Rep3 Reloaded: On the Cost of Function-Dependent Preprocessing in Semi-Honest 3PC with Honest Majority

Marcel Keller

CSIRO's Data61
marcel.keller@data61.csiro.au
mks.keller@gmail.com

Abstract. Rep3 denotes the implementation of semi-honest three-party computation with an honest majority in MP-SPDZ (CCS'20). It uses replicated secret sharing with one message per multiplication and party as proposed by Araki et al. (CCS'16). This approach is rivaled by Astra (CCSW'19) and Trio (PETS'25), which use function-dependent preprocessing. The latter is more involved than, e.g., Beaver triples which can be used as a commodity.
In this work, we present a full implementation of Astra and Trio in MP-SPDZ, and we evaluate the costs of the different approaches. We show the equivalence of the schemes, which implies that a protocol in any of the schemes can be translated to one in another with the same overall communication cost. We also present an improvement to two important building blocks for privacy-preserving computation, namely secure comparison and probabilistic truncation used in fixed-point arithmetic. To evaluate our implementation, we have benchmarked machine learning training and inference in all three schemes, improving on Keller and Sun (ICML'22) by over 30%. Our implementation also highlights the large storage requirements of function-dependent preprocessing as it runs the two phases separately. To the best of our knowledge, this is the first implementation to do so.

Keywords: Secure multiparty computation, replicated secret sharing, privacy-preserving machine learning

1 Introduction

Multi-party computation (MPC) is one of only a few privacy-enhancing technologies, together with homomorphic encryption, differential privacy, and trusted execution environments. It allows a set of parties to compute on data such that only desired outputs are revealed to some parties while input and intermediate data remain secret as long as some defined assumptions on the parties' behavior remain true.

A core aspect of multi-party computation is secret sharing, which is an umbrella term for schemes that enable the parties to keep information in a distributed way such that individual parties or unqualified sets of parties cannot learn anything about the secret but qualified sets can. Early and commonly used examples of secret sharing schemes are additive secret sharing and Shamir's secret sharing scheme [Sha79]. The additive secret sharing scheme involves random values in a domain that sum up to the secret while Shamir's secret sharing scheme derives shares using a random polynomial in a finite field. The degree of the polynomial determines how many shares are needed for reconstruction. The distribution of fewer shares than the threshold can be proven to be uniform and independent of the secret, ensuring the scheme's security.

While Shamir's secret sharing scheme is very flexible as it allows any number of parties and thresholds as well as requiring minimal information per party (one field element), and the underlying mathematics allows building an MPC scheme from the properties of polynomial arithmetic [BGW88,CCD88]. However, it requires a field such as the integers modulo a prime and thus operations that are not readily available on common hardware. On the other hand, additive secret sharing is not known for directly enabling multi-party computation. Instead, it requires additional primitives such as oblivious transfer [GMW87] or homomorphic encryption [DJ01].

For a low number of parties, in particular three parties, replicated secret sharing [ISN89] provides a compromise. It is based on additive secret sharing (thus working in any domain) but has some parties holding more than a single additive share. As research into practical aspects increased, it was first used as a mere tool to Shamir-based schemes [CDI05,DGKN09] before becoming the main secret sharing scheme in Sharemind [BLW08], one of the earliest implementation of multi-party computation.

If the amount of replication fulfills some condition, it is possible to construct an MPC protocol without further primitives [CDM00]. In the case of three parties, the easiest such case is every party getting two different out of three additive shares. Multiplication can then be computed with every party only sending a single element [AFL$^+$16]. We refer to the protocol suite based on this multiplication as Rep3 as it is done in the documentation of MP-SPDZ [Kel20].

ABY3 [MR18] supplemented this approach with primitives that are indispensable in the context of applications such as privacy-preserving machine learning, most notably secure comparison and probabilistic truncation. The latter is a common ingredient in fractional computation going back to Catrina and Saxena [CS10] who observed that it is more efficient to compute an imprecise (possibly off-by-one) truncation instead of a precise one.

For secure comparison, ABY3 uses the mixed-circuit paradigm introduced by ABY [DSZ15]. It is based on the observation that the computation of binary circuits is more efficient in the matching domain ($\mathsf{GF}(2)$ mathematically speaking). Secure comparison is fundamentally linked to binary circuits. ABY3 converts locally share by share from the arithmetic domain (secret sharing modulo a power of two) to the binary domain. The addition required for reconstruction in the

arithmetic is then executed within the binary domain. Table 2 summarizes the cost of this approach.

Table 1. Cost of multiplication across protocols. l denotes the bit length of the arithmetic domain, that is, commutation modulo 2^l. Both communication and phase storage are understood to be the sum across all parties.

Protocol	Prep. comm.	Rounds	Online comm.	Rounds	Phase storage
Rep3 [AFL+16]	0	0	$3l$	1	0
Astra [CCPS19]	l	1	$2l$	1	$4l$
Trio [HKSW+25]	l	1	$2l$	1	$4l$

Table 2. Cost of secure comparison components across protocols. l denotes the bit length of the arithmetic domain (that is, computation modulo 2^l), and k denotes the bit length of the value (that is, the secret value being in $[-2^k, 2^k - 1]$). Furthermore, λ denotes the security parameter for garbled circuits (GC) used by Astra, commonly 128. *denotes malicious security.

Protocol	Component	Communication	Rounds
ABY3 [MR18]	MSB extraction (3-way)	$3 \cdot (4 \cdot (k-1) - \log_2 k)$	$\log_2 k + 1$
	MSB extraction (2-way)	$k + 3 \cdot (3k - 2 - \log_2 k)$	$\log_2 k + 2$
	Bit injection	$4l$	2
	Bit injection	$6l$	1
Astra [CCPS19]	MSB extraction (GC)	$8k\lambda$	1
	Bit injection	$\perp$	
Swift* [KPPS21]	MSB extraction	$18k - 12$	$\log_2 k + 1$
	Bit injection	$19l$	2
Trio [HKSC24]	MSB extraction	(same as ABY3)	
	Bit injection	$4l$	2
Ours	MSB extraction	(same as ABY3)	
	Bit injection	$3l$	2

For probabilistic truncation, ABY3 only considered an approach that requires the secret be in a negligibly small part of the secret sharing domain. To alleviate this, Dalskov et al. [DEK20] introduced a protocol that grows this range to half the secret sharing domain.

All protocols mentioned so far either do not require preprocessing or function-independent preprocessing comparable to Beaver multiplication triples [Bea92]. Preprocessing in this context refers to a part of the computation that can happen before the inputs are known. Astra [CCPS19] was the first work to feature

Table 3. Cost of probabilistic truncations with small gap across protocols. $l$denotes the bit length of the arithmetic domain (that is, computation modulo 2^l). $\perp$ denotes that no protocol has been proposed.

Protocol	Communication	Rounds
ABY3 [MR18]	$\perp$	
Dalskov et al. [DEK20]	$8l$	3
Astra [CCPS19]	$\perp$	
Trio [HKSW$^+$25]	$5l$	2
Ours	$4l$	2

function-dependent preprocessing in the context of secret sharing. The difference to the former is that the exact computation is required to be known for preprocessing whereas function-independent only requires the number of certain operations such as multiplications for execution. This difference might seem to be minor but we will see that it does make a practical difference because the computation as such will have to be run twice with all the overhead that this comes with.

More recently, Harth-Kitzerow [HKSW$^+$25] adapted Astra's approach to reduce the local computation, and a follow-up work [HKSC24] considers a probabilistic truncation based on top. Table 1 compares the cost of all multiplication protocols.

1.1 Our Contributions

We further the study of semi-honest three-party computation with honest majority as follows:

1. We establish the equivalence of three different secret sharing schemes introduced earlier: replicated secret sharing as used by Araki et al. [AFL$^+$16], Astra [CCPS19], and Harth-Kitzerow et al. [HKSW$^+$25] in Section 4. This implies that any protocol in any of the secret sharing schemes can be converted into a protocol in another one.
2. We generalize an idea of Attrapadung et al. [AHI$^+$22] on domain conversion to the protocols covered by this work in Section 3. This results in a primitive for the multiplication of a secret value and a value known by two of the three parties.
3. We use the above primitive to construct the most efficient protocols for probabilistic truncation (Section 5) and secure comparison (Section 6). As shown in Tables 3 and 2, this improves the communication cost by at least 25% and 20%, respectively, compared to prior work.
4. We have implemented all our protocols in MP-SPDZ [Kel20], and we intend to make the implementation available as open source.
5. To the best of our knowledge, our implementation is the first one that is complete and publicly available in this line of work of function-dependent

secret sharing. By complete, we mean that the preprocessing can be run independently while preserving all the preprocessed data for a separate online phase. In comparison, the only other implementation we are aware of is the one by Harth-Kitzerow et al. [HKSW+25],[1] which requires all preprocessing to be held in memory thus limiting execution to relatively small programs or interleaving the two phases. We argue that the latter voids the purpose of preprocessing as the overall complexity is the same as with Rep3 while making the protocols more complicated.

6. We benchmark all protocols with machine learning applications in Section 7. This further showcases the increased cost of function-dependent preprocessing because either phase alone turned out to be slower than Rep3, most likely due to the cost of storing or retrieving the preprocessing data on disk.

