Honest Majority MPC with Abort with Minimal Online Communication ⋆

Daniel Escudero[1] and Anders Dalskov[2]

[1] `escudero@cs.au.dk` Aarhus University, Denmark
[2] `anderspkd@fastmail.com` Partisia, Denmark

Abstract. In this work we focus on improving the communication complexity of the *online phase* of honest majority MPC protocols. To this end, we present a general and simple method to compile arbitrary secret-sharing-based passively secure protocols defined over an arbitrary ring that are secure up to additive attacks in a malicious setting, to actively secure protocols with abort. The resulting protocol has a total communication complexity in the online phase of $1.5(n-1)$ shares, which amounts to 1.5 shares per party asymptotically. An important aspect of our techniques is that they can be seen as generalization of ideas that have been used in other works in a rather *ad-hoc* manner for different secret-sharing protocols. Thus, our work serves as a way of unifying key ideas in recent honest majority protocols, to understand better the core techniques and similarities among these works. Furthermore, for $n = 3$, when instantiated with replicated secret-sharing-based protocols (Araki et al. CCS 2016), the communication complexity in the online phase amounts to only 1 ring element per party, matching the communication complexity of the BLAZE protocol (Patra & Suresh, NDSS 2020), while having a much simpler design.

1 Introduction

Multiparty Computation, or MPC for short, is a cryptographic technique that allows multiple parties to compute a given function f on private inputs without revealing anything beyond the output of the computation, even if an adversary collectively corrupts a subset of the parties. Different types of MPC protocols exist depending on the desired security level and desired guarantees about the output of the computation. For example, regarding the level of security, a typical dividing line lies in the fraction of parties is allowed to corrupt: In the dishonest majority setting the adversary is allowed to corrupt all but one of the parties, whereas in the honest majority setting the adversary can only corrupt less than half of the parties. The adversary in the former scenario is much stronger and therefore much harder to achieve, and protocols in this

⋆ This work has been supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme under grant agreements No 669255 (MPCPRO), and the Danish Independent Research Council under Grant-ID DFF-6108-00169 (FoCC).

setting, like [KOS16,DKL$^+$13,KPR18,BCS19,CDE$^+$18], are computationally expensive and must rely in computational hardness assumptions. In contrast, honest majority protocols are possible without relying on computational assumptions [BFO12,BTH06,DN07], which makes them more resilient to attacks and also more efficient due to their simplicity and non-reliance on a computational security parameter.

Another dividing line is drawn with respect to the type of corruption the adversary is allowed to make. Typically, two types of corruptions are considered: passive and active corruptions, with the former type consisting of corrupt parties respecting the protocol specifications (but trying to learn as much as they can from sent/received messages); in contrast, actively corrupt parties can deviate arbitrarily. Finally, regarding the output of the computation there are three typical notions considered: guaranteed output delivery, where the honest parties must be able to get output regardless of the adversarial behavior, fairness, where the honest parties must get output *if* the adversary gets output, and security with abort, where either the honest parties get the correct output or they abort.

Many different protocols are designed and optimized for different scenarios and use-cases. However, in spite of being an active and fruitful research field for more than three decades, state-of-the-art MPC protocols still add a considerable overhead with respect to plain "insecure" computation, which makes some potential applications out of reach for the time being. Most of these complexities appear from the interactive nature of MPC, which requires parties to constantly communicate large amounts of data distributed across multiple rounds, which is severely penalized by slow networks and largely distributed parties. Hence, an important task in MPC today is minimizing the communication complexity of the protocols for all kind of adversarial settings. One successful approach at achieving this consists of splitting the computation into an offline and online phase [Bea92], with the former consisting of all the interaction that is independent of the parties' inputs and the latter, which tends to be orders of magnitude more efficient, made of the execution that requires knowledge of the parties' inputs. Given this separation, it is natural to optimize mostly the online phase since it dictates the total latency from the time the inputs are provided to the time the output is obtained.

Among all the possible adversarial settings, it is fair to say that honest majority MPC with abort is one of the scenarios that has received a lot of attention due to its practical, concrete efficiency [CGH$^+$18,LN17,NV18,ADEN19,EKO$^+$20], and it has been already used for concrete applications like secure training and prediction of machine learning models [WGC19,MR18,DEK20,CCPS19,AEV21]. Moreover, several recent works have focused on improving the concrete communication complexity of protocols in these settings. For example, BLAZE [PS20] achieves secure computation for three parties and one active corruption over rings with a communication complexity in the online phase of only 1 ring element per party. For an arbitrary number of parties, very recently, Goyal and Song [GS20] showed how to improve the overall communication complexity of Shamir-based honest majority protocols by presenting a novel method to check the correctness of multiplication gates with a communication complexity that is essentially independent of (specifically, logarithmic in) the circuit size, achieving an amortized communication complexity of 6 field elements per multiplication gate, distributed as 4 elements in the offline phase and 2 elements in the online phase.

1.1 Our Contribution

In this work we focus on improving the communication complexity of the *online phase* of honest majority MPC protocols that achieve security with abort. To this end, we present a general method to obtain from *any* secret-sharing-based passively secure protocol defined over an arbitrary ring that is secure up to additive attacks in a malicious setting, an actively secure protocol with abort over the same ring. The resulting protocol has a total communication complexity in the online phase of $1.5(n-1)$ shares, where n is the number of parties, which amounts to 1.5 shares per party asymptotically. For three parties, when instantiated with protocols based on replicated secret-sharing [AFL^{+}16], the communication complexity in the online phase amounts to $1.5 \cdot 2 = 3$ shares in total, so only 1 share element per party, which matches the communication complexity of state-of-the-art protocols like BLAZE [PS20], while having a much simpler and generalizable design.

In addition to this, our construction has the appealing feature that, in the online phase, a secure dot product of arbitrary length can be computed with a communication complexity that is independent of the length of the input vectors.[3] This is in contrast to other protocols, specially these in the dishonest-majority setting, that require L secure multiplications to produce a dot product of length L. This feature enables highly efficient secure linear algebra, which can be applied for example to linear machine learning models such as linear regression or neural networks.

Our main protocol, presented in Section 3, is considerably simple and general, and provides great efficiency. Furthermore, it achieves the strongest privacy notion in the online phase, namely, perfect security. On top of matching the online complexity of state-of-the-art protocols, our protocol allows us to interpret the core techniques behind some of the most efficient protocols for specific settings like 3 and 4-party computation, namely [CCPS19,PS20,CRS20], in a unified framework which highlights the main tools used in these works to achieve such low communication complexity. These protocols are constructed in an ad-hoc manner, introducing specific building blocks that seem to be inherently entangled to the particular setting (3-4 parties). However, our techniques enable the identification of a common and generalizable pattern behind these constructions. This is generally very useful as, on top of achieving good efficiency for more general settings instead of only scenarios with 3 and 4 parties, it enhances the understanding of these protocol by establishing clear relations among their design. We discuss this in Section 6.

[3] More precisely, at the time of securely computing the dot product only the cost of a single multiplication must be paid. However, this does not rule out the potential cost that had to be paid to obtain shares of the inputs to this dot product in a first place.

The communication pattern of our protocol allows almost half of the parties to be shut down during most of the online phase, which saves in costs and reduces communication channels. However, in some concrete settings it would be ideal if these parties could be shut down during *all* of the online phase, as this could potentially help saving in operational costs. As an additional contribution, we present in Section 4 a variant of our first protocol that allows to shut down essentially half of the parties during the whole online phase.

