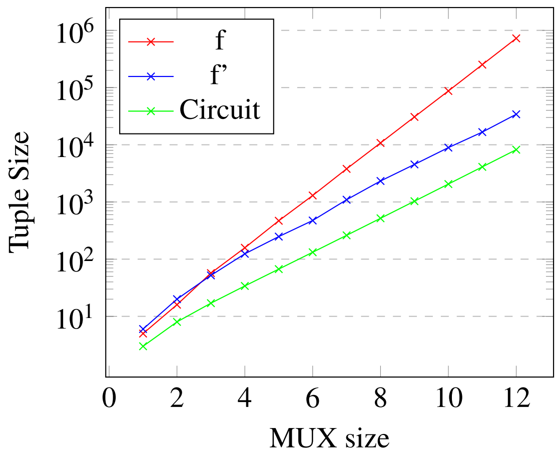
Using these values, we can later also see how many tuple entries were reused in Algorithm 4.3 on the input of a polynomial of several monomials.

5.2 Multiplexer

In Section 4.2, we introduced several possibilities to implement MUXs. We defined two different polynomials f_k and f'_k describing a k-MUX. The difference between f_k and f'_k was that for f' additionally, the negated inputs $\bar{s}_0, \dots, \bar{s}_{k-1}$ were used so that products including $(1-s_i)$ did not have to be simplified.

n	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Tuple Size	3	7	13	21	29	38	47	63	79	91	103	118	133	145	157
Bandwidth	3	4	7	8	9	13	17	18	19	20	21	27	33	38	43

Table 5.1: Tuple size and bandwidth to compute $x_0 \cdot \dots \cdot x_{n-1}$ using Arithmetic Tuples as determined by Algorithm 4.3.



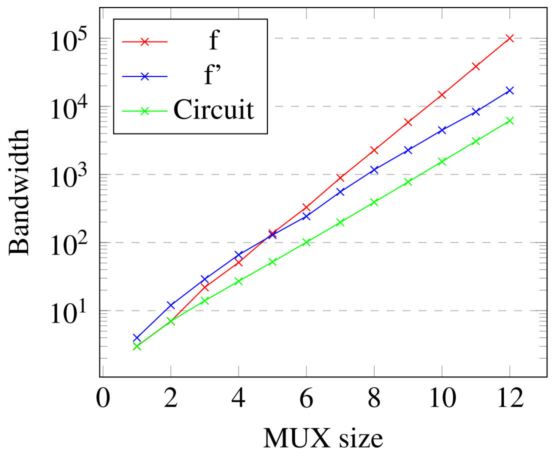


Figure 5.1: Line diagrams comparing the tuple size and bandwidth for MUXs using Beaver Triples.

For beaver triples, we also considered a third approach based on a circuit that implements a k-MUX using $2^k - 1$ 1-MUXs. The round complexity differs for the three versions. With the third approach, one round for each of the k layers in the circuit is necessary. For the two polynomials f_k and f'_k, $k + 1$ is the highest degree the monomials in f_k, f'_k have. So here we get a round complexity of $\lceil \log(k + 1) \rceil$, which is smaller than the k rounds of the third approach.

Figure 5.1 compares the tuple size and bandwidth of the three approaches. The tuple size and bandwidth for f_k and f'_k are determined by Algorithm 4.1, and for the circuit-based computation, the tuple size is $2(2^k - 1) + k$ and the bandwidth $2^{k-1} + k + 2^k - 1$. Here the circuit-based method performs best. For a 12-MUX, we get a tuple size of 725376 for f_k, 34138 for f'_k and 8202 for the circuit. The circuit-based approach thus leads to significantly smaller tuple sizes. Still, we will use f'_k to compare beaver triples to Binomial and Arithmetic Tuples because they perform better than f_k for larger MUXs and achieve a more minor round complexity than the circuit (4 vs 12 rounds for a 12-MUX). Both Binomial and Arithmetic Tuples minimize the round complexity, so choosing an approach with fewer rounds increases comparability.