2 Preliminaries

Notation We use $[\cdot]$ for arithmetic secret sharing where the secret is computed from the shares in $\mathbb{Z}_{2^l} = \mathbb{Z}/(2^l\mathbb{Z})$, that is, with integers modulo 2^l. On the other hand, we use $\langle\cdot\rangle$ for binary secret sharing, that is, the computation takes place in $\mathbb{Z}_2 = \mathbb{Z}/(2\mathbb{Z}) = \mathsf{GF}(2)$. We do not use different notations for different secret sharing schemes as the higher-level protocols will be the same for all schemes.

We will use $\mathsf{PRNG}(k)$ for invoking a pseudo-random number generator with a specified key k known to several parties as specified.

Security Three-party computation with honest majority implies that there is only one corrupted party. Furthermore, we only consider semi-honest security. Together, this means that security requires the ability to simulate any party's view from the result of a computation assuming that the other parties behave as prescribed by the protocol.

2.1 Binary Adders

Binary adders are an essential component in CPUs but also appear in multi-party computation because computations such as a simple comparison are not provided by the core protocols. Therefore, dealing with the binary representation of numbers is often the only possible approach.

A core component in some binary adders is the so-called full adder, which adds up to three bits to a two-bit representation. Algorithm 1 presents the implementation that is optimal in the context of multi-party computation modulo 2, that is, using XOR/addition and AND/multiplication while minimizing the number of AND gates [SZ13].

The correctness s is trivial, and the correctness of c_{out} can be seen as follows:

$$a \oplus (t \wedge (c_{in} \oplus a)) = \begin{cases} a & t = 0 \\ c_{in} & t = 1. \end{cases}$$

[1] https://github.com/chart21/hpmpc

Algorithm 1: Full adder for MPC

Input Bits a, b, c_{in}
Output Bits s, c_{out} such that $s + 2 \cdot c_{out} = a + b + c_{in}$

1: $t \leftarrow a \oplus b$
2: $s \leftarrow s \oplus c$
3: $c_{out} \leftarrow a \oplus (t \wedge (c_{in} \oplus a))$
4: **return** s, c_{out}

In the first case, $a = 1 \Leftrightarrow a + b = 2$, and, in the second case, $a + b = 1$. The construction is also known as a MUX gate.

k-bit addition can be implemented by chaining k full adders after which the i-th bit of the output is computed as $a_i \oplus b_i \oplus c_{i+1}$ where c_{i+1} denotes the carry of the $(i-1)$-th bit. This is known as a textbook or ripple-carry adder. However, the sequential nature does not scale well in silicon or MPC. We will discuss two variants that decrease the number of rounds at the expense of more multiplications. Both of them are concerned with computed carry bits in a non-sequential fashion.

Parallel-prefix/Ladner-Fischer adder This adder uses a classical result from the literature on parallel computation [LF80]. At its core lies the question of how to compute the carry bit of a k-bit addition in a divide-and-conquer manner. In other words, if $k = 2l$ (for simplicity), what information can be computed on the l most and least significant bits independently such that the two parts can be combined to compute the respective information for all k bits? It turns out that this information consists of two bits called the generation bit g and the propagation bit p. The first captures whether a block of bits will always generate a carry bit, and the second captures whether a block of bits will generate a carry bit if a lower block feeds in a carry. The combination operation then looks as follows

$$p = p_L \wedge p_H$$
$$g = g_H \vee (g_L \wedge g_H)$$

where (p_L, g_L) and (p_H, g_H) denote the inputs relating to the lower and higher part respectively. It is straightforward to establish the correctness: Propagation overall is conditioned on both parts having propagation where generation is either given by generation in the upper part of propagation in the higher and generation in the lower part.

It is also easy to see that the operation is associative, that is, if we define the operator $\circ$ to achieve $(p, g) = (p_L, g_L) \circ (p_H, g_H)$ as above, it generally holds that

$$(x \circ y) \circ z = x \circ (y \circ z).$$

This is a condition for the parallel-prefix algorithm by Ladner and Fischer, which allows computing $x_1, x_1 \circ x_2, \ldots, x_1 \circ \cdots x_n$ for any $x_1, \ldots, x_n$ in $O(\log n)$ rounds of $O(n)$ invocations of $\circ$. As a result, we can compute all carries with a similar complexity in a binary addition of $x = \sum 2^i \cdot x_i$ and $y = \sum_i 2^i \cdot y_i$ because the base case of propagation and generation can be computed as $(p_i, g_i) = (x_i \oplus y_i, x_i \wedge y_i)$. Finally, if c_i denotes the carry resulting from the prefix operation at position, the result bit can be computed as stated earlier:

$$(x + y)_i = x_i \oplus y_i \oplus c_i.$$

Carry-select adder This adder explores computing carries using blocks of full adders [OB88]. The inputs are separated into blocks of variable length, and for each block, the output carries are computed for both possible input carries. This gives two options for each block output carry, which can then be sequentially computed using MUX gates. Once a block input carry is available, the actual carries inside the block can also be computed using MUX gates. In the context of multi-party computation, the cost of a MUX gate equals the cost of AND, which makes this adder a viable option [ABF+18]. As a result, MP-SPDZ uses it by default for shorter lengths.[2]

Extraction of most significant bit The secure comparison protocols considered in this work compute the result by extracting the most significant bit (which indicates the sign) from the difference of the two inputs. This extraction involves a binary adder without using any of the lower bits. This changes the evaluation of the different adders. For example, the ripple-carry adder clearly has to compute the complete addition in order to compute the most significant bit, resulting in k AND gates in k rounds for k-bit inputs. On the other hand, some invocations of the propagation-generation operator in the parallel-prefix adder become irrelevant as their output only feeds into the lower bits of the output. Table 4 shows how the different variants compare for full addition and MSB extraction across a range of bit lengths. If choosing round-optimal parallel-prefix addition, the cost of MSB extraction is roughly $3k - \log_2 k$ for k bits.

3 Protocol Building Blocks

In this section, we introduce the most basic aspects of the protocols considered in this work. This includes secret multiplication, secret inputs, and a special multiplication. The latter will prove to be the core component of our improvements for secure comparison and probabilistic truncation.

3.1 Rep3

Rep3 is the name used in MP-SPDZ for the protocol by Araki et al. [AFL+16]. It uses three-party replicated sharing, that is, the secret x is shared as a random

[2] `https://github.com/data61/MP-SPDZ/blob/698b015064b00c80629ff97a3e7c15e6b4ecf270/Compiler/types.py#L3445`

Table 4. Cost of full addition and MSB extraction across bit adders. RCA denotes ripple-carry/textbook addition, PPA denotes parallel-prefix addition (optimized for rounds or ANDs), and CSA denotes carry-select addition.

		#AND	#Rd	#AND	#Rd	#AND	#Rd	#AND	#Rd	#AND	#Rd
Full	RCA	16	16	32	32	64	64	128	128	256	256
	PPA (rds)	63	4	159	5	383	6	895	7	2047	8
	PPA (ANDs)	51	6	113	8	239	10	493	12	1003	14
	CSA	45	6	94	9	189	12	389	17	765	24
MSB	RCA	16	16	32	32	64	64	128	128	256	256
	PPA (rds)	40	4	87	5	182	6	373	7	756	8
	PPA (ANDs)	40	4	87	5	182	6	373	7	756	8
	CSA	32	6	65	9	130	12	264	17	518	24

sum $x = x_0 + x_1 + x_2$, and Party P_i holds (x_{i-1}, x_{i+1}) where the indices are understood modulo three. Clearly, the parties can add locally by adding their shares as $x + y = (x_0 + y_0) + (x_1 + y_1) + (x_2 + y_2)$ if $y = y_0 + y_1 + y_2$ in addition to the above equation. Protocol 1 shows the multiplication protocol. Note that it works for any ring.

Protocol 1: Rep3 multiplication

Setup PRNG keys k_i known to P_{i-1} and and P_{i+1} for all $i \in \{0, 1, 2\}$
Input Replicated secrets $[x]$ and $[y]$
Output Replicated secret $[x \cdot y]$

1: Every P_i computes $z_{i-1} = x_{i-1} \cdot (y_{i-1} + y_{i+1}) + x_{i+1} \cdot y_{i-1} + \mathsf{PRNG}(k_{i-1}) - \mathsf{PRNG}(k_{i+1})$ and sends it to P_{i+1}.
2: P_i receives z_{i+1} and defines $[z] = (z_{i-1}, z_{i+1})$.
3: **return** $[z]$

The correctness follows trivially, and the security follows from the fact that the received elements are obscured by a random number unknown to the receiver. The communication cost is one element per party, and every party has to generate two random elements.

The low cost of the protocol increases the impact of the concrete implementation. In particular, if the domain is the ring modulo 2^{64}, every addition and multiplication costs only one CPU clock cycle. Similarly, a vector of 64 shares can be operated on simultaneously for binary shares in $\mathbb{Z}_2$.

Furthermore, we found that the AES-based PRNG in MP-SPDZ can output 4 GB per second on a single core of a commodity CPU. This is similar to the throughput of a strong network link and thus unlikely to be a clear bottleneck.

Therefore, we propose the *transition complexity* as a measure for the cost of maintaining the state during a protocol. This matters because it is beneficial to buffer the communication of several elements due to the cost of network rounds. This buffering requires the maintenance of data structures like C++ vectors, which in turn involves memory management and pointer arithmetic. Concretely, Rep3 multiplication has a transition complexity of 3 because every party needs to store z_{i-1} during the communication. We will later see that the transition complexity corresponds to the cost of a concrete implementation.