Finally, we note that the description of our protocols from Sections 3 and 4 considers computation over a field $\mathbb{F}$. However, the works discussed in Section 6 operate over a ring $\mathbb{Z}_{2^k}$, so our protocols are in principle not compatible. To alleviate this disparity, we discuss in Section 5 how to extend our protocols from Sections 3 and 4 to the ring setting. One of the necessary steps to achieve this involves extending the results from [GS20], which are set over fields, to the ring setting, which may be of independent interest.

1.2 Overview of our Techniques

First Protocol Our compiler is conceptually simple and efficient, and requires little modifications to the underlying MPC protocol that is used as a basis. We achieve our results by leveraging a function-dependent preprocessing that reduces the communication complexity of a multiplication gate in the online phase to that of opening one single shared element, coupled with the simple but crucial observation that, in the honest-majority setting, only $t + 1$ parties are required to open a shared value non-robustly.

To provide a high level description of the techniques mentioned above, let us illustrate our compiler for the case of Shamir secret sharing over a field $\mathbb{F}$. Let $[x]_d$ denote a degree-d sharing of $x \in \mathbb{F}$. We assume the existence of a method to multiply two shared values $[z]_t \leftarrow [x]_t \cdot [y]_t$, where $z = x \cdot y + \delta$ and δ is an additive error known by the adversary in the case of an active attack. For our compiled protocol, we define an alternative type of sharings: We write $\langle x \rangle$ if the parties have shares of a random mask $[\lambda_x]_t$, together with the public value $e_x = x - \lambda_x$. This method for secret-sharing is inspired in the work of Ben-Efraim et al. [BNO19], which shows how to reduce the communication of the SPDZ protocol by half by making use of a circuit-dependent preprocessing. Furthermore, the core idea of having a masked version of a secret together with shares of the mask has been used already in previous works, such as [KKW18] and the works we consider in Section 6. We write $\langle x \rangle = ([\lambda_x]_t, e_x)$.

Processing addition gates in the scheme $\langle \cdot \rangle$ can be done locally, while processing multiplication gates require some interaction: To obtain $\langle x \cdot y \rangle$ from $\langle x \rangle$ and $\langle y \rangle$ we assume that the parties have shares $[\lambda_x \cdot \lambda_y]_t$ from a preprocessing phase, which we produce in our work via the protocol from [GS20]. Let $[\lambda_z]_t$ be the random mask that will be used for this multiplication. The parties can obtain $e_z = x \cdot y - \lambda_z$ by opening $e_x e_y + e_x [\lambda_y]_t + e_y [\lambda_x]_t + [\lambda_x \lambda_y] - [\lambda_z]_t$, thus preserving the invariant. Notice that the shares $([\lambda_x]_t, [\lambda_y]_t, [\lambda_x \lambda_y]_t)$ correspond to circuit-dependent multiplication triples, which can be preprocessed very efficiently using the novel batch-checking technique of Goyal and Song [GS20].

Using the techniques sketched above, each multiplication gate in the online phase reduces to opening one single shared value. However, the most efficient method for achieving this, which uses the "king idea" from [DN07], requires $n-1$ parties to send their share to one single party (the "king"), who then reconstructs the secret and sends the result to the other parties. Overall, this implies an overall communication complexity of $2 \cdot (n-1)$ field elements. To further reduce this count, we notice that it is not necessary for all the parties to send their share to the king. Indeed, a subset of t parties, plus the king, suffices, as these together hold $t+1$ shares—enough to reconstruct the secret. Assuming $n = 2t+1$, this reduces the number of field elements transmitted to $t + (n-1) = \frac{3}{2}(n-1)$.

The optimization above comes with a downside in terms of security: By using $t+1$ shares to reconstruct the secret rather than all the n shares, it is possible for a malicious party to fool an honest king into reconstructing an incorrect value.[4] To overcome this issue we observe that, even if the intermediate shares were opened non-robustly using only $t+1$ shares, the parties still have full degree-t shares of these values. To verify that the openings were done correctly, we let the parties take a random linear combination of these shared values, open this single result *robustly* (i.e. using all the shares), and then compare the opened value against the corresponding combination of the values that were non-robustly opened during the execution of the protocol. This idea can be seen as an adaptation of the "partial opening" procedure of the SPDZ protocol to the honest majority setting.

Second Protocol Due to the fact that we open shared values using only $t+1$ parties, we can set the communication pattern of the protocol sketched above in such a way that only some *fixed* subset of $t+1$ parties communicates during the online phase, until the output phase. In the final check, the parties in this subset would broadcast the intermediate opened values to the other t parties, and the check would then be executed.

The fact that all the parties need to be available for the final check is a downside if one wants to shut down these parties "for good", once the online phase has started. To allow the $t+1$ parties that are active during the online phase to perform the final correctness check without the help of the other parties, we resort to a rather standard technique in the dishonest majority setting involving the use of MACs: A value x is shared as $[x]$ together with $[r \cdot x]$ for a random and global $[r]$. Then, to check that $[x]$ is opened correctly, the $t+1$ parties that were active during the online phase (among which a dishonest majority may be corrupt) use the MAC $[r \cdot x]$ and the key $[r]$. Furthermore, this check can be easily batched so that its communication complexity is independent of the number of openings being verified.

Extension to Rings Most of the techniques used in our work carries over seamlessly to the ring setting. For example, the online phase of our first protocol from

[4] As presented here, a corrupt king can already disrupt the reconstruction of the secret even if all n shares are used. This is handled in [DN07] by considering multiple kings and using error correction techniques.

Section 3 does not exploit any specific property of fields, and so works the same over $\mathbb{Z}_{2^k}$. The main issue appears in the offline phase, given that the protocol from [GS20] that we use to produce the necessary preprocessing material only works over fields. To extend the protocol from [GS20] to the ring setting, we observe that the core technique used in this protocol is basic polynomial interpolation, which can be made to work over $\mathbb{Z}_{2^k}$ by taking a Galois ring extension of large enough degree, as done in [ACD$^+$19]. This idea was already used in [BGIN19] for the three-party setting, whereas our results are applicable to an arbitrary number of parties and any linear secret-sharing scheme.

Finally, the online phase of our second protocol from Section 4 does not work directly over $\mathbb{Z}_{2^k}$ as it relies on the AMD code $(x, r \cdot x, r)$, which does not provide any integrity of $\mathbb{Z}_{2^k}$ due to the lack of invertible elements. Fortunately, the work of [CDE$^+$18] shows how to extend this integrity mechanism to the ring $\mathbb{Z}_{2^k}$ by operating over a larger ring $\mathbb{Z}_{2^{k+\kappa}}$, where κ is the statistical security parameter. We show in Section 5 that this technique can be also applied to our protocol.

1.3 Related Work

Honest majority MPC with a small communication footprint has been studied in a number of works in the last decade or so. One attractive feature of such protocols, is that they work well for a large number of parties where it is reasonable to assume that not everyone colludes (in particular, where only a minority colludes). This setting was investigated in [BHKL18], which demonstrated a concretely efficient protocol for large number of parties, based on a protocol adapted from [BTH08].

While [BTH08] has a linear communication complexity, recent research (which we have already mentioned) have brought this down to logarithmic [GS20]. Moreover, for the specific setting of 3 and 4 parties, specialized protocols have been shown to be concretely efficient.

Compiling honest majority protocols from passive to active security have also been studied previously. [CGH$^+$18] show how to efficient convert a passively secure honest majority protocol with the same properties we require (security against an active adversary up to additive attacks), into an active secure protocol with a concretely very small overhead. The authors [ADEN19] show, employing a similar approach as [CGH$^+$18], how to get a similar compiler for ring based protocols.