For Binomial Tuples, we have already shown that using f_k leads to a smaller tuple size. Therefore we now compare the tuple size and bandwidth to compute f'_k using Beaver Triples, f_k using Binomial Tuples and f_k, f'_k using Arithmetic tuples. Furthermore, we also need to compare if using f_k or f'_k is better in the case of Arithmetic Tuples.

For this, Figure 5.2 compares the tuple size and bandwidth of Arithmetic Tuples (for f_k and f'_k, Binomial Tuples (for f_k) and Beaver Triples (for f'_k). As could be expected, Beaver Triples achieve the smallest tuple size by allowing a non-minimal round complexity. In the case of one-round protocols, Arithmetic Tuples for f_k are best for k-MUXs with $k \leq 3$ because no mask for $\bar{s}_i, 0 \leq i < 2$ is necessary and other than with Binomial Tuples the additive tuple entries are combined. But for higher values of k, Arithmetic Tuples for f'_k have the smallest tuple size. Generally, it can be observed that for higher values of k, the savings from arithmetic tuples increase compared to binomial tuples. For $k = 16$, the Binomial Tuple is of size 4295032831 and the Arithmetic Tuple for f'_k is only of size 6781756, which is a reduction by more than 99.84%.

k	1	2	3	4	5	6	7	8	9	10
$\lvert T_k \rvert$	6	28	104	336	928	2432	6016	16128	40448	93184
Alg. 4.3	5	21	77	232	633	1607	3859	10238	25778	57477
Reduction	17%	25%	26%	31%	32%	34%	36%	37%	36%	38%

Table 5.2: Tuple size to compute the polynomial f'_k with (Alg. 4.3) and without ($|T_k|$) reusing tuple elements among the monomials.

In a theoretical scenario where storing a field element like a tuple entry takes 8 Byte (64 Bit) and 8 MB of Random Access Memory (RAM) being available, it would be possible to store at most 10^6 elements. In this scenario, with Binomial Tuples, we could compute at most a 9-MUX, and with Arithmetic Tuples, a 13-MUX, which has 16 times more inputs.

The bandwidth is smallest for Binomial Tuples since here, only one field element in addition to the $2^k + k$ inputs is transmitted. Apart from that, Arithmetic Tuples for f'_k have a reduced bandwidth compared to Arithmetic Tuples for f_k even though for f'_k, the k additional values $\bar{s}_i$ have to be opened in the first opening round.

Based on MUXs, we now want to examine how significant the savings potential is through reusing and combining tuple elements over several monomials. For this, we calculate the tuple size if each monomial in f'_k would be built separately $|T_k|$ and compare it to the results of Algorithm 4.3. $|T_k|$ is given by the sum of the tuple sizes for each monomial in f'_k as they can be taken from Table 5.1. In Table 5.2, these values are displayed for $1 \leq k \leq 10$. The percentage reduction achieved is also given. This number increases with k, and for $k = 16$, it reaches 44%. For $k = 1$, the only reduction is achieved by combining the additive tuple entries of the two monomials. However, since the reduction through combining is at most $2^k - 1$, it can be seen that reusing does have a significantly positive effect on the tuple size.

A variation of Algorithm 4.4 was to use a different splitting strategy. Since in f_k each monomial is a product of selection variables s_i and one data variable x_i, it could be beneficial to separate the selection from the data variables so that the product of selection variables can be reused for multiplication with several data variables.

However, this only achieves a slight reduction in tuple size for $k = 3$ because for $k < 3$, the monomials are directly built using binomial tuples; therefore, this change in the splitting strategy does not matter. For $k > 3$, the penalty for not splitting the monomial in the middle outweighs the potential for reusing more tuple entries.