Finally, we note that Araki et al. and later Harth-Kitzerow et al. [HKSW+25] postulate that this multiplication requires a division by three. However, the construction by Lindell and Nof [LN17, Section 6.2.2] already presents the protocol without division.

From now on, we will use $[x]_i$ to denote x_i as above, that is, the additive share known to parties P_{i+1} and P_{i-1}. This allows using x_i otherwise.

3.2 Constructing Rep3 Protocols from the Input Protocol

Protocol 2 shows the most communication-efficient input protocol for Rep3 as presented by Eerikson et al. [EKO+20]. It is easy to see that the result is a secret sharing of the input, and the security is straight-forward based on the security of the PRG. The cost is the communication of one element and two PRG invocations.

Protocol 2: Rep3 input

Input x known by Party P_i
Output $[x]$ shared between the parties
Setup PRG k_{ij} known to parties P_i and P_j where $i \neq j$

1: P_i and P_j generate a random value r using their shared PRG key
2: P_i sends $x - r$ to P_k for $k \notin \{i, j\}$
3: The parties define $[x]$ using $[x]_i = 0$, $[x]_j = x - r$, and $[x]_k = r$
4: **return** $[x]$

Correctness Trivially, $x = 0 + x - r + r$.

Security P_i does not learn any information, and P_j only learns a random value by definition. The same holds for P_k as $x - r$ is uniformly random if r is so, and r is not learnt by P_k. The following theorem states that even the combination of several inputs are simulatable for every individual party.

Theorem 1. *Let $[y] = \sum a_i[x_i]$ where x_i has been input by party P_i using the input protocol. Then, the shares of y are simulatable by every party for every choice of a_i.*

Proof. W.l.o.g., we assume that $j = i + 1$ when running the input protocol with P_i holding the input. Thus,

$$\begin{aligned}
[y]_0 &= a_0 \cdot 0 + a_1 \cdot r_1 + a_2 \cdot (x_2 - r_2) = a_1 \cdot r_1 + a_2 \cdot (x_2 - r_2) \\
[y]_1 &= a_0 \cdot (x_0 - r_0) + a_1 \cdot 0 + a_2 \cdot r_2 = a_0 \cdot (x_0 - r_0) + a_2 \cdot r_2 \\
[y]_2 &= a_0 \cdot r_0 + a_1 \cdot (x_1 - r_1) + a_2 \cdot 0 = a_0 \cdot r_0 + a_1 \cdot (x_1 - r_1)
\end{aligned}$$

if r_i is the random value generated for P_i's input. This is equivalent to the result of the following calculation

$$\begin{pmatrix} 0 & a_1 & -a_2 \\ -a_0 & 0 & a_2 \\ a_0 & -a_1 & 0 \end{pmatrix} \cdot \begin{pmatrix} r_0 \\ r_1 \\ r_2 \end{pmatrix} + \begin{pmatrix} a_2 \cdot x_2 \\ a_0 \cdot x_0 \\ a_1 \cdot x_1 \end{pmatrix}$$

Due to symmetry, we only consider P_0 being corrupted. If $a_1 = 0$ or $a_2 = 0$, knowledge of y of a_0 implies knowledge of x_2 or x_1. We therefore assume that $a_1 \neq 0$ and $a_2 \neq 0$. In this case, if a_1 is invertible (i.e., odd), the multiplication with r_1 unknown to P_0 perfectly randomizes the distribution of $[y]_0$ and $[y]_2$. If a_1 is a zero divisor, i.e., $a_1 = 2^k \cdot a^*$ for odd a^* and $k > 0$, consider that P_0 can derive the lowest k bits of x_2 from the knowing x_0:

$$2^k \cdot a^* \cdot x_1 + a_2 \cdot x_2 = y - a_0 \cdot x_0.$$

This proofs that, should P_0 learn the complete sharing, they will not learn anything about x_1 and x_2 that could be derived from y. □

Protocol 3 restates the multiplication protocol using the input protocol. While the result is the same as above, we consider the more modular presentation is crucial to an improved understanding.

Protocol 3: Rep3 multiplication

Input Sharings $[x]$ and $[y]$
Output Sharing $[x \cdot y]$

1: Every party P_i computes $z_i = [x]_{i-1} \cdot ([y]_{i-1} + [y]_{i+1}) + [x]_{i+1} \cdot [y]_{i-1}$ and shares it to $[z_i]$ using the input protocol.
2: The parties compute $[z] = \sum_i [z_i]$.
3: **return** $[z]$

The correctness can be seen as follows:

$$\begin{aligned} z &= \sum_i z_i \\ &= \sum_i [x]_{i-1} \cdot ([y]_{i-1} + [y]_{i+1}) + [x]_{i+1} \cdot [y]_{i-1} \\ &= \sum_i [x]_i \cdot ([y]_i + [y]_{i+2}) \sum_i [x]_i \cdot [y]_{i+1} \\ &= \sum_i [x]_i \cdot \sum_i [y]_i \\ &= x \cdot y. \end{aligned}$$

The security furthermore follows directly from the security of the input protocol.

The main cost comes from the three invocations of the input protocol, resulting in the communication of three elements and six invocations of the PRG evenly distributed across the parties. In terms of mathematical operations, there are two multiplications and four additions/subtractions per party taking into consideration that the input protocol produces some zero shares.

The formulation of the multiplication protocol as three calls to the input protocol directly leads to a more efficient protocol for multiplication between a secret value and one known to two of the three parties. While this might seem oddly specific, there is a useful application in bit injection as discovered by Attrapadung et al. [AHI+22]. Protocol 4 shows the details. The correctness follows from the definition as $x = \sum_i [x]_i$, and the security follows from the security of the input protocol.

Protocol 4: Rep3 special multiplication

Input Secret $[x]$, y known to P_1 and P_2
Output $[x \cdot y]$

1: P_1 computes $z_1 = y \cdot ([x]_2 + [x]_3)$, and P_2 computes $y \cdot a[x]_1$.
2: P_1 and P_2 share z_1 and z_2, respectively, using the input protocol.
3: The parties compute $[z] = [z_1] + [z_2]$.
4: **return** $[z]$

3.3 Astra

Astra [CCPS19] operates in the same security model as Rep3 but uses different semantics allowing the operation on partial shares before the inputs are available. More concretely, a secret x is shared as $(\mathsf{m}_x, \lambda_{x,1}, \lambda_{x,2})$ such that

$$x = \mathsf{m}_x - \lambda_{x,1} - \lambda_{x,2}.$$

As with Rep3, each party holds two of the shares but $(\lambda_{x,1}, \lambda_{x,2})$ are already generated during the offline phase. Thus, party P_0 holding it is not required to be present in the online phase. This is unlike parties P_1 and P_2 holding $(\mathsf{m}_x, \lambda_{x,1})$ and $(\mathsf{m}_x, \lambda_{x,2})$, respectively. Protocol 5 outlines the multiplication protocol.

Protocol 5: Astra multiplication

Setup PRNG keys k_{01} known to (P_0, P_1) and k_{02} known to (P_0, P_2)
Input Astra secrets $[x]$ and $[y]$
Output Astra secret $[x \cdot y]$
Preprocessing

1: P_0 and P_1 sample $\lambda_{z,1}$ and $\gamma_{xy,1}$ using $\mathsf{PRNG}(k_{01})$.
2: P_0 and P_2 sample $\lambda_{z,2}$ using $\mathsf{PRNG}(k_{02})$.
3: P_0 computes $\gamma_{xy} = (\lambda_{x,1} + \lambda_{x,2}) \cdot (\lambda_{y,1} + \lambda_{y,2})$ and $\gamma_{xy,2} = \gamma_{xy} - \gamma_{xy,1}$.
4: P_0 sends $\gamma_{xy,2}$ to P_2.

Online

1: For $i \in \{1, 2\}$, P_i computes $\mathsf{m}_{z,i} = (i-1)\mathsf{m}_x\mathsf{m}_y - \mathsf{m}_x\lambda_{y,i} - \mathsf{m}_y\lambda_{x,i} + \lambda_{y,i} + \gamma_{xy,i}$.
2: P_1 and P_2 exchange $\mathsf{m}_{z,1}$ and $\mathsf{m}_{z,2}$ and set $\mathsf{m}_z = \mathsf{m}_{z,1} + \mathsf{m}_{z,2}$.

While the overall communication and randomness generation is the same as in Rep3, it is unevenly distributed among the parties. The transition complexity is 6 due to the storage of $\lambda_{z,i}$, $\gamma_{xy,1}$, and $\mathsf{m}_{z,i}$ by P_i for $i \in \{1, 2\}$.

3.4 Trio

Trio [HKSW+25] is a variant of Astra by defining the secret sharing of x as

$$x = \mathsf{m}_{x,1} - \lambda_{x,1}$$
$$x = \mathsf{m}_{x,2} - \lambda_{x,2}$$

where P_0 again knows $(\lambda_{x,1}, \lambda_{x,2})$ but P_i knows $(\mathsf{m}_{x,3-i}, \lambda_{x,i})$ for $i \in \{1, 2\}$. It is easy to see that this is interchangeable with Astra's secret sharing using

$$\mathsf{m}_x = \mathsf{m}_{x,i} + \lambda_{x,3-i}$$

for $i \in \{1, 2\}$. Protocol 6 shows the adapted multiplication protocol. Notably, the online phase is not symmetric between the active parties.

The complexity of the domain computation is slightly less than Astra with 5 rather than 6 multiplications but the transition complexity is slightly higher at 7 due to the storage of $\lambda_{z,1}$, $\lambda_{z,2}$, $r_{0,1}$, V_1, V_2, M_1, and M_2 across state transitions.