2 Preliminaries

2.1 Notation

We let n be the number of parties among which t are corrupt. In the honest majority setting it holds that $t < n/2$, but for simplicity in the presentation, we will assume that $n = 2t + 1$.[5] Let $\mathcal{R}$ be a ring of the form $\mathbb{Z}_{p^k}$ or $\mathsf{GF}(p^k)$ for some prime p and some non-negative integer k. We write $s \in_R A$ when s is uniformly sampled from a finite set A. We let κ denote the statistical security parameter.

2.2 Security Definition

In this work we assume a synchronous network of secure point-to-point channels, together with a broadcast channel. We consider simulation-based security, which can be either the universally composability framework [Can01] or the stand-alone setting [Gol01] as our techniques apply to both. We focus on providing security with abort, in which the adversary can make the honest parties abort at any point in the protocol; in particular, the adversary itself may get output before the honest parties and immediately abort the computation. We assume the existence of a broadcast channel with abort (that is, the parties either get the value that was broadcast, or they abort), and we assume that when an honest party aborts, *all* honest parties abort, which can be easily achieved using the broadcast channel. Since the broadcast channel allows for abort, it can be efficiently instantiated via echo-broadcast by letting the sender distribute the value to be broadcast via the point-to-point channels, and then letting the receiving parties exchange hashes of his value and abort if an inconsistency is detected.

When measuring communication complexity, $M + \mathsf{BC}(N)$ means that M ring elements are communicated over point-to-point channels, and N ring elements are communicated over the broadcast channel. With the broadcast method sketched above, we would have that $\mathsf{BC}(N) = (n-1) \cdot N$, which amounts to the messages sent by the broadcaster (we ignore the cost of the hash exchange as it is independent of N).

2.3 Linear Secret Sharing

We consider a linear secret sharing scheme (LSSS) $[\cdot]$ over $\mathcal{R}$. Such a scheme for our purposes consists of a randomized injective function $\mathsf{share} : \mathcal{R} \to (\mathcal{R}^m)^n$ such that the following holds for all $x \in \mathcal{R}$. Below, we let $\mathsf{share}(x) = \{x_i\}_{i=1}^n$.

- *Privacy.* For any subset $J \subseteq \{1, \ldots, n\}$ such that $|J| \leq t$, the mutual information between $\{x_i\}_{i \in J}$ and x is 0.
- *Reconstruction.* There is a function $\mathsf{rec} : (\mathcal{R}^m)^{t+1} \to \mathcal{R}$ such that, for any subset $J \subseteq \{1, \ldots, n\}$ such that $|J| = t + 1$, it holds that $\mathsf{rec}\left(\{x_i\}_{i \in J}\right) = x$ (the function rec is implicitly taking the set J as a parameter).
- *Linearity.* Given $\{y_i\}_{i=1}^n = \mathsf{share}(y)$, it holds that $\{x_i + y_i\}_i$ lies in the image of $\mathsf{share}(x + y)$.[6]

[5] Some technical complications arise if $2t + 1 < n$, like the fact that the set of honest parties is strictly larger than $t+1$, so different subsets of honest parties may reconstruct shared secrets to different values. This is not really a problem with the protocols, but it introduces some notational overhead that we would like to avoid.

[6] This extends naturally to multiplication by constants. Furthermore, addition by a constant z can be obtained by the parties generating public shares $[z]$ using some canonical randomness, and then using the linearity between shared values.

To ease the notation a bit, we may write $[x]^J := \{x_i\}_{i \in J}$, and $[x] := [x]^{\{1,\ldots,n\}}$.

The definition of rec is extended to more than $t+1$ shares as follows: Let $K \subseteq \{1,\ldots,n\}$ such that $|K| \geq t+1$. We write $\mathsf{rec}\left(\{x_i\}_{i\in K}\right) = x$ if, for all $J \subseteq K$ with $|J| = t+1$, it holds that $\mathsf{rec}\left(\{x_i\}_{i\in J}\right) = x$. Else, we write $\mathsf{rec}\left(\{x_i\}_{i\in K}\right) = \perp$. In the former case we say that the shares $\{x_i\}_{i\in K}$ are *consistent*, and in the latter case we say they are *inconsistent*.

An important factor of honest-majority secret sharing (and one which does not hold for a dishonest-majority) is that, when the sharings are consistent, it is ensured that the correct value will be reconstructed. More precisely, if the adversary modifies the t entries in $[x]$ corresponding to the corrupt parties, but if $\mathsf{rec}([x]) = x' \neq \perp$, then it is guaranteed that $x' = x$. This is because, if H denotes the set of honest parties, which satisfies $|H| = n - t = t + 1$, $\mathsf{rec}([x]) = x' \neq \perp$ implies that $\mathsf{rec}\left([x]^H\right) = x'$, but since the shares $[x]^H$ are not modified, this implies that $x' = x$.

2.4 Reconstruction Protocols

We defined secret sharing above as a set of functions, but in practice it is used as a set of protocols for distributing and reconstructing data among n parties. In this section we discuss different ways in which the parties can reconstruct a shared value $[x] = \{x_i\}_i$, where party P_i has the share x_i. We consider two variants: Robust opening, where the value that is opened is guaranteed to be correct, and a much more efficient non-robust—or "loose"—opening, where the value that is opened may be incorrect. As we will see in subsequent sections, a key optimization in this work lies in using loose openings for the majority of our protocols in way which does not harm correctness or privacy.

Improving Communication Complexity The public reconstruction protocols Π_{Rec} and Π_{LooseRec}, as described in Fig. 1, have a communication complexity that is quadratic in the number of parties as they require (almost) all parties sending shares to all other parties. This can be improved to linear communication in n as follows: For $\Pi_{\mathsf{LooseRec}}([x])$, the $t+1$ parties $P_1, \ldots, P_{t+1}$ send their shares to P_1, who reconstructs x using these shares, and broadcasts this value to all parties. The total communication is then t ring elements over point-to-point channels, plus 1 ring element over the broadcast channel.

Unfortunately, this does not work for Π_{Rec} since the king can lie about the reconstructed value. To handle this, one can use the techniques from [DN07] to obtain linear *amortized* complexity in n (for multiple simultaneous openings). In a nutshell, this works by batching a sequence of secrets to be opened into a vector, and encoding this vector using a linear error-correcting code. Then, each secret in the codeword is opened by using the king idea from above with a different king for each symbol, and then error correction/detection is applied to the resulting opened codeword.

Reconstruction Protocols

ROBUST RECONSTRUCTION

$\Pi_{\mathsf{PrivRec}}([x], i)$**.** Robustly reconstructs x towards party P_i. Each party P_j sends its share x_j to P_i, who invokes rec on the received shares and outputs what rec outputs.

$\Pi_{\mathsf{Rec}}([x])$**.** Robustly reconstructs x towards all parties.

1. Parties call $\Pi_{\mathsf{PrivRec}}([x], i)$ for all $i \in \{1, \dots, n\}$.
2. If P_i outputs $\perp$ from its reconstruction above, then P_i aborts.

LOOSE RECONSTRUCTION

$\Pi_{\mathsf{LoosePrivRec}}([x], i)$**.** Non-robustly reconstructs x towards party P_i. Each party P_j for $j = 1, \dots, t+1$ sends its share x_j to P_i, who invokes rec on the received shares and outputs whatever rec outputs.

$\Pi_{\mathsf{LooseRec}}([x])$**.** Non-robustly reconstructs x towards all parties. Each party P_j for $j = 1, \dots, t+1$ broadcasts its share x_j. Each party invokes rec on the received shares and outputs whatever rec outputs.