5.2.1 Multi round evaluation

As described in Section 4.2.2, we exemplary want to build a 16-MUX in two rounds using Arithmetic Tuples. In the second column of Table 5.3 is the tuple size to build the 2^{16-k} $(16 - k)$-MUXs in the first round. Here the reduction through reusing among the multiplexers is taken into account. The third column shows the tuple size for the single k-MUX in the second round, and the fourth column the total tuple size. The fourth column is the percentage reduction compared to the tuple size

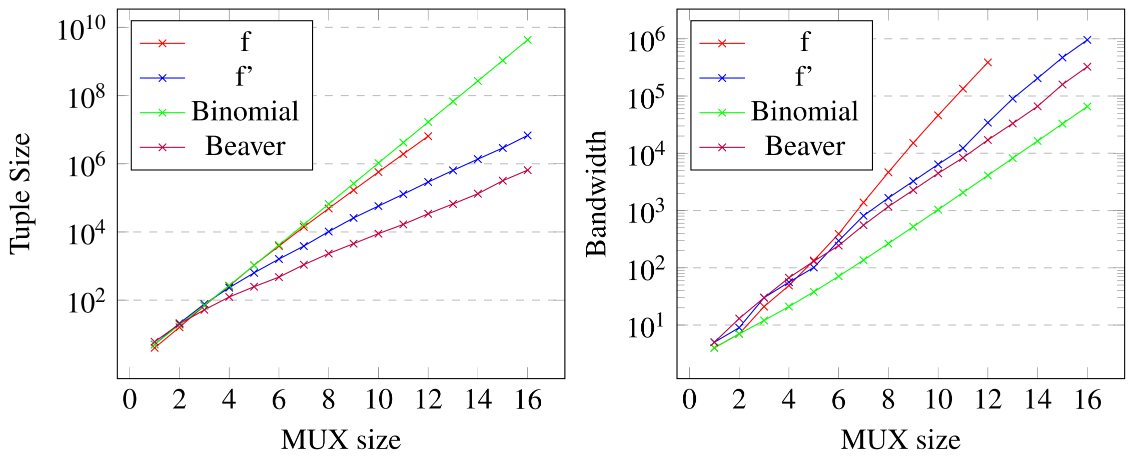


Figure 5.2: Line diagrams comparing the tuple size and bandwidth for MUXs using Beaver Triples, Binomial Tuples and Arithmeitc Tuples.

k	First Round	Second Round	Σ	Reduction	Bandwidth
1	4969274	5	4969279	27%	764985
2	4218652	21	4218673	38%	597646
3	3792776	77	3792853	44%	491332
4	3547520	232	3547752	48%	402952
5	3112176	633	3112809	54%	266317
6	2702532	1607	2704139	60%	264747
7	2431793	3859	2435652	64%	264161
8	1903613	10238	1913851	72%	264441
9	1377938	25778	1403716	79%	265697
10	1115654	57477	1173131	83%	203115
11	919736	128209	1047945	85%	143373
12	659527	291185	950712	86%	165352
13	401436	638849	1040285	85%	220980
14	213000	1367833	1580833	77%	269962
15	98306	2904889	3003195	56%	535607

Table 5.3: Tuple sizes to compute a 16-MUX in two rounds with Arithmetic Tuples. It also shows the reduction in comparison to one round and the bandwidth.

required to build a 15-MUX in a single round. Finally, the last column contains the total bandwidth, which consists of the $2^{16} + 2k$ initial masked values and the building blocks that are opened in the two rounds.

We obtain the best results in terms of tuple size for $k = 12$ when using 16 multiplexers of size four and a 12-MUX to combine them. By this, a reduction of 86% in tuple size compared to the one-round version is achieved. Also, the bandwidth is reduced by 83% from 955669 to 165352 but is lowest for $k = 11$