Protocol 6: Trio multiplication

Setup PRNG keys k_{01} known to (P_0, P_1) and k_{02} known to (P_0, P_2)
Input Trio secrets $[x]$ and $[y]$
Output Trio secret $[x \cdot y]$
Preprocessing

1: P_0 and P_1 sample $\lambda_{z,1}$ and $r_{0,1}$ using $\mathsf{PRNG}(k_{02})$.
2: P_0 and P_2 sample $\lambda_{z,2}$ using $\mathsf{PRNG}(k_{01})$.
3: P_0 computes $M_0 = \lambda_{x,2} \cdot \lambda_{y,2} - (\lambda_{x,1} - \lambda_{y,2}) \cdot (\lambda_{x,2} - \lambda_{y,1}) + r_{0,1}$.
4: P_0 sends M_0 to P_2.

Online

1: P_1 computes $V_1 = \mathsf{m}_{x,2} \cdot \lambda_{y,1} + \mathsf{m}_{y,2} \cdot \lambda_{x,1} + r_{0,1}$ and $M_1 = V_1 - \lambda_{z,1}$
2: P_2 computes $V_2 = \mathsf{m}_{x,1} \cdot \mathsf{m}_{y,1} + M_0$ and $M_2 = V_2 + \lambda_{z,2}$.
3: P_1 and P_2 exchange M_1 and M_2.
4: P_1 computes $\mathsf{m}_{z,1} = M_2 - V_1$ and P_2 computes $\mathsf{m}_{z,2} = V_2 - M_1$.

3.5 Astra Input Protocols

The asymmetry of Astra requires two different input protocols depending on whether the inputting party is party 0 or not. Protocol 7 shows the adaption of the Rep3 input protocol for inputting party 0. It requires the input to be available during the offline phase as party 0 does not participate in the online phase. While this might seem contradictory, its use will be seen in the protocols that follow. Correctness is straightforward as

$$\mathsf{m}_x - \lambda_{x,1} - \lambda_{x,2} = 0 - \lambda_{x,1} - (\lambda_{x-1} - x) = x$$

by definition. It is also easy to see that the views of P_1 and P_2 alone are independent of the input x.

Unlike Protocol 7, the input protocol for parties 1 and 2 only requires the input to be available during the online phase. Correctness follows directly from the definition, and the security follows the fact that m_x perfectly masks x.

3.6 Trio Input Protocols

The respective protocols for Trio are very similar to Astra apart from the fact that Trio secret sharing does not feature m_x. See Protocols 9 and 10 for details. Correctness of Protocol 9 follows from

$$\begin{aligned}
\mathsf{m}_{x,i} - \lambda_{x,i} &= -\lambda_{x,3-i} - \lambda_{x,i} \\
&= -\lambda_{x,1} - \lambda_{x,2} \\
&= -(-\lambda_{x,2} - x) - \lambda_{x,2} \\
&= x
\end{aligned}$$

Protocol 7: Input from party 0 in Astra

Setup PRNG key k_{02} known to parties P_0 and P_2
Input P_0 holds x.
Output Parties hold $[x]$.
Preprocessing

1: P_0 samples $\lambda_{x,2}$ using $\mathsf{PRNG}(k_{02})$ and computes $\lambda_{x,1} = -\lambda_{x,2} - x$.
2: P_0 sends $\lambda_{x,1}$ to P_1.
3: P_2 samples $\lambda_{x,2}$ using $\mathsf{PRNG}(k_{02})$.

Online

1: P_1 and P_2 set $\mathsf{m}_x = 0$.

Protocol 8: Input from party 1 or 2 in Astra

Setup PRNG keys k_{jk} known to parties P_j and P_k for $k \neq j$
Input P_i holds x for $i =\in \{1, 2\}$
Output Parties hold $[x]$.
Preprocessing

1: P_0 and P_i sample $\lambda_{x,i}$ using $\mathsf{PRNG}(k_{0,i})$.
2: P_0 and P_{3-i} set $\lambda_{x,3-i} = 0$.

Online

1: P_i computes $\mathsf{m}_x = x + \lambda_{x,i}$ and sends it to P_{3-i}.

for $i \in \{1, 2\}$ while correctness of Protocol 10 follows from

$$\mathsf{m}_{x,i} - \lambda_{x,i} = x$$
$$\mathsf{m}_{x,3-i} - \lambda_{x,3-i} = x$$

by definition for inputting party i.

Protocol 9: Input from party 0 in Trio

Setup/Input/Output/Preprocessing See Protocol 7.
Online

1: P_i sets $\mathsf{m}_{x,3-i} = -\lambda_{x,i}$ for $i = 1, 2$.

The input protocols now allow us to replicate the special multiplication for the protocols with function-dependent preprocessing, see Protocols 11 and 12. Correctness follows from

$$z_1 + z_2 = (\mathsf{m}_x - \lambda_{x,1} - \lambda_{x,2}) \cdot y = x \cdot y$$

Protocol 10: Input from party 1 or 2 in Trio

Setup/Input/Output/Preprocessing See Protocol 8.
Online

1: P_i sends $\mathsf{m}_{x,i} = x - \lambda_{x,i}$ to P_{3-i} and sets $\mathsf{m}_{x,3-i} = x$.

for Astra and from

$$z_1 + z_2 = (\mathsf{m}_{x,2} - \lambda_{x,2}) \cdot y = x \cdot y$$

for Trio. Furthermore, security is implied by the security of the input protocols.

Protocol 11: Special multiplication in Astra

Setup Pre-shared keys as in input protocols
Input Secret $[x]$ and y known to P_1 and P_2 at compile time
Output $[x \cdot y]$
Preprocessing The parties run the preprocessing for one input from each of P_1 and P_2 resulting in $\lambda_{z_1,1}, \lambda_{z_1,2}, \lambda_{z_2,1}, \lambda_{z_2,2}$.
Online

1: The parties let P_1 input $z_1 = (\mathsf{m}_x - \lambda_{x,1}) \cdot y$ resulting in $[z_1]$.
2: The parties let P_2 input $z_2 = -\lambda_{x,2} \cdot y$ resulting in $[z_2]$.
3: **return** $[z_1] + [z_2]$

Protocol 12: Special multiplication in Trio

Setup/Input/Output/Preprocessing As in Protocol 12
Online

1: The parties let P_1 input $z_1 = \mathsf{m}_{x,2} \cdot y$ resulting in $[z_1]$.
2: The parties let P_2 input $z_2 = -\lambda_{x,2} \cdot y$ resulting in $[z_2]$.
3: **return** $[z_1] + [z_2]$

3.7 Two-Party Input Protocols

A minor building block we will need later is a two-party input protocol where two parties turn a value they both know into a valid secret sharing. The secret sharing schemes allow this without communication as demonstrated in Protocols

13 and 14. The latter is limited to the special case we need in this work, but it is easy to generalize it. The correctness is trivial.

Protocol 13: Rep3 two-party input

Input $x \in \mathbb{Z}_{2^l}$ known to P_i and P_j, $i \neq j$
Output $[x]$

The parties set $[x]_i = [x]_j = 0$ and $[x]_k = x$ for $k \notin \{i, j\}$.

Protocol 14: Astra and Trio two-party input

Input $x \in \mathbb{Z}_{2^l}$ known to P_i and P_j, $i \neq j$
Output $[x]$

The parties set $\lambda_{x,1} = \lambda_{x,2} = 0$ and $\mathsf{m}_x = x$.

3.8 Benchmarks

We have implemented and carefully optimized all protocols in MP-SPDZ and benchmarked the multiplication on a commodity laptop. Table 5 shows that neither Astra nor Trio improves on Rep3 even when subtracting the cost for phase storage, that is, the time it takes to read preprocessing material from disk.

Table 5. Time in seconds by party for 10 million multiplications modulo 2^{64} in 100 rounds in a single thread on a commodity laptop.

Protocol and phase	Total	Communication	Randomness	Phase storage
Rep3	0.36/0.36/0.35	0.06/0.06/0.06	0.05/0.05/0.05	0.00/0.00/0.00
Astra preprocessing	0.44/0.47/0.52	0.04/0.00/0.03	0.07/0.05/0.02	0.00/0.12/0.12
Astra online	0.43/0.43	0.05/0.05	0.00/0.00	0.04/0.04
Trio preprocessing	0.45/0.49/0.48	0.04/0.00/0.03	0.07/0.05/0.02	0.00/0.13/0.13
Trio online	0.45/0.45	0.05/0.05	0.00/0.00	0.04/0.04

4 Equivalence of the Secret Sharing Schemes

The Rep3 and Astra secret sharing schemes are equivalent. In both cases, the parties hold three values in a way that two parties can reconstruct the secret. Furthermore, the following definition allows a conversion from Rep3 to Astra:

$$\begin{aligned}\mathsf{m}_x &= x_0\\ \lambda_x^1 &= -x_2\\ \lambda_x^2 &= -x_1.\end{aligned}$$