Fig. 1. Different reconstruction protocols we use in our work

2.5 Sampling Shares of Random Values

We assume two functionalities for sampling shared randomness.

- $\mathcal{F}_{\mathsf{Rand}}$: Produces a shared value $[r]$ where $r \in_R \mathcal{R}$.
- $\mathcal{F}_{\mathsf{Coin}}$: Produces a value $r \in_R \mathcal{R}$ known to all parties.

$\mathcal{F}_{\mathsf{Rand}}$ can be instantiated, for example, using the techniques outlined in [DN07], which are based on Vandermonde matrices, although more efficient instantiations of $\mathcal{F}_{\mathsf{Rand}}$ exist for particular secret-sharing schemes like replicated secret sharing cf. [ADEN19,CGH$^+$18]. $\mathcal{F}_{\mathsf{Coin}}$ can be instantiated by calling $\mathcal{F}_{\mathsf{Rand}}$ to sample $[r]$, followed by $r \leftarrow \mathsf{rec}([r])$, but, like $\mathcal{F}_{\mathsf{Rand}}$, this functionality can be instantiated more efficiently in certain cases.

2.6 Correct Multiplication

We assume a functionality $\mathcal{F}_{\mathsf{CorrectMult}}$ that takes as input two shared values $[x], [y]$, and returns $[x \cdot y]$. This functionality will be only used in the preprocessing.

In this work we consider a very efficient instantiation of $\mathcal{F}_{\mathsf{CorrectMult}}$ denoted $\Pi_{\mathsf{CorrectMult}}$ that combines secure multiplication up to additive attacks, together with a method to verify that a multiplication triple is correctly computed. We outline this approach next.

Secure Multiplication up to Additive Attacks Consider a functionality $\mathcal{F}_{\mathsf{MultAddAtt}}$ that takes as input two shared values $[x], [y]$, in addition to a value $\delta \in \mathcal{R}$ from the adversary, and returns $[x \cdot y + \delta]$. As shown in [GIP$^+$14,ADEN19,LN18], this functionality can be instantiated by popular *passively secure* multiplication protocols when set in the active setting, which are considerably more efficient than a protocol that guarantees correctness under an active adversary. This property has already been used as a first step in multiple works to obtain actively secure multiplication [LN18,ADEN19,LN17], and in this work we also leverage the same techniques.

Verifying Multiplication Triples The second ingredient needed to instantiate $\mathcal{F}_{\mathsf{CorrectMult}}$ efficiently is a method to verify that the additive error introduced by $\mathcal{F}_{\mathsf{MultAddAtt}}$ is 0. When $\mathcal{R}$ is a field, a simple and widely used approach involves using the so-called *sacrifice* techniques [DKL$^+$13], where the consistency of a multiplication is checked by computing a random tuple $([a], [b], [c])$ where c is supposed to be equal to $a \cdot b$, and "sacrificing" this tuple in some way to check the consistency of the other. This approach requires opening one value per multiplication being checked. This was later improved in [BFO12] by removing this dependency when many multiplications are being checked, but it still requires parties to compute one extra multiplication per multiplication being checked.

Very recently, this was overcome in the work of [GS20], which presents a method to verify the validity of m multiplications with a communication cost of $O(\kappa \cdot n + n^2)$. In particular, communication is *independent* of m.

Although the techniques introduced in [GS20] are presented in the context of Shamir secret-sharing specifically, it is possible to check that these techniques extend to any linear secret-sharing scheme over fields, and in particular those considered in this work. The only requirement is the existence of a protocol $\Pi_{\mathsf{DotAddAtt}}$ that takes as input two arrays of shared values $([x_i])_{i=1}^{L}, ([y_i])_{i=1}^{L}$ and outputs $[z] = \left[\delta + \sum_{i=1}^{L} x_i y_i\right]$, with δ an error introduced by the adversary. Finally, $\Pi_{\mathsf{DotAddAtt}}$ should have a communication complexity independent of L.

On the other hand, generalizing to rings like $\mathbb{Z}_{2^k}$ is not straightforward, as the techniques in [GS20] rely heavily on polynomial interpolation and other properties that—unlike the case for fields—do not directly hold over $\mathbb{Z}_{2^k}$. In Section 5 we discuss how to extend the check from [GS20] to $\mathbb{Z}_{2^k}$ in order to provide a valid instantiation of $\mathcal{F}_{\mathsf{CorrectMult}}$ over this type of ring.

3 Optimizing the Online Phase

In this section we present our first protocol whose online phase is optimized so that the parties only send, in total, $\frac{3}{2}(n-1) \cdot k \cdot \log(p)$ bits per multiplication gate. On top of being conceptually very simple, our optimization allows for a communication pattern in which only $t+1$ parties are present for most of the online phase, except that the remaining t parties must return for the output phase. In Section 4 we present a protocol for which this is not required, that is, only $t+1$ parties are required to run all of the online phase, including the output phase.

We begin by presenting in Section 3.1 the secret sharing construction we will use in our protocols. Then, in Section 3.2, we present an intuitive overview of our protocol, and finally, in Section 3.3, we describe our protocol in detail, analyze its complexity and discuss its security.

3.1 Masked Secret Sharing

Let $[\cdot]$ be a secret sharing scheme over $\mathcal{R}$, as defined in Section 2.3. We define the following LSSS over $\mathcal{R}$ that builds on top of $[\cdot]$.

- $\mathsf{share}_{\langle\cdot\rangle}(x)$: Sample a random mask $\lambda_x \in \mathcal{R}$, call $\mathsf{share}_{[\cdot]}(\lambda_x)$ and append to each share the value $\mu_x = x - \lambda_x$. We denote this by $\langle x \rangle = ([\lambda_x], \mu_x = x - \lambda_x)$.
- $\mathsf{rec}_{\langle\cdot\rangle}([\lambda_x], \mu_x)$: Call $\lambda_x \leftarrow \mathsf{rec}_{[\cdot]}([\lambda_x])$ and if $\lambda_x \neq \bot$ then output $\lambda_x + \mu_x$, else output $\bot$.

It is easy to see that this new scheme is additively homomorphic. Indeed, given $\langle x \rangle = ([\lambda_x], \mu_x)$ and $\langle y \rangle = ([\lambda_y], \mu_y)$ parties can compute shares of the sum as $\langle z \rangle = ([\lambda_x + \lambda_y], \mu_x + \mu_y)$.

3.2 General Overview

We begin by providing a high-level view of our protocol. To this end, it is instructive to begin with a very simple and naive protocol that makes use of the original LSSS $[\cdot]$, together with correct multiplication triples. We consider this below in Section 3.2, and then discuss our optimizations in Section 3.2.

Naive Protocol As we mentioned in Section 2.3, a crucial property of honest majority LSSS is that either the correct value is reconstructed or an abort signal is generated. This property can be leveraged to obtain simple and efficient actively MPC protocol as follows:

A Naive Protocol

Input phase. To share its input x_i, P_i proceeds as follows:

1. In a preprocessing phase the parties generate consistent shares $[r_i]$ where $r_i \in \mathcal{R}$ is uniformly random and only known to P_i. Such consistent shares can be generated efficiently as sketched in Section 2.5.
2. To share x_i, P_i broadcasts the value $x_i - r_i$, and the parties compute the shares $[x_i] = (x_i - r_i) + [r_i]$. If a proper broadcast channel is used, the resulting shares are guaranteed to be consistent.

Addition gates. Addition gates are handled locally by using the linearity property of the LSSS.