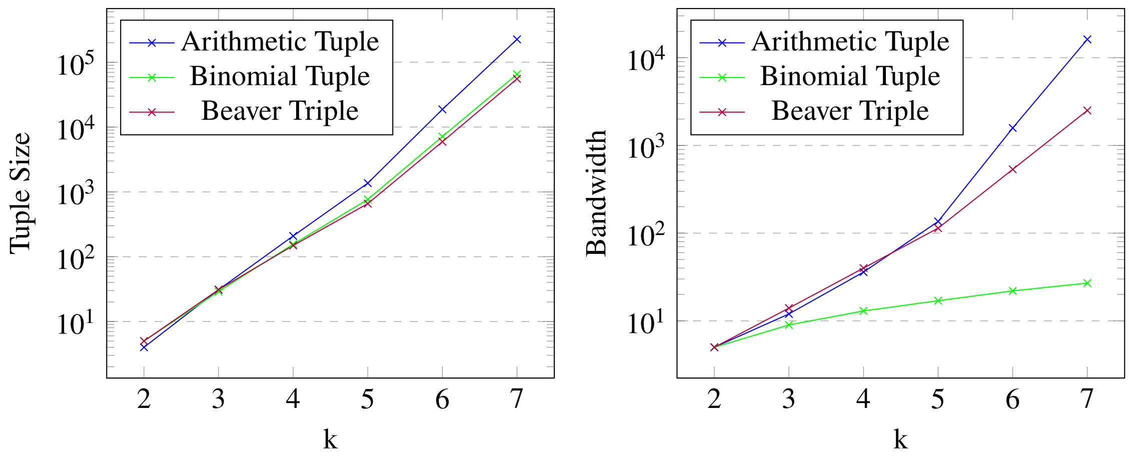


Figure 5.3: ,
Line diagrams comparing the tuple size and bandwidth for Permutations of k Variables using Beaver Triples, Binomial Tuples and Arithmetic Tuples.

5.2.2 Permutations

In the evaluation, we used Algorithms 4.1 to 4.3 (more specifically, a small variation for several polynomials) to find the tuple size and bandwidth to compute the polynomials p_i describing permutations. As already mentioned, using the definition f' for the MUXs' polynomials leads to larger tuples for Beaver Triples, Binomial and Arithmetic Tuples. Therefore in Figure 5.3 for each of the approaches, f is used to find a polynomial p_i for each of the outputs. Here, Binomial Tuples achieve smaller tuple sizes than arithmetic tuples and also have a significantly smaller bandwidth.

We also tested the mentioned rotations and shifts. For both Binomial Tuples performed better than Arithmetic Tuples in bandwidth and tuple size. All three functions have in common that the data variables x_i are input to multiple, and for rotations and permutations, even all, data input slots of the MUXs. Therefore in the polynomials arise more (or even all possible) products of selection variables s_i and one data variable x_i, explaining why Binomial Tuples outperform Arithmetic Tuples for these functions.

5.3 Demultiplexer

To determine the tuple sizes with Algorithms 4.1 to 4.3 for a k-DEMUX, we can either use the first approach with 2^k monomials of degree $k+1$ or build each monomial in $f_0 = (1-s_0)\cdot\dots\cdot(1-s_{k-1})\cdot x$ separately to reuse them for the other $2^k - 1$ functions. f_0 only leads to one monomial of degree $k + 1$ and $2^k - 1$ monomials of a lower degree. As could be expected, for Beaver Triples, Binomial- and Arithmetic Tuples for f_0 achieve smaller tuple sizes than the other possibility. Therefore, we will only compare tuple sizes and bandwidth using f_0 in detail. The round complexity is, of course, one for Binomial- and Arithmetic Tuples and $\lceil \log k + 1 \rceil$ for Beaver Triples. Figure 5.4 shows the tuple size and bandwidth for DEMUXs of size one to twelve. Notable here is that the tuple size of Beaver Triples is greater than the tuple size of Binomial Tuples for $k \leq 6$. The higher potential