Recall that, in Rep3, party P_i holds (x_{i-1}, x_{i+1}). Thus, it is easy to see that the conversion above can be done locally as P_0 holding (x_2, x_1) in Rep3 corresponds to P_0 holding $(\lambda_x^1, \lambda_x^2)$ in Astra, P_1 holding (x_0, x_1) in Rep3 corresponds to P_1 holding $(\mathsf{m}_x, \lambda_x^1)$ in Astra, and P_2 holding (x_1, x_0) Rep3 corresponds to P_2 holding $(\mathsf{m}_x, \lambda_x^2)$ in Astra. This implies that the conversion is secure because no information is exchanged. Furthermore, the correctness can be seen as follows:

$$x = x_0 + x_1 + x_2 = \mathsf{m}_x - \lambda_x^1 - \lambda_x^2 = \mathsf{m}_x - \lambda_x.$$

In Protocol 15, we demonstrate how this equivalence can be used to transform the Astra multiplication protocol. In Rep3, all shares become available at the same time. Thus, there is no distinction between the preprocessing and the online phase. The resulting protocol is more involved than the Rep3 multiplication. In particular, it takes two rounds instead of because the Astra online phase depends on the output from the preprocessing phase. The correctness of the result can be seen as follows:

$$\begin{aligned}z_0 + z_1 + z_2 &= \mathsf{m}_{z,1} + \mathsf{m}_{z,2} + z_1 + z_2\\ &= x_0y_2 + x_2y_0 - z_2 + \gamma_{xy,1} + x_0y_0 + x_0y_1 + x_1y_0 - z_1 + \gamma_{xy,1}\\ &= x_0y_2 + x_2y_0 - z_2 + x_0y_0 + x_0y_1 + x_1y_0 + (x_1 + x_2)\cdot(y_1 + y_2)\\ &= (x_0 + x_1 + x_2)\cdot(y_0 + y_1 + y_2) = xy.\end{aligned}$$

With some modification, Protocol 15 can be changed to a protocol that takes only one round but preserves all all properties, that is, the arithmetic, randomness generation, and communication amount as well as the direction thereof. Protocol 16 shows the result. It calls the PRNG twice on a key shared between P_0 and P_1 (steps 1 and 2) and once on a key shared between P_0 and P_2. It involves each party sending one value, from P_0 to P_2 (step 1) as well between P_1 and P_2 (steps 2 and 3). All this shows the relevant properties of Protocols 15 are retained.

Protocol 15: Astra multiplication converted to Rep3

Setup PRNG keys k_i known to P_{i-1} and P_{i+1} for all $i \in \{0,1,2\}$
Input Rep3 secrets $[x]$ and $[y]$
Output Rep3 secret $[xy]$

1: P_0 and P_1 sample z_2 and r using $\mathsf{PRNG}(k_2)$.
2: P_0 and P_2 sample z_1 using $\mathsf{PRNG}(k_1)$.
3: P_0 computes $\gamma_{xy} = (x_1 + x_2) \cdot (y_1 + y_2)$ and $\gamma_{xy,2} = \gamma_{xy} - \gamma_{xy,1}$
4: P_0 sends $\gamma_{xy,2}$ to P_2
5: For $i \in \{1,2\}$, P_i computes $\mathsf{m}_{z,i} = (i-1)x_0y_0 + x_0y_{3-i} + x_{3-i}y_0 - z_{3-i} + \gamma_{xy,i}$
6: P_1 and P_2 exchange $\mathsf{m}_{z,1}$ and $\mathsf{m}_{z,2}$ and set $z_0 = \mathsf{m}_{z,1} + \mathsf{m}_{z,2}$.
7: Set $[z]_i = z_i$ for $i \in \{0,1,2\}$.

Protocol 16: Astra multiplication converted to Rep3 (simplified)

Setup PRNG keys as required for Rep3 input (Protocol 2)
Input Rep3 secrets $[x]$ and $[y]$
Output Rep3 secret $[xy]$

The parties run the following in parallel:
1: P_0 uses the Rep3 input protocol with $j = 1$ to share $(x_1+x_2)\cdot(y_1+y_2)$ as $[a_0]$.
2: P_1 uses the input protocol with $j = 0$ to share $x_0y_2 + x_2y_0$ as $[a_1]$.
3: P_2 uses the Rep3 input protocol with $j = 1$ to share $x_0y_0+x_0y_2+x_2y_0$ as $[a_2]$.
4: **return** $\sum_i [a_i]$

4.1 Trio

Similarly to above, the following definition allows converting between Rep3 and Trio:

$$\begin{aligned} \mathsf{m}_{x,1} &= x_0 + x_1 \\ \mathsf{m}_{x,2} &= x_0 + x_2 \\ \lambda_x^1 &= -x_2 \\ \lambda_x^2 &= -x_1. \end{aligned} \tag{1}$$

The correctness can be seen as follows:

$$\begin{aligned} x = x_0 + x_1 + x_2 &= \mathsf{m}_{x,1} - \lambda_x^1 \\ &= \mathsf{m}_{x,2} - \lambda_x^2 \end{aligned}$$

5 Fixed-Point Computation using Probabilistic Truncation

In privacy-preserving machine learning, it is common to represent fractional numbers as integers using a fixed precision, that is, to represent $x \in \mathbb{R}$ as $\lfloor x \cdot 2^f \rceil$ for some $f \in \mathbb{N}$. See Ng and Chow [NC23] for an overview. This has the advantage that integer computation can be used instead of the more expensive floating-point computation. In particular,

$$\lfloor (x+y) \cdot 2^f \rceil \approx \lfloor x \cdot 2^f \rceil + \lfloor y \cdot 2^f \rceil \quad \forall x, y \in \mathbb{R},$$

and integer-like addition does not require communication in most arithmetic MPC schemes. For multiplication, consider that

$$\lfloor (x \cdot y) \cdot 2^f \rceil \approx \lfloor x \cdot 2^f \rceil \cdot \lfloor y \cdot 2^f \rceil \cdot 2^{-f} \quad \forall x, y \in \mathbb{R}.$$

This gives rise to the problem of approximately multiplying with 2^{-f} and doing so as cheaply as possible. Catrina and Saxena [CS10] were the first to propose to use of probabilistic truncation for this. This is a right shift with a random rounding error that depends on the part that is removed. More concretely, for an integer x, the f-bit probabilistic truncation is $\lfloor x/2^f \rfloor + e$ for $e \in \{0, 1\}$ such that $\Pr[e = 1] = (x \bmod 2^f)/2^f$. The usefulness of this follows naturally as values are more likely to be rounded to one end as closer they are. For example, if $x = 0 \mod 2^f$, $e = 0$ will always hold. In this section, we will present a truncation protocol with small gap that improves on any prior work. A small gap means the input value can span at least half of the secret sharing domain. This is different to the approach first proposed by Mohassel and Zhang [MZ17], which requires that the input value lies within a range that is 2^s smaller than the secret sharing domain for a security parameter s. Dalskov et al. [DEK21] show that accommodating this constraint can destroy any efficiency savings made by the protocol as a larger domain impacts all aspects of the computation and not just the truncation.

5.1 Probabilistic Truncation in Rep3

Protocol 17 presents the Rep3 truncation protocol that combines the improvements by Harth-Kitzerow et al. [HKSC24] to the protocol by Dalskov et al. [DEK20] with the observation by Attrapadung et al. [AHI+22]. It calls the input protocol twice and the special multiplication in Protocol 4 once, resulting in a communication of $4l$ bits in two rounds. This improves on Harth-Kitzerow et al. by l bits.

Correctness By definition,

$$x = x_0 + x_1 + x_2 = c + d + 1 \mod 2^l.$$

Protocol 17: Probabilistic truncation in Rep3

Input
- Arithmetic domain modulo 2^l
- Number of relevant bits $k < l$
- Number of bits to truncate $m < k$
- Arithmetic secret $[x]$ such that $0 < x < 2^k$

Output Arithmetic secret $[y]$ such that $y = \lfloor x/2^m \rfloor + e$ where $e \in \{0, 1\}$ and $\Pr[e = 1] = (x \bmod 2^m)/2^m$

1: Party P_0 uses the input protocol to share c_{l-1} and $c' = \sum_{i=m}^{l-1} c_i \cdot 2^{i-m}$ where $\{c_i\}_{i=0}^{l}$ is the bit decomposition of $c = [x]_1 + [x]_2 - 1 \bmod 2^l$, that is, $c = \sum_i c_i \cdot 2^i$. This results in $[c_{l-1}]$ and $[c']$.
2: Parties P_1 and P_2 compute d_{l-1} and $d' = \sum_{i=m}^{l-1} d_i \cdot 2^{i-m}$ where $\{d_i\}_{i=0}^{l}$ is the bit decomposition of $d = [x]_0$, that is, $d = \sum_i d_i \cdot 2^i$. The parties then use the two-party input procedure to get $[d']$ and $[d_{l-1}]$.
3: The parties use the special multiplication procedure to multiply $[c_{l-1}]$ and d_{l-1}.
4: The parties compute $[y] = ([c_{l-1}] + [d_{l-1}] - 2 \cdot [c_{l-1} \cdot d_{l-1}]) \cdot 2^{k-m} + c' + d' - 1$.