Multiplication gates. To multiply $[x]$ and $[y]$, the parties proceed as follows:

1. In a preprocessing phase, sample random shares $([a], [b], [c])$, where $c = a \cdot b$. Such triples can be generated as presented in Section 2.6, for example.
2. In the online phase, the parties reconstruct $d \leftarrow \mathit{\Pi}_{\mathsf{Rec}}([x] - [a])$, $e \leftarrow \mathit{\Pi}_{\mathsf{Rec}}([y] - [b])$ (aborting if this is either reconstruction outputs $\perp$), and locally compute $[x \cdot y] = d\,[b] + e\,[a] + [c] + d \cdot e$.

Output gates. If party P_i is intended to learn the value of $[x]$, the parties call $\mathit{\Pi}_{\mathsf{PrivRec}}([x], i)$.

It is easy to see that this protocol satisfies security with abort. First, correctness holds given that addition gates are local and the formula used for the multiplication gates satisfies

$$d \cdot b + e \cdot a + c + d \cdot e = x \cdot y,$$

as can be verified. In regards to privacy, begin by observing that every shared value throughout the computation is consistent. This is because consistent sharings are assumed to be produced in the preprocessing, and a proper broadcast channel is used in the input phase, which ensures this also extends to the input sharings and also for subsequent wires in the circuit as these are computed using only linear operations.

Finally, notice that due to the robustness properties of the LSSS and the consistency of the sharings, the adversary cannot cheat in any opening without causing an abort, so the only values opened are the $d = x - a$ and $e = y - b$ from the multiplication gates, which leak nothing about the inputs x and y given that a and b are uniformly random elements in $\mathcal{R}$ unknown to the adversary.

Remark 1. In the template above we pushed all the complexities of dealing with the additive errors to the preprocessing, where the multiplication triples are produced. This is good for the problem we have at hand, which is optimizing the communication complexity in the online phase. However, a different approach would be to deal with the additive errors in a "post-processing" phase, that is, one may allow additive errors during the multiplications in the online phase (for which one could either use potentially incorrect triples, or use the assumed multiplication produce directly in the online phase, avoiding extra preprocessing), and then perform some check that guarantees that these errors are zero.

This is what is done by many of the existing honest-majority protocols that have been proposed in recent years. For example, this approach is taken in [FLNW17], where cut-and-choose and triple sacrificing techniques are used to ensure all multiplications are handled correctly. This approach is also considered in [GS20], where, instead of using their novel triple verification techniques in the preprocessing phase, as we do here, the authors use the check after the online phase has been executed to check all multiplications are correct. [EKO$^+$20] also follows a similar "post-processing" approach.

Optimizing the Naive Protocol As remarked, the basic template presented above can be (and has been) optimized in many different ways. However, in our work we aim at optimizing the online phase as much as possible, which implies that our preprocessing phase may be more inefficient than some of the existing works. The two optimizations we incorporate to the basic template sketched above are the following:

1. We use the secret-sharing scheme $\langle\cdot\rangle$ instead of $[\cdot]$, and we handle multiplication gates as described in Section 3.1 instead of using triples directly in the online phase (correct triples must still be preprocessed, as described in Section 3.1). This lowers the complexity of a multiplication gate from two openings to only one opening.[7]
2. Instead of performing each opening robustly, the parties perform the openings using Π_{LooseRec}, which is cheaper but may cause reconstructed values to be incorrect. After all loose openings are done, but before the final output gates, the correctness of these openings is checked by taking a random linear combination of the opened values, and opening robustly the same linear combination over the corresponding shares.[8]

The first optimization only has an effect on the amount of communication in the online phase. However, the second optimization, on top of reducing the overall amount of bits sent, contributes in a much more impactful way: By using loose openings instead of robust openings, and by cleverly rearranging the communication pattern, the online phase can be run by just the parties

[7] This optimization was already introduced in [BNO19] in the context of the SPDZ protocol.

[8] This is similar to what is done in the SPDZ protocol [DKL$^+$13], where values are "partially opened" for each multiplication gate (that is, without using the MACs), and only at the end of the computation these openings are checked by taking a random linear combination.

$P_1, \dots, P_{t+1}$, while the remaining parties $P_{t+2}, \dots, P_n$ only have to come back for the final check. Removing communication channels among the parties is likely to have a much more noticeable impact in the efficiency than merely lowering the communication complexity. Furthermore, as these servers do not participate for the majority of the computation, this also frees up computing resources and is more energy efficient.

We remark that, even though we described the masked secret-sharing construction and the naive starting protocol over the arbitrary ring $\mathcal{R}$, in our actual protocol below the computation ring $\mathcal{R}$ is assumed to be a field (which we will denote as $\mathbb{F}$). We discuss in Section 5 how to extend our protocol so that it also works over a ring of the form $\mathbb{Z}_{2^k}$.

3.3 Main Protocol

With the above intuitive explanation of our protocol, we proceed to a more formal description of our protocol shown in Fig. 2.

Remark 2. Observe that in the online phase the additions and multiplications can be handled only by $P_1, \dots, P_{t+1}$, by performing the openings only among these parties. Hence, most of the online phase involves communication only among $P_1, \dots, P_{t+1}$, which in practical terms means that parties $P_{t+2}, \dots, P_n$ can go offline until the checking phase is reached. At this point, the offline parties must rejoin the computation, receive the partially opened values from the other parties and participate in the checking and output procedures. Having the ability to shut down parties has many relevant effects in practice. For instance, it can help in saving operational costs, as well as allowing parties to allocate resources more effectively by, say, placing most of the computation on the more powerful servers. Additionally, shutting down communication channels is particularly good in wide area networks, where strong use of communication is heavily penalized.

It can be checked that

$$\mu_x\mu_y + \mu_x\lambda_y + \mu_y\lambda_x + \lambda_x\lambda_y - \lambda_z = x \cdot y - \lambda_z,$$

which shows that the multiplication gates lead to correct $\langle\cdot\rangle$-sharings of the product of the inputs, and in particular shows that the protocol produces the right output if all openings are done correctly. Furthermore, to see that the protocol preserves privacy, observe that, before the checking phase, all intermediate values remain private since the only openings done throughout the protocol are the the values $\mu_x = x - \lambda_x$, and since the masks λ_x are uniformly random and secret-shared among the parties, μ_x looks uniformly random as well.

It only remains to be checked that openings are correct with high probability. As in the protocol, let $[\eta_1], \dots, [\eta_M]$ be the shares that were loosely opened to $\eta'_1, \dots, \eta'_M$ during the computation phase. Write $\eta'_i = \eta_i + \delta_i$, that is, we express the loosely opened value as the correct one plus an additive error from the adversary. In the checking phase, parties open

$$\eta' - \eta = \sum_{i=1}^{M} \alpha_i(\eta'_i - \eta_i) = \sum_{i=1}^{M} \alpha_i(\eta_i + \delta_i - \eta_i) = \sum_{i=1}^{M} \alpha_i\delta_i,$$

Protocol $\mathit{\Pi}_{\mathsf{MPCLowOnline}}$

OFFLINE PHASE

- For every wire in the circuit x the parties call $[\lambda_x] \leftarrow \mathcal{F}_{\mathsf{Rand}}$.
- For every input gate x corresponding to a party P_i, the parties call $\mathit{\Pi}_{\mathsf{PrivRec}}([\lambda_x], i)$.
- For every multiplication gate with inputs x, y and output z, the parties call $[\lambda_x \lambda_y] \leftarrow \mathcal{F}_{\mathsf{CorrectMult}}([\lambda_x], [\lambda_y])$.