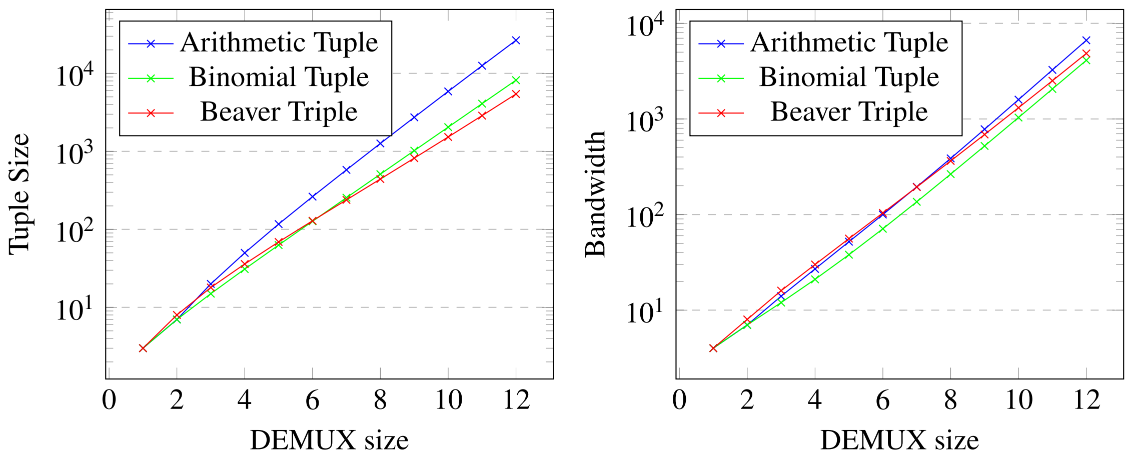


Figure 5.4: Line diagrams comparing the tuple size and bandwidth for DEMUXs using Beaver Triples, Binomial Tuples and Arithmetic Tuples.

for reusing tuple entries of Binomial Tuples outweighs the smaller tuple size to compute a single monomial with Beaver Triples. For similar reasons, also Arithmetic Tuples lead to larger tuples for all $k \leq 12$, and the difference is even increasing.

5.4 Prefix Products

Figure 5.5 shows the tuple size and bandwidth for prefix products determined as described in Section 4.4. The approach that combines Arithmetic and Binomial Tuples achieves the smallest tuple size. In contrast, the Binomial Tuple's size grows exponentially in the number of variables and quickly becomes impractical. Binomial Tuples, however, lead to the smallest bandwidth and to get shares of the prefix products, no additional round to the initial round is necessary. Note that here for each approach, the given bandwidth does not include opening the shares of the results. With Arithmetic Tuples, it is possible to get shared and public values after one round. However, with the combined approach, the shares obtained after one round would still have to be opened to receive the absolute values.

When opening the prefix products of $x_0 \cdot \cdots \cdot x_{2^k-1}$, it is possible to calculate the inputs x_i for $0 \leq i < 2^k$ by solving $x_i = \frac{x_0 \cdot \cdots \cdot x_i}{x_0 \cdot \cdots \cdot x_{i-1}}$. Therefore in most cases, obtaining shares is sufficient, and then the Arithmetic Tuples and the combined approach lead to the same round complexity. In this case, it is better to use the combined approach from [11] because of the smaller tuple size and bandwidth.