Thus,

$$\begin{aligned} c' + d' &= \lfloor c/2^m \rfloor + \lfloor d/2^m \rfloor \\ &= \lfloor (x-1)/2^m \rfloor + E \cdot 2^{l-m} + F \end{aligned}$$

for $E, F \in \{0, 1\}$ indicating whether the addition of c and d modulo 2^l and 2^m overflows, respectively. We now argue that $E = c_{l-1} \vee d_{l-1}$. Consider that $c + d = x - 1$ and that $0 < x < 2^k$. It follows that the most significant bit of $c + d \bmod 2^l$ is 0. The overflow bit is equivalent to the carry output of a full adder gate on (c_{l-1}, d_{l-1}, C) where C denotes the carry bit from the addition of the lower $l - 1$ bits. The carry output is defined as the majority of the inputs. Furthermore, the most significant bit of $c+d \bmod 2^l$ is defined as $c_{l-1} \oplus d_{l-1} \oplus C$, which we know to be 0. Thus, $c_{l-1} \oplus d_{l-1} = C$, which implies that the majority of (c_{l-1}, d_{l-1}, C) is $c_{l-1} \vee c_{l-1}$. We derive that

$$\begin{aligned} y &= (c_{l-1} + d_{l-1} - c_{l-1} \cdot d_{l-1}) \cdot 2^{l-m} + c' + d' + 1 \\ &= (c_{l-1} \vee d_{l-1}) \cdot 2^{l-m} + c' + d' + 1 \\ &= E \cdot 2^{l-m} + c' + d' + 1 \\ &= \lfloor (c+d)/2^m \rfloor + 1 + F \\ &= \lfloor (x-1)/2^m \rfloor + 1 + F. \end{aligned}$$

It remains to show that the subtraction and addition of one serve to achieve the desired error distribution in the output. This is related to the error caused by the carry bit in the addition of c and d:

$$\begin{aligned} F &= (c \bmod 2^m) + (d \bmod 2^m) - ((c+d) \bmod 2^m) \\ &= (c \bmod 2^m) + ((x-1-c) \bmod 2^m) - ((x-1) \bmod 2^m) \\ &= \begin{cases} 0 & c \bmod 2^m \leq (x-1) \bmod 2^m \\ 1 & \text{otherwise.} \end{cases} \end{aligned}$$

By the definition of the secret sharing, $c = (x_1 + x_2 - 1) \bmod 2^l$ is uniformly random in $[0, 2^l)$. It follows that

$$\begin{aligned} \Pr[F = 1] &= 1 - \Pr[F = 0] \\ &= 1 - \Pr[c \bmod 2^m \leq (x-1) \bmod 2^m] \\ &= (x \bmod 2^m)/2^m. \end{aligned}$$

To understand the last equality, consider that, for $x = 0 \mod 2^m$, $c \bmod 2^m \leq -1 \bmod 2^m = 2^m - 1$ always happens. We can now conclude

$$\begin{aligned} \Pr[e = 1] &= \Pr[y = \lfloor x/2^m \rfloor + 1] \\ &= \Pr[\lfloor (x-1)/2^m \rfloor + 1 + F = \lfloor x/2^m \rfloor + 1] \\ &= \Pr[\lfloor (x-1)/2^m \rfloor + F = \lfloor x/2^m \rfloor] \\ &= \begin{cases} \Pr[F = 1] = 0 & x = 0 \mod 2^m \\ \Pr[F = 0] = (x \bmod 2^m)/2^m & \text{otherwise.} \end{cases} \end{aligned}$$

This is the desired distribution for probabilistic truncation.

Security The security properties are inherited from the underlying building blocks because there is information revealed outside them.

5.2 Truncation with Astra and Trio

The equivalence spelled out in Section 4 allows translating the Rep3 truncation protocol to Astra and Trio. More concretely, P_0 can compute $[x]_1 = -\lambda_{x,1}$ and $[x]_2 = -\lambda_{x,2}$, and P_i for $i = 1, 2$ can either compute $[x]_0 = \mathsf{m}_x$ or $[x]_0 = \mathsf{m}_{x,i} + \lambda_{x,3-i}$.

5.3 Cost and Benchmarks

We present the communication cost by phase for all protocols in Table 6. Furthermore, Table 7 shows our results for fixed-point multiplications in the different protocols considered. The online phases of the function-dependent protocol slightly outperform Rep3, but the total cost is considerably higher.

Table 6. Cost of our probabilistic truncation across secret sharing schemes. l denotes the bit length of the arithmetic domain, that is, commutation modulo 2^l. Both communication and phase storage are understood to be the sum across all parties.

Protocol	Prep. comm.	Rounds	Online comm.	Rounds	Phase storage
Rep3	0	0	$4l$	2	0
Astra/Trio	$2l$	1	$2l$	1	$6l$

Table 7. Time in seconds by party for 10 million fixed-point multiplications modulo 2^{64} in 100 rounds in a single thread on a commodity laptop.

Protocol and phase	Total	Communication	Randomness	Phase storage
Rep3	1.07/1.07/1.07	0.37/0.26/0.27	0.12/0.12/0.16	0.00/0.00/0.00
Astra preprocessing	1.39/1.45/1.44	0.26/0.06/0.08	0.20/0.12/0.08	0.00/0.53/0.49
Astra online	0.96/0.96	0.10/0.10	0.00/0.00	0.13/0.12
Trio preprocessing	1.40/1.50/1.46	0.22/0.08/0.14	0.21/0.13/0.09	0.00/0.52/0.48
Trio online	1.01/1.01	0.10/0.13	0.00/0.00	0.13/0.12

6 Secure Comparison

Secure comparison, that is, computing the bit $(a \overset{?}{<} b)$ in secret for $[a]$ and $[b]$ without revealing either is an important primitive as it is a building block for many computations in general and deep learning components like ReLU and max-pooling in particular. Dalskov et al. [EGK+20] have proposed a generic approach using mixed-circuit computing using so-called edaBits, a generalized version of daBits [RW19]. However, additive and replicated secret sharing allow a more efficient approach going back to Demmler et al. [DSZ15] for the two-party setting, which was later adapted by Mohassel and Rindal [MR18] as well as Araki et al. [ABF+18].

This approach computes the relevant bit in the binary domain (secret sharing modulo 2) before potentially converting it back to the arithmetic domain (secret sharing modulo 2^l). The first involves extracting the most significant bit of $a-b$, which indicates the sign of difference. If $a < b$, then $a - b$ will be negative and vice versa. Converting it back allows for further computation in the arithmetic domain where linear operations are cheaper. If further operations are computed in the binary domain, for example, an oblivious selection, it is necessary to convert longer values, which is more expensive than converting a single bit.

6.1 Splitting

A crucial component in the above works is the observation that the shares $[x]_1, \ldots, [x]_n$ of $[x]$, that is, $x_1, \ldots, x_n$ such that $x = \sum_i x_i$ can be converted

into binary secret sharings of $x_1, \ldots, x_n$ by defining

$$\langle x_i \rangle_j := \begin{cases} x_i & i = j \\ 0 & i \neq i, \end{cases} \tag{2}$$

the correctness of which is trivial. Once $\langle x_1 \rangle, \ldots, \langle x_n \rangle$ have been obtained, $\langle x \rangle$ can be computed using binary addition. As noted by Demmler et al., there are several options for this, and the optimal choice depends on the relative cost of rounds and operations.

In the context of three parties, Mohassel and Rindal showed that one can reduce from three-summand to two-summand addition using k full adders where k is the number of bits, similarly to Wallace trees used for binary multiplication [Wal64]. However, they also propose a splitting protocol that makes use of the fact that every party knows two out of three summands. Protocol 18 presents a detailed description. The cost is one invocation of the input protocol amounting to l bits in one round if the arithmetic domain is modulo 2^l. Given that a full adder costs one AND (three bits), two-way splitting is more efficient if $k > l/3$.

Protocol 18: Two-way splitting (A2B) in Rep3

Input Arithmetic secret $[x]$
Output Binary secrets $\langle x_0 \rangle$ and $\langle x_1 \rangle$ such that $x = x_0 + x_1$

1: The parties compute $\langle x_0 \rangle$ from $[x]_0$ as in Equation 2.
2: Party P_0 computes $y = [x]_1 + [x]_2$ and uses the input protocol to share it, resulting in $\langle x_1 \rangle$.
3: **return** $\langle x_0 \rangle, \langle x_1 \rangle$

As earlier, the equivalence in Section 4 can be used to construct a protocol for Astra and Trio, see Protocol 19. It demonstrates that the communication can entirely be done during the preprocessing phase.

The above considerations only cover general conversion from arithmetic to binary secret sharing. However, a comparison can be computed by extracting only one bit, namely the most significant bit of $[a - b]$, if a and b are guaranteed to be in a range covering only half of the domain.[3]

Protocol 20 details our secure less-than-zero or MSB extraction protocol. Note that it outputs a binary secret, which needs to be converted using bit injection for operations in the arithmetic domain. For completeness, we show the protocol using a three-way split in Protocol 21. Both are independent of the underlying secret sharing. Table 4 shows the cost of MSB extraction across a range of bit lengths.

[3] For example, they could be in $[-2^{k-2}, 2^{k-2})$ for computation modulo 2^k. This is a common assumption in practical multi-party computation [DEK20].

Protocol 19: Two-way splitting (A2B) in Astra and Trio

Input Arithmetic secret $[x]$
Output Binary secrets $\langle x_0 \rangle$ and $\langle x_1 \rangle$ such that $x = x_0 + x_1$
Preprocessing

1: P_0 computes $y = -(\lambda_{x,1} + \lambda_{x,2})$ and uses the input protocol to share it, resulting in $\langle x_1 \rangle$.
2: The parties set $\lambda_{x_0,0} = \lambda_{x_0,1} = 0$.

Online phase The parties set $\mathsf{m}_{x_0} = \mathsf{m}_x$.