ONLINE PHASE

Input Gates. For every input gate x owned by party P_i, the parties do the following:

1. P_i uses the broadcast channel to send $\mu_x = x - \lambda_x$ to all parties.
2. Upon receiving this value, the parties set $\langle x \rangle = ([\lambda_x], \mu_x)$

Addition Gates. For every addition gate with inputs $\langle x \rangle = ([\lambda_x], \mu_x)$ and $\langle y \rangle = ([\lambda_y], \mu_y)$, the parties locally set $\langle x + y \rangle = ([\lambda_x] + [\lambda_y], \mu_x + \mu_y)$.

Multiplication Gates. For every multiplication gate with inputs $\langle x \rangle = ([\lambda_x], \mu_x)$ and $\langle y \rangle = ([\lambda_y], \mu_y)$, the parties proceed as follows:

1. Call $\mu_z \leftarrow \mathit{\Pi}_{\mathsf{LooseRec}}(\mu_x\mu_y + \mu_x[\lambda_y] + \mu_y[\lambda_x] + [\lambda_x\lambda_y] - [\lambda_z])$
2. Set $\langle x \cdot y \rangle = ([\lambda_z], \mu_z)$.

Checking Phase. Before any output gate is reconstructed, the parties proceed as follows. Let $[\eta_1], \ldots, [\eta_M]$ be the shares that were loosely opened to $\eta'_1, \ldots, \eta'_M$ during the computation phase (these correspond to $[\mu_z]$ for every wire z that is the output of a multiplication gate).

1. For $i = 1, \ldots, M$ call $\alpha_i \leftarrow \mathcal{F}_{\mathsf{Coin}}$.
2. Let $[\eta] = \sum_{i=1}^{M} \alpha_i \cdot [\eta_i]$ and $\eta' = \sum_{i=1}^{M} \alpha_i \cdot \eta'_i$. The parties call $z \leftarrow \mathit{\Pi}_{\mathsf{Rec}}(\eta' - [\eta])$. If $z \neq 0$ then the parties abort.

Output Gates. If the parties did not abort above, then for every output gate $\langle x \rangle = ([\lambda_x], \mu_x)$ that is supposed to be learned by P_i, the parties call $\mathit{\Pi}_{\mathsf{PrivRec}}([\lambda_x], i)$, and if it succeeds then P_i outputs $\mu_x + \lambda_x$.

Fig. 2. Our first protocol with low online communication. The algebraic structure used for the computation is assumed to be a field $\mathbb{F}$ with $|\mathbb{F}| \geq 2^\kappa$.

and check whether it is 0. Because each α_i is sampled *after* the adversary introduces the error δ_i (specifically, α_i is sampled after the computation concludes), the above sum is 0 with high probability if and only if $\delta_i = 0$ for all $i = 1, \dots, M$.

Communication Complexity. The offline phase of our protocol consists of sampling the masks $[\lambda_x]$ and preprocessing the triples $([\lambda_x], [\lambda_y], [\lambda_z])$. This process overall has a linear communication complexity with respect to the number of parties n. For the online phase, which is of our particular interest, the total communication per multiplication gate, ignoring the check phase and the calls to $\mathcal{F}_{\mathsf{Coin}}$, amounts to one call to Π_{LooseRec} which equals $t + \mathsf{BC}(1)$ ring elements. Using the broadcast protocol with abort sketched in Section 2.2, this amounts to $t + (n-1) = \frac{3}{2}(n-1)$.

4 Removing the Extra Parties from the Output Phase

As we discussed before, our protocol from Section 3 has the appealing feature that most of the online phase can be run only by $t+1$ parties. More precisely, the only part in which the extra t parties are required is in the preprocessing and in the output phase. This not only helps in saving in communication, but it allows these extra t parties to be turned off during the online phase. However, it would be ideal if the online phase could be run in its entirety, including the output phase, by $t+1$ parties only. This would allow these extra t parties to be switched off "for good" once the offline phase is finished, which can represent noticeable savings in many practical scenarios, as for example, when parties run in servers that are charged for usage time.

In this section we show that this is indeed possible without affecting the communication complexity of the online phase. This is done at the expense of having more communication in the offline phase, and slightly more computation in the online phase. This comes at the expense of requiring the ring $\mathcal{R}$ to be a field $\mathbb{F}$, although we show in Section 5 how to overcome this limitation.

We begin by providing a general overview of our protocol in Section 4.1, and then we describe our protocol in detail in Section 4.2, together with its complexity analysis and security proof.

4.1 General Overview

The intuition behind our protocol is that, if $2t+1$ participates in the offline phase, then it suffices—by being clever as to what kind of material is produced during preprocessing—to only have $t+1$ parties participate in the online phase.

More precisely, consider the case of a field $\mathbb{F}$, and let us revisit our protocol from Section 3. Recall that the online phase can be run by $t+1$ parties only since, after the preprocessing is done, only openings are required during the online phase. However, since the threshold of the secret sharing scheme is t, $t+1$ parties only do not provide enough redundancy, which allows an active adversary to additively tamper these opened values. We solved this issue in our previous protocol from Section 3 as follows. Let $[\eta_1], \dots, [\eta_M]$ the shares that were opened to $\eta'_1, \dots, \eta'_M$.

1. The parties sample random public values $\alpha_1, \ldots, \alpha_M \in \mathbb{F}$. Let $[\eta] = \sum_{i=1}^{M} \alpha_i [\eta_i]$ and $\eta' = \sum_{i=1}^{M} \alpha_i \eta'_i$.
2. It suffices to check that $[\eta]$ opens to η', which is done by opening $\eta' - [\eta]$ using all of the $2t + 1$ parties, which guarantees that this value is correct.

In the protocol we present in this section we replicate the same steps, except that we do not want to involve the extra t parties to open $[\eta]$ robustly. As mentioned above, $t + 1$ parties alone cannot open $[\eta]$ robustly, and so we resort to a technique from the dishonest majority MPC literature: Instead of sharing a value $v \in \mathbb{F}$ as $[v]$, it is shared as $([v], [r \cdot v])$, where $r \in \mathbb{F}$ is a global (i.e. it is the same for all shared values) random key that is also shared as $[r]$. This "new" sharing scheme is easily verified to be linear and therefore its invariant can be kept throughout the whole computation.

The fact that $[r]$ is hidden, and that the sharing $([v], [r \cdot v])$ is linear, can be used to ensure correctness, provided errors inserted by the adversary during the check are independent of the honest parties shares. In order to enforce this, we take a "commit-and-open" approach. In more detail, to open a value $[v]$, each P_i first commits to their share $v^{(i)}$ of v using an ideal commitment functionality $\mathcal{F}_{\mathsf{Commit}}$, after which they call Π_{LooseRec} where the received share is checked against the committed value. While this still permits the adversary to reveal the *wrong* value, the error that is induced is nevertheless going to be independent of the honest parties shares. Note that this commit-and-open is only needed in the checking and output phase of the protocol; during computation, invoking Π_{LooseRec} suffices.

Using this technique, once the M values $[\eta_1], \ldots, [\eta_M]$ have been opened to $\eta_1 + \delta_1, \ldots, \eta_M + \delta_m$ by the $t + 1$ parties running the online phase, these parties can also check the correctness of these openings without involving the other t parties by using the extra sharings $[r \cdot \eta_1], \ldots, [r \cdot \eta_M], [r]$ as follows:

1. Like in our protocol from Section 3, begin by sampling random public values $\alpha_1, \ldots, \alpha_M$. Let $[r \cdot \eta] = \sum_{i=1}^{M} \alpha_i [r \cdot \eta_i]$ and $\eta' = \sum_{i=1}^{M} \alpha_i(\eta_i + \delta_i)$. Also, let $[\beta] = [r \cdot \eta] - \eta' \cdot [r]$.
2. The parties loosely open $\beta + \epsilon \leftarrow [\beta]$ and abort if this is not equal to 0.