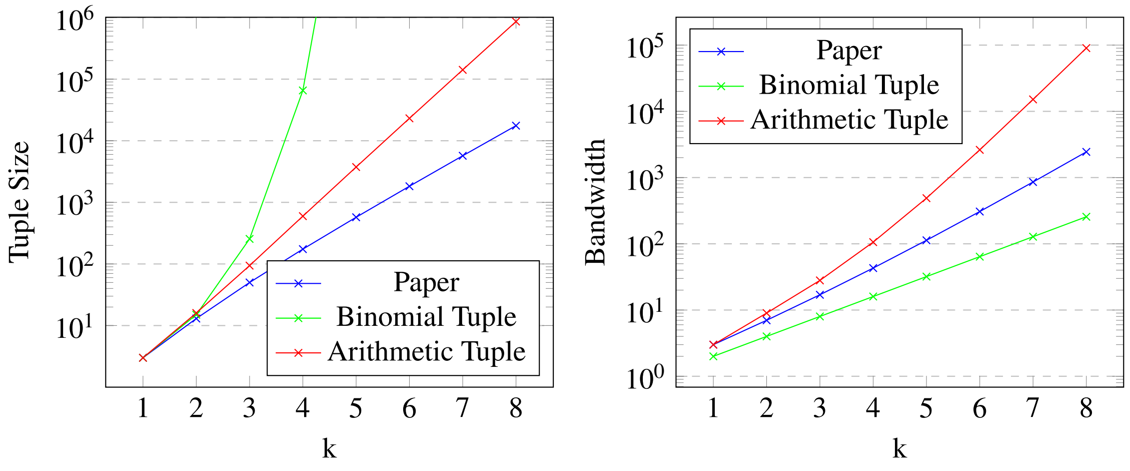


Figure 5.5: Line diagrams comparing the tuple size and bandwidth for the prefix product of $x_0 \cdot \cdots \cdot x_{2^k-1}$ using the method from the paper [11], Binomial Tuples and Arithmetic Tuples.

6 Conclusion

In this work, we analyzed several possible applications for Arithmetic Tuples regarding important metrics in SMPC. The possibility and speed of carrying out a calculation depends on the size of the correlated randomness (tuple size), the number of field elements transmitted (bandwidth) and the number of rounds (round complexity). We used the here presented algorithms to measure these metrics or found closed formulas.

One important application were multiplexers, which can be used in many different ways. For multiplexers, we could show that Arithmetic Tuples achieve a significantly smaller tuple size compared to the other one-round protocol, Binomial Tuples. Whereas Binomial Tuples perform better in terms of bandwidth. Here we get a typical trade-off between tuple size, bandwidth and round complexity. For the smallest tuple size with no requirements for minimal round complexity, it is best to use the circuit-based approach with Beaver Triples. For minimal rounds and a moderate Tuple size, it is best to choose Arithmetic Tuples, and for minimal rounds and minimal bandwidth, Binomial Tuples are most suitable.

With multiplexers, we have also exemplified the potential of multi-round evaluation with Arithmetic Tuples. Adding only one round makes it possible to build a 16-MUX with an at most by 86% reduced tuple size and an at most by 85% reduced bandwidth. With a version of Beaver Triples, we need five rounds, and the tuple size is decreased by 90% compared to one round with Arithmetic Tuples. This value was almost reached by the Arithmetic Tuples based approach that needs three fewer rounds.

For other applications, we have seen that Binomial Tuples can perform better in tuple size than our Arithmetic Tuples approach. The polynomials for these applications comprise a large proportion of the possible multilinear monomials with these variables. For such “dense” polynomials, Binomial Tuples have the advantage of better reusability of tuple entries.

Finally, we have considered prefix products for which the combined approach based on Arithmetic and Binomial Tuples presented in [11] produced the best tuple size. In contrast, a more straightforward approach based on Binomial Tuples led to the smallest bandwidth but with exponentially growing tuple sizes.

In summary, with multiplexers and prefix products, we have seen two practical applications that benefit from the Arithmetic Tuples approach. Moreover, we have observed that slightly more sophisticated methods can significantly improve the metrics. For multiplexers, additionally publishing the negated selection bits and for prefix products combining Arithmetic and Binomial Tuples both lead to smaller tuple sizes and bandwidths.

Outlook

In this work, we only looked at a few example applications. Still, the here presented algorithms can be applied to any multilinear polynomials and could even be extended to general polynomials, as mentioned in Section 4.1.3. This invites one to test out more applications. As we have seen, defining a different polynomial for an application or combining different approaches, as with prefix product, can significantly impact the metrics. Therefore experimenting with other polynomial definitions and general approaches could also be a starting point for further research.

Bibliography

[1] D. Beaver. “Efficient Multiparty Protocols Using Circuit Randomization”. In: *Advances in Cryptology — CRYPTO ’91*. Ed. by J. Feigenbaum. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 420–432. ISBN: 978-3-540-46766-3. DOI: 10.1007/3-540-46766-1_34 (cit. on pp. 15, 20).