Protocol 20: Less-than-zero/MSB extraction using 2-way splitting

Input Arithmetic secret $[x]$ for $x \in [-2^k, 2^k - 1]$ and $x^{(j)} \in \{0, 1\}$ such that $x = \sum_j 2^j \cdot x^{(j)} \bmod 2^k$
Output Binary secret bit $\langle x \overset{?}{<} 0 \rangle = \langle x^{k-1} \rangle$

1: Use two-way splitting to obtain $\langle x_0 \rangle, \langle x_1 \rangle$ such that $x = \sum_i x_i$
2: Compute the carry-out $\langle c_o \rangle$ of the addition of $\langle x_0 \rangle$ and $\langle x_1 \rangle$
3: **return** $\langle c_o \rangle \oplus \langle x_0^{(k-1)} \rangle \oplus \langle x_1^{(k-1)} \rangle$

6.2 Bit Injection

It is possible to execute the splitting in Protocol 18 in the other direction. However, since computing XOR in the arithmetic costs one multiplication instead of being a local operation, this has an overall higher cost. We nevertheless present the details in Protocol 22 for completeness and because it is equivalent to the conversion proposed by Harth-Kitzerow et al. [HKSW+25, Figure 19].

The total communication is $3l$ bits in two rounds, l bits for the input protocol and $2l$ in the special multiplication. This beats both Harth-Kitzerow et al. [HKSW+25] and ABY3 [MR18], both of which take $4l$.

6.3 Cost and Benchmarks

We present the communication cost by phase for all protocols in Table 8. We use absolute costs for easier readability. Refer to Table 2 for a parametrized version of the total cost.

Table 9 shows our benchmarks for the different protocols and phases. The two phases of the function-dependent protocols take a similar amount to the whole of Rep3, highlighting the additional cost of phase separation.

Table 10 shows our benchmark for the complete comparison. The cost is again similar between the separate phases and the complete Rep3 execution, with the preprocessing alone coming in higher as the complete Rep3 protocol.

Table 8. Cost of our 32-bit comparison across secret sharing schemes when using a 64-bit domain. Both communication and phase storage are understood to be the sum across all parties.

Protocol	Prep. comm.	Rounds	Online comm.	Rounds	Phase storage
Rep3	0	0	537	8	0
Astra/Trio	179	7	358	7	716

Table 9. Time in seconds by party for 10 million bit injections modulo 2^{64} in 100 rounds in a single thread on a commodity laptop

Protocol and phase	Total	Communication	Randomness	Phase storage
Rep3	0.21/0.21/0.21	0.09/0.08/0.07	0.04/0.04/0.04	0.00/0.00/0.00
Astra preprocessing	0.18/0.23/0.22	0.03/0.00/0.02	0.06/0.04/0.02	0.00/0.10/0.10
Astra online	0.18/0.18	0.04/0.04	0.00/0.00	0.04/0.04
Trio preprocessing	0.18/0.23/0.22	0.03/0.00/0.02	0.06/0.04/0.02	0.00/0.10/0.10
Trio online	0.20/0.20	0.04/0.04	0.00/0.00	0.04/0.04

Table 10. Time in seconds by party for 10 million 32-bit comparisons modulo 2^{64} in 100 rounds in a single thread on a commodity laptop

Protocol and phase	Total	Communication	Randomness	Phase storage
Rep3	1.41/1.41/1.41	0.28/0.25/0.26	0.00/0.00/0.00	0.00/0.00/0.00
Astra preprocessing	1.57/1.62/1.67	0.20/0.00/0.09	0.00/0.00/0.00	0.00/0.41/0.41
Astra online	1.23/1.23	0.13/0.13	0.00/0.00	0.12/0.12
Trio preprocessing	1.77/1.94/1.90	0.33/0.00/0.09	0.00/0.00/0.00	0.00/0.65/0.63
Trio online	1.34/1.34	0.14/0.14	0.00/0.00	0.12/0.12

Protocol 21: Less-than-zero/MSB extraction using 3-way splitting

Input Arithmetic secret $[x]$ for $x \in [-2^k, 2^k - 1]$ and $x^{(j)} \in \{0, 1\}$ such that $x = \sum_j 2^j \cdot x^{(j)} \bmod 2^k$

Output Binary secret bit $\langle x \overset{?}{<} 0 \rangle = \langle x^{k-1} \rangle$

1: Use three-way splitting to obtain $\langle x_0 \rangle, \langle x_1 \rangle, \langle x_2 \rangle$ such that $x = \sum_i x_i$
2: Use k half adders to obtain $\langle s \rangle, \langle c \rangle$ such that $x = s + c$
▷ k ANDs in one round
3: Compute the carry-out $\langle c_o \rangle$ of the addition of $\langle s \rangle$ and $\langle c \rangle$
4: **return** $\langle c_o \rangle \oplus \langle c^{(k-1)} \rangle \oplus \langle s^{(k-1)} \rangle$

Protocol 22: Bit injection using two-way split

Input Binary secret $\langle x \rangle$ where $x \in \{0, 1\}$
Output Arithmetic secret $[x]$

1: The parties compute $[x_0]$ from $\langle x \rangle_0$ as in Equation 2.
2: Party P_0 computes $y = \langle x \rangle_1 \oplus \langle x \rangle_2$ and uses the input protocol to share it, resulting in $[y]$.
3: Parties P_1 and P_2 use the two-party input protocol to create $[x_0]$ from x_0.
4: The parties use the special multiplication protocol to compute $[x] = [x_0 \oplus y] = [x_0] + [y] - 2 \cdot [x_0] \cdot [y]$. This is facilitated by P_1 and P_2 knowing x_0.
5: **return** $[x]$

7 Deep Learning Benchmarks

We have implemented deep learning applications based on the code by Keller and Sun [KS22] available in MP-SPDZ, and we have used three colocated AWS `c5.9xlarge` instances for benchmarks. Table 11 shows our results for LeNet training. LeNet [LBBH98] is a well-known network created for recognition of hand-written characters in the MNIST dataset [LCB10]. Our benchmarks demonstrates that the amount of data transmitted between the phases is of the same order of magnitude as the communication data. This matches the analysis of the building blocks. Sufficient training of MNIST would require about ten epochs, amounting to over three terabytes of data to be stored between the phases with function-independent preprocessing. The table also shows that we improve on Keller and Sun [KS22] as they report 343 seconds and 352 GB per epoch whereas our result is 256 seconds and 233 GB.

Table 11. One epoch of training LeNet with batch size 128, computed from benchmarking ten rounds.

Protocol	Phase	Communication	Time	Phase transmission
Rep3		222.80 GB	255.38 seconds	
Astra	Preprocessing	87.27 GB	290.19 seconds	320.37 GB
	Online	145.86 GB	320.61 seconds	
	Total	233.13 GB	610.80 seconds	
Trio	Preprocessing	87.27 GB	287.95 seconds	320.37 GB
	Online	145.86 GB	310.81 seconds	
	Total	233.13 GB	598.77 seconds	

Similarly, Table 12 shows our results for the AlexNet-like network in Falcon [WTB+21]. We improve on the result by Keller and Sun [KS22] from 771 to 513 GB and from 1603 to 805 seconds.

Table 12. One epoch of training Falcon's AlexNet-like network with batch size 128, computed from benchmarking ten rounds.

Protocol	Phase	Communication	Time	Phase transmission
Rep3		513.11 GB	804.87 seconds	
Astra	Preprocessing	233.22 GB	4254.35 seconds	820.69 GB
	Online	354.28 GB	868.01 seconds	
	Total	587.50 GB	5122.36 seconds	
Trio	Preprocessing	233.22 GB	679.95 seconds	820.69 GB
	Online	354.28 GB	866.71 seconds	
	Total	587.50 GB	1546.66 seconds	

We have also run VGG-16 inference for all the protocols on the same AWS instances, see Table 13.

8 Optimizations

In this section, we will highlight the low-level optimizations that we found relevant to the implementation of the protocols in this work.

Buffering System calls are relatively expensive, which makes it more efficient to buffer elements from multiple operations as they are relatively small (for example, 8 bytes for a 64-bit number). We have made sure to make the buffering as efficient as possible, which means allocating enough memory in advance and making sure to use fixed lengths in copying (allowing using registers instead of calls to `memcpy`).

Table 13. VGG-16 inference.

Protocol	Phase	Communication	Time	Phase transmission
Rep3		2.43 GB	11.80 seconds	
Astra	Preprocessing	0.89 GB	14.74 seconds	
	Online	1.56 GB	15.46 seconds	3.33 GB
	Total	2.44 GB	30.21 seconds	
Trio	Preprocessing	0.89 GB	14.52 seconds	
	Online	1.56 GB	15.18 seconds	3.33 GB
	Total	2.44 GB	29.70 seconds	

Keeping state As mentioned in Section 3, keeping state across communication rounds and phases is a crucial cost factor. We keep fixed-length elements in C++ vectors to avoid using variable-length copying. We also make sure to allocate enough memory before a loop to avoid reallocation, and we use iterators to minimize pointer arithmetic.

Acknowledgments

We acknowledge the use of an AI-enabled tool to correct spelling and grammar.

References

ABF+18. Toshinori Araki, Assi Barak, Jun Furukawa, Marcel Keller, Yehuda Lindell, Kazuma Ohara, and Hikaru Tsuchida. Generalizing the SPDZ compiler for other protocols. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 880–895. ACM Press, October 2018.

AFL+16. Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 805–817. ACM Press, October 2016.