It is easy to see that the check passes if and only if $r \cdot \left(\sum_{i=1}^{M} \alpha_i \delta_i\right) = \epsilon$, which happens with probability at most $1/|\mathbb{F}|$ if there is at least one δ_i that is non-zero.[9] This idea is already widely used in other dishonest-majority protocols like [KOS16,DKL$^+$13,KPR18,BCS19,CDE$^+$18], but, to the best of our knowledge, our work is the first to make use of this technique in the honest-majority setting with the goal of reducing the amount of parties needed for robust opening.[10]

[9] For simplicity we assume that $|\mathbb{F}|$ is big enough so that $1/|\mathbb{F}|$ is negligible. The general case is easily handled by iterating the current construction with multiple r's.

[10] [CGH$^+$18] also uses this idea, but in a different way and with a different goal. In [CGH$^+$18], the MACs are used not to ensure correct openings, since a complete honest majority is used for reconstruction, but to disallow additive attacks after multiplications.

Observe that the communication complexity of the online phase of this new approach is essentially the same as the one from the protocol in Section 3, given that the online phase is comprised mostly of loose openings. However, the offline phase of this new approach is more expensive since, on top of generating the necessary multiplication triples, it also generates the necessary MACs, which essentially doubles the required amount of preprocessed material.

We present the protocol in full detail in the next section.

4.2 Main Protocol

We present our optimized protocol in full detail in Fig. 3.

As we mentioned already, this protocol can be seen as an adaptation of [BNO19] to the honest majority setting, where a dishonest majority is used for the online phase. Its security follows directly from the security of [BNO19], which essentially boils down to the following two observations. First, privacy is preserved throughout the protocol execution because of the same reason as in $\Pi_{\mathsf{MPCLowOnline}}$ (only masked values μ_x are ever opened). Secondly, in the checking phase the sharing $[\beta]$ is opened to $\beta + \epsilon$. It is easy to see that $\beta + \epsilon$ is equal to 0 if and only if $r \cdot \sum_{i=1}^{M} \alpha_i \delta_i = \epsilon$ and that this happens with low probability if there is some $\delta_i \neq 0$. Indeed, in this case we have $\sum_{i=1}^{M} \alpha_i \delta_i \neq 0$ with overwhelming probability, which in turn implies that $r = \epsilon \cdot \left(\sum_{i=1}^{M} \alpha_i \delta_i\right)^{-1}$. However, this cannot be the case except this negligible probability as it implies the adversary could compute r before it was opened; an impossible task considering r is a uniform random value. A similar argument holds for the check done in the output phase.

Communication Complexity. Our protocol communicates a linear number of field elements (in n) in the offline phase, as in protocol $\Pi_{\mathsf{MPCLowOnline}}$. For the online phase, ignoring the final checking phase, the protocol requires $t + \mathsf{BC}(1)$ field elements communicated per multiplication gate. Since the broadcast involves only the parties $P_1, \ldots, P_{t+1}$, we have that $\mathsf{BC}(1) = t$, so the communication per multiplication gate is of $t + t = n - 1$ field elements.

5 Extension to Rings

In some natural scenarios, one would like to perform computation over $\mathbb{Z}_{2^k}$ rather than $\mathbb{F}$. Arithmetic modulo 2^{32} and 2^{64} is supported implemented native in hardware (as it corresponds to operating with fixed width integers) and is thus very efficient. Moreover, some important primitives like secure truncation and secure comparison are more efficient when instantiated over rings, as has been verified experimentally in works like [DEK20,DEF$^+$19]. As such, the task of extending our protocols from a field $\mathbb{F}$ to a ring of the form $\mathbb{Z}_{2^k}$ is a well motivated one, and this is an issue we address in this section.

To begin, we identify the parts that break in each of our two protocols when ported from $\mathbb{F}$ to $\mathbb{Z}_{2^k}$. We start with our protocol from Fig. 2 in Section 3. The

Protocol $\Pi_{\mathsf{MPCLowChannels}}$

OFFLINE PHASE

- The parties call $[r] \leftarrow \mathcal{F}_{\mathsf{Rand}}$.
- For every wire in the circuit x the parties call $[\lambda_x] \leftarrow \mathcal{F}_{\mathsf{Rand}}$.
- For every input gate x corresponding to a party P_i, the parties call $\Pi_{\mathsf{PrivRec}}([\lambda_x], i)$.
- For every multiplication gate with inputs x, y and output z, the parties call $[\lambda_x \lambda_y] \leftarrow \mathcal{F}_{\mathsf{CorrectMult}}([\lambda_x], [\lambda_y])$.
- For every wire in the circuit x the parties call $[r\lambda_x] \leftarrow \mathcal{F}_{\mathsf{CorrectMult}}([r], [\lambda_x])$. For every multiplication gate with inputs x and y, the parties call $[r\lambda_x\lambda_y] \leftarrow \mathcal{F}_{\mathsf{CorrectMult}}([r], [\lambda_x\lambda_y])$

ONLINE PHASE

Input Gates. For every input gate x owned by party P_i, the parties do the following:
1. P_i uses the broadcast channel to send $\mu_x = x - \lambda_x$ to all parties.
2. Upon receiving this value, the parties set $\langle x \rangle = ([\lambda_x], \mu_x)$

Only the parties $P_1, \dots, P_{t+1}$ participate in what follows.

Addition Gates. For every addition gate with inputs $\langle x \rangle = ([\lambda_x], \mu_x)$ and $\langle y \rangle = ([\lambda_y], \mu_y)$, the parties locally set $\langle x + y \rangle = ([\lambda_x] + [\lambda_y], \mu_x + \mu_y)$.

Multiplication Gates. For every multiplication gate with inputs $\langle x \rangle = ([\lambda_x], \mu_x)$ and $\langle y \rangle = ([\lambda_y], \mu_y)$, the parties proceed as follows:
1. Call $\mu_z \leftarrow \Pi_{\mathsf{LooseRec}}(\mu_x\mu_y + \mu_x[\lambda_y] + \mu_y[\lambda_x] + [\lambda_x\lambda_y] - [\lambda_z])$
2. Set $\langle x \cdot y \rangle = ([\lambda_z], \mu_z)$.

Checking Phase. Before any output gate is reconstructed, the parties proceed as follows. Let $[\eta_1], \dots, [\eta_M]$ be the shares that were loosely opened to $\eta'_1, \dots, \eta'_M$ during the computation phase (these correspond to $[\mu_z]$ for every wire z that is the output of a multiplication gate). Notice that the parties also have shares $[r \cdot \eta_i]$ for every $i = 1, \dots, M$.
1. For $i = 1, \dots, M$ call $\alpha_i \leftarrow \mathcal{F}_{\mathsf{Coin}}$.
2. Let $[r \cdot \eta] = \sum_{i=1}^{M} \alpha_i [r \cdot \eta_i]$, $\eta' = \sum_{i=1}^{M} \alpha_i \eta'_i$ and $[\beta] = [r \cdot \eta] - \eta' \cdot [r]$.
3. Each P_i commits to their share $\beta^{(i)}$ of β using $\mathcal{F}_{\mathsf{Commit}}$.
4. Finally, parties open and check the commitments, compute $\beta = \mathsf{rec}([\beta])$ and abort if $\beta \neq 0$.