[2] I. Damgård, M. Fitzi, E. Kiltz, J. B. Nielsen, T. Toft. “Unconditionally Secure Constant-Rounds Multi-party Computation for Equality, Comparison, Bits and Exponentiation”. In: *Theory of Cryptography*. Ed. by S. Halevi, T. Rabin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 285–304. ISBN: 978-3-540-32732-5. DOI: 10.1007/11681878_15 (cit. on p. 17).

[3] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, N. P. Smart. “Practical Covertly Secure MPC for Dishonest Majority – Or: Breaking the SPDZ Limits”. In: *Computer Security – ESORICS 2013*. Ed. by J. Crampton, S. Jajodia, K. Mayes. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–18. ISBN: 978-3-642-40203-6. DOI: 10.1007/978-3-642-40203-6_1 (cit. on pp. 15, 17).

[4] D. Evans, V. Kolesnikov, M. Rosulek. “A Pragmatic Introduction to Secure Multi-Party Computation”. In: *Foundations and Trends® in Privacy and Security* 2.2-3 (2018), pp. 70–246. ISSN: 2474-1558. DOI: 10.1561/3300000019 (cit. on p. 17).

[5] C. Gentry, K. A. Goldman, S. Halevi, C. Julta, M. Raykova, D. Wichs. “Optimizing ORAM and Using It Efficiently for Secure Computation”. In: *Privacy Enhancing Technologies*. Ed. by E. De Cristofaro, M. Wright. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–18. ISBN: 978-3-642-39077-7. DOI: 10.1007/978-3-642-39077-7_1 (cit. on p. 17).

[6] A. Goel, M. Hall-Andersen, A. Hegde, A. Jain. “Secure Multiparty Computation with Free Branching”. In: *Advances in Cryptology – EUROCRYPT 2022*. Ed. by O. Dunkelman, S. Dziembowski. Cham: Springer International Publishing, 2022, pp. 397–426. ISBN: 978-3-031-06944-4. DOI: 10.1007/978-3-031-06944-4_14 (cit. on p. 17).

[7] B. R. Heap. “Permutations by Interchanges”. In: *The Computer Journal* 6.3 (Nov. 1963), pp. 293–298. ISSN: 0010-4620. DOI: 10.1093/comjnl/6.3.293 (cit. on p. 36).

[8] D. Lu, A. Yu, A. Kate, H. Maji. *Polymath: Low-Latency MPC via Secure Polynomial Evaluations and its Applications*. Cryptology ePrint Archive, Paper 2021/978. 2021. URL: https://eprint.iacr.org/2021/978 (cit. on p. 17).

[9] S. Ohata, K. Nuida. “Communication-Efficient (Client-Aided) Secure Two-Party Protocols and Its Application”. In: *Financial Cryptography and Data Security*. Ed. by J. Bonneau, N. Heninger. Cham: Springer International Publishing, 2020, pp. 369–385. ISBN: 978-3-030-51280-4. DOI: 10.1007/978-3-030-51280-4_20 (cit. on p. 15).

[10] R. Ostrovsky, V. Shoup. “Private information storage”. In: *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. 1997, pp. 294–303. DOI: 10.1145/258533.258606 (cit. on p. 17).

[11] P. Reisert, M. Rivinius, T. Krips, R. Kuesters. *Arithmetic Tuples for MPC*. Cryptology ePrint Archive, Paper 2022/667. 2022. URL: https://eprint.iacr.org/2022/667 (cit. on pp. 15, 17, 21, 23, 38, 39, 46, 47, 49).

[12] R. Sedgewick. “Permutation generation methods”. In: *ACM Computing Surveys (CSUR)* 9.2 (1977), pp. 137–164 (cit. on p. 36).

[13] A. Shamir. “How to Share a Secret”. In: *Commun. ACM* 22.11 (Nov. 1979), pp. 612–613. ISSN: 0001-0782. DOI: 10.1145/359168.359176 (cit. on p. 19).

All links were last followed on December 18, 2022.

Declaration