AHI+22. Nuttapong Attrapadung, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Takahiro Matsuda, Ibuki Mishina, Hiraku Morita, and Jacob C. N. Schuldt. Adam in private: Secure and fast training of deep neural networks with adaptive moment estimation. *PoPETs*, 2022(4):746–767, October 2022.

Bea92. Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 420–432. Springer, Berlin, Heidelberg, August 1992.

BGW88. Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th ACM STOC*, pages 1–10. ACM Press, May 1988.

BLW08. Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In Sushil Jajodia and Javier López, editors, *ESORICS 2008*, volume 5283 of *LNCS*, pages 192–206. Springer, Berlin, Heidelberg, October 2008.

CCD88. David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (abstract) (informal contribution). In Carl Pomerance, editor, *CRYPTO'87*, volume 293 of *LNCS*, page 462. Springer, Berlin, Heidelberg, August 1988.

CCPS19. Harsh Chaudhari, Ashish Choudhury, Arpita Patra, and Ajith Suresh. Astra: High throughput 3pc over rings with application to secure prediction. In *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*, CCSW'19, page 81–92, New York, NY, USA, 2019. Association for Computing Machinery.

CDI05. Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In Joe Kilian, editor, *TCC 2005*, volume 3378 of *LNCS*, pages 342–362. Springer, Berlin, Heidelberg, February 2005.

CDM00. Ronald Cramer, Ivan Damgård, and Ueli M. Maurer. General secure multi-party computation from any linear secret-sharing scheme. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 316–334. Springer, Berlin, Heidelberg, May 2000.

CS10. Octavian Catrina and Amitabh Saxena. Secure computation with fixed-point numbers. In Radu Sion, editor, *FC 2010*, volume 6052 of *LNCS*, pages 35–50. Springer, Berlin, Heidelberg, January 2010.

DEK20. Anders P. K. Dalskov, Daniel Escudero, and Marcel Keller. Secure evaluation of quantized neural networks. *PoPETs*, 2020(4):355–375, October 2020.

DEK21. Anders P. K. Dalskov, Daniel Escudero, and Marcel Keller. Fantastic four: Honest-majority four-party secure computation with malicious security. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 2183–2200. USENIX Association, August 2021.

DGKN09. Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous multiparty computation: Theory and implementation. In Stanislaw Jarecki and Gene Tsudik, editors, *PKC 2009*, volume 5443 of *LNCS*, pages 160–179. Springer, Berlin, Heidelberg, March 2009.

DJ01. Ivan Damgård and Mats Jurik. A generalisation, a simplification and some applications of Paillier's probabilistic public-key system. In Kwangjo Kim, editor, *PKC 2001*, volume 1992 of *LNCS*, pages 119–136. Springer, Berlin, Heidelberg, February 2001.

DSZ15. Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *NDSS 2015*. The Internet Society, February 2015.

EGK+20. Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 823–852. Springer, Cham, August 2020.

EKO+20. Hendrik Eerikson, Marcel Keller, Claudio Orlandi, Pille Pullonen, Joonas Puura, and Mark Simkin. Use your brain! Arithmetic 3PC for any modulus with active security. In Yael Tauman Kalai, Adam D. Smith, and Daniel

Wichs, editors, *ITC 2020*, volume 163 of *LIPIcs*, pages 5:1–5:24. Schloss Dagstuhl, June 2020.

GMW87. Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987.

HKSC24. Christopher Harth-Kitzerow, Ajith Suresh, and Georg Carle. Truncation untangled: Scaling fixed-point arithmetic for privacy-preserving machine learning to large models and datasets. Cryptology ePrint Archive, Report 2024/1953, 2024.

HKSW+25. Christopher Harth-Kitzerow, Ajith Suresh, Yongqin Wang, Hossein Yalame, Georg Carle, and Murali Annavaram. High-throughput secure multiparty computation with an honest majority in various network settings. *Proc. Priv. Enhancing Technol.*, 2025:250–272, 2025.

ISN89. Mitsuru Ito, Akira Saito, and Takao Nishizeki. Secret sharing scheme realizing general access structure. *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)*, 72(9):56–64, 1989.

Kel20. Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1575–1590. ACM Press, November 2020.

KPPS21. Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. SWIFT: Super-fast and robust privacy-preserving machine learning. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 2651–2668. USENIX Association, August 2021.

KS22. Marcel Keller and Ke Sun. Secure quantized training for deep learning. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 10912–10938. PMLR, 17–23 Jul 2022.

LBBH98. Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

LCB10. Yann LeCun, Corinna Cortes, and CJ Burges. MNIST handwritten digit database. *ATT Labs [Online]. Available: `http://yann.lecun.com/exdb/mnist`*, 2, 2010. Creative Commons Attribution-Share Alike 3.0 license, `https://creativecommons.org/licenses/by-sa/3.0/`.

LF80. Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *J. ACM*, 27(4):831–838, October 1980.

LN17. Yehuda Lindell and Ariel Nof. A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 259–276. ACM Press, October / November 2017.

MR18. Payman Mohassel and Peter Rindal. ABY^3: A mixed protocol framework for machine learning. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 35–52. ACM Press, October 2018.

MZ17. Payman Mohassel and Yupeng Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy*, pages 19–38. IEEE Computer Society Press, May 2017.

NC23. Lucien K. L. Ng and Sherman S. M. Chow. SoK: Cryptographic neural-network computation. In *2023 IEEE Symposium on Security and Privacy*, pages 497–514. IEEE Computer Society Press, May 2023.

OB88. Vojin G. Oklobdzija and Earl R. Barnes. On implementing addition in vlsi technology. *Journal of Parallel and Distributed Computing*, 5(6):716–728, 1988.

RW19. Dragos Rotaru and Tim Wood. MArBled circuits: Mixing arithmetic and Boolean circuits with active security. In Feng Hao, Sushmita Ruj, and Sourav Sen Gupta, editors, *INDOCRYPT 2019*, volume 11898 of *LNCS*, pages 227–249. Springer, Cham, December 2019.

Sha79. Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979.

SZ13. Thomas Schneider and Michael Zohner. GMW vs. Yao? Efficient secure two-party computation with low depth circuits. In Ahmad-Reza Sadeghi, editor, *FC 2013*, volume 7859 of *LNCS*, pages 275–292. Springer, Berlin, Heidelberg, April 2013.

Wal64. C. S. Wallace. A suggestion for a fast multiplier. *IEEE Transactions on Electronic Computers*, EC-13(1):14–17, 1964.

WTB+21. Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. Falcon: Honest-majority maliciously secure framework for private deep learning. *PoPETs*, 2021(1):188–208, January 2021.

A B2A Conversion with daBits

DaBits [RW19] are a general tool for converting between an arithmetic and the binary domain as shown in Protocol 23. This is not necessarily the most optimal protocol for converting several secret bits to one arithmetic secret. However, it suffices for the single-bit base, which we mostly use in this work.

Protocol 23: Generic conversion from binary to arithmetic (B2A)

Input Binary secret $\langle x \rangle$
Output Arithmetic secret $[x]$

1: Generate daBit $([b], \langle b \rangle)$
2: Reveal $c \leftarrow \langle x \rangle \oplus \langle b \rangle = \langle x \rangle + \langle b \rangle$
3: Compute $[x] \leftarrow c \oplus [b] = c + [b] - 2 \cdot c \cdot [b]$
4: **return** $[a]$

Protocol 24 shows our daBit generation protocol inspired by Attrapadung et al. [AHI+22]. It aims to compute $b_0 \oplus b_1 \oplus b_2 \bmod 2^k$ for some $k > 1$ where b_i has been generated using a PRNG shared between P_{i-1} and P_{i+1}. As $x \oplus y \bmod 2^k$ corresponds to $x + y - 2 \cdot x \cdot y \bmod 2^k$, this could easily be implemented using the

standard multiplication. However, this would $2k$ bits of communication per party. Instead, we use the fact that the original inputs are known to the individual, reducing the complexity to k bits per party.

The protocol has two stages. First, party P_0 shares the result of $b_1 \cdot b_2$, which allows to compute $[b_1 \oplus b_2]$. Then, parties P_1 and P_2 compute shares of $(b_1 \oplus b_2) \cdot b_0$, which is possible because they both know b_0, and together they have a complete sharing of $[b_1 \oplus b_2]$.

Protocol 24: daBit Generation in Rep3

1: For all i, parties P_{i-1} and P_{i+1} generate $b_i \in \{0, 1\}$ using their shared PRNG, define the resulting sharing as $[b_i]$
2: Party P_0 uses the input protocol to input $b_1 \cdot b_2$, resulting in $[x]$.
3: The parties compute $[y] = [b_0] + [b_1] - [x] = [b_0 \oplus b_1]$.
4: Parties P_1 and P_2 use the input protocol to share $([y]_0 + [y]_2) \cdot b_2$ and $[y]_1 \cdot b_2$, respectively, resulting in $[z_1]$ and $[z_2]$.
5: The parties compute $[a] = [y] + [b_2] - 2 \cdot ([z_1] + [z_2])$.
6: **return** $[a], \langle b \rangle$ where $\langle b \rangle_i = b_i$

The correctness can be seen as follows:

$$\begin{aligned} a &= y + b_2 - 2 \cdot (z_1 + z_2) \\ &= y + b_2 - 2 \cdot b_2([y]_0 + [y]_1 + [y]_2) \\ &= y + b_2 - 2 \cdot b_2 \cdot y \\ &= y \oplus b_2 \\ &= (b_0 \oplus b_1) \oplus b_2. \end{aligned}$$

Furthermore, the security follows from the fact that the parties only use secure lower-level primitives.