Output Gates. If no honest party aborted above, the parties $P_1, \dots, P_{t+1}$ proceed as follows to open the final output gate x.
1. Parties commit-and-open $[\lambda_x]$ for all output wires x.[a]
2. Let $[\rho] = [r\lambda_x] - \lambda'_x [r]$. The parties commit-and-open their share of $[\rho]$ to the other parties, and abort if $\rho \neq 0$.
3. If the procedure above does not abort, the parties output $\mu_x + \lambda'_x$.

[a] Since parties $P_{t+2}, \dots, P_n$ disconnected, they would not be receiving output. If they were supposed to receive output, they could return to this final phase. The difference with respect to our protocol from Section 3 is that, in this case, these returning parties do not need to participate in the protocol, they only need to receive the shares from the other parties.

Fig. 3. Our second protocol with minimal online complexity. In this protocol, the online phase can be run by only $t + 1$ parties, unlike the protocol from Fig. 2. This protocol requires the underlying algebraic structure to be a large enough field $\mathbb{F}$.

first issue lies in the instantiation of the $\mathcal{F}_{\mathsf{CorrectMult}}$ functionality discussed in Section 2.6. Recall that this functionality is in charge of producing shares of a product, given shares of the inputs, that is, $[x \cdot y]$ from $[x]$ and $[y]$. In Section 2.6 we described a general two-step recipe to instantiate such a functionality: (1) perform the multiplication of $[x]$ and $[y]$ using a protocol for secure multiplication up to additive attacks to get $[x \cdot y + \delta]$, and (2) check that $\delta = 0$ (or, in other words, that the multiplication was done correctly).

In our instantiation over $\mathbb{Z}_{2^k}$ we will assume that the base secret-sharing scheme $[\cdot]$ admits a protocol for multiplication up to additive attacks. (This is the case for practical instantiations like Shamir and replicated secret-sharing. See [ADEN19] for concrete constructions and proofs.) With this in hand, the remaining issue becomes checking the correctness the multiplications. Over fields, we discussed in Section 2.6 that this could be handled by using the techniques over fields from [GS20] which, although described in the context of Shamir secret-sharing, can be easily shown to be extendable to any secret-sharing scheme *over fields*. Extending this check to the $\mathbb{Z}_{2^k}$ setting is not trivial and will be the topic of Section 5.1.

The second issue when porting our first protocol from $\mathbb{F}$ to $\mathbb{Z}_{2^k}$ lies in the check performed at the end that involves taking a random linear combination of the opened values and contrasting it with the corresponding shares. As we showed in Section 3.3, successfully cheating in this check amounts to finding $\delta_1, \dots, \delta_M$, not all zero, and ϵ, such that $\sum_{i=1}^{M} \alpha_i \delta_i = \epsilon$ for some random $\alpha_1, \dots, \alpha_M$. When working over a field $\mathbb{F}$, $\delta_i \neq 0$ translates into δ_i being invertible, which allows us to rewrite the equation above as $\alpha_i = \delta_i^{-1} \cdot \left(\epsilon - \sum_{j \neq i} \alpha_j \delta_j\right)$. As argued, this holds with probability $1/|\mathbb{F}|$ given that $\alpha_i \in \mathbb{F}$ is uniformly random.

Unfortunately, this argument above not hold over $\mathbb{Z}_{2^k}$ as $\delta_i \neq 0$ does not imply that δ_i is invertible (recall that only odd numbers are invertible modulo 2^k). In fact, the equation $\sum_{i=1}^{M} \alpha_i \delta_i = \epsilon$ can be satisfied with high probability $(1/2)$ by choosing $\delta_1 = 2^{k-1}$, $\delta_i = 0$ for $i > 1$ and $\epsilon = 0$, as it only requires α_1 to be even in order to hold (observe that $\delta_1 \alpha_1 = 2^{k-1}(2m) = 2^k m$ which is 0 modulo 2^k, regardless of m). We deal with this issue in Section 5.2.

Finally, our second protocol from Fig. 3 in Section 4, that requires only $t+1$ parties need to stay in the online phase, makes use of MACs in order to enable to computation to take place among $t+1$ parties. To add a MAC on a shared value $[x]$, we let the parties have shares $[r \cdot x]$ and $[r]$ for some global and random r. As we argued in Section 4.2, successfully forging a MAC is equivalent to satisfying an equation similar to the one described above, and the same issues as before, over $\mathbb{Z}_{2^k}$, appear. A MAC scheme for secure computation over $\mathbb{Z}_{2^k}$ was presented in [CDE$^+$18], so for this particular part of our protocol we can directly use the MAC scheme defined in that work, which we refer the readers to for more details.

5.1 Checking Multiplications over $\mathbb{Z}_{2^k}$

Suppose a set of multiplication triples $\{([a_i], [b_i], [c_i])\}_{i=1}^{M}$ have been produced, with $c_i = a_i \cdot b_i + \delta_i$, and where $\delta_i \in \mathbb{Z}_{2^k}$ is an additive error introduced by the adversary. Our task is to check that $\delta_i = 0$ for all $i = 1, \dots, M$. As discussed in Section 2.6, the most efficient method to perform this check *over fields* is presented in [GS20]. However, this check can be roughly seen as a "recursive" refinement of the check from [BFO12], which is the one we will focus on extending to $\mathbb{Z}_{2^k}$ in this section.

Checking Triples over $\mathbb{F}$ First we recall the check from [BFO12] over $\mathbb{F}$. The protocol requires $|\mathbb{F}| \geq 2M - 1$. Also, we assume that one of the triples included in the check is random and will not be used.

1. Find $\lambda_1, \dots, \lambda_{2M-1} \in \mathbb{F}$ such that $\lambda_i - \lambda_j$ is invertible (i.e. non-zero) for any $i \neq j$. Such a sequence exists since $|\mathbb{F}| \geq 2M - 1$.
2. Let $f(X)$ and $g(X)$ be polynomials over $\mathbb{F}$ of degree at most $M - 1$ such that $f(\lambda_i) = a_i$ and $g(\lambda_i) = b_i$ for $i = 1, \dots, M$, and let $a_j := f(\lambda_j)$ and $b_j := g(\lambda_j)$ for $j = M + 1, \dots, 2M - 1$. The parties compute $[a_j]$ and $[b_j]$ for $j = M + 1, \dots, 2M - 1$, which can be done *locally* from $[a_i], [b_i]$ for $i = 1, \dots, M - 1$ using Lagrange coefficients.
3. Use a multiplication protocol that is secure up to additive attacks to compute $[c_j = a_j \cdot b_j + \delta_j]$ for $j = M + 1, \dots, 2M - 1$.
4. The parties call $\mathcal{F}_{\mathsf{Rand}}$ to obtain $[r]$.
5. Let $h(X)$ be the polynomial of degree at most $2M - 2$ such that $h(\lambda_i) = c_i$ for $i = 1, \dots, 2M - 1$. The parties compute $[h(r)]$ locally from $\{[c_i]\}_{i=1}^{2M-1}$ using Lagrange coefficients.
6. The parties compute $[f(r)]$ locally from $\{[a_i]\}_{i=1}^{M}$, and $[g(r)]$ locally from $\{[b_i]\}_{i=1}^{M}$, using Lagrange coefficients.
7. The parties open $f(r)$, $g(r)$ and $h(r)$, and check that $h(r) = f(r)g(r)$.

First we notice that opening the shares in the final step does not undermine privacy given that these shares are a linear combination of the original triples, and there is one of these that is random and is not opened. Now, we argue that the protocol checks correctly that $\delta_i = 0$, for $i = 1, \dots, M$. Indeed, if there is some δ_i that is not zero, then $h(X) \neq f(X)g(X)$ as polynomials and in particular, $h(r) \neq f(r)g(r)$ except with probability $(2M - 2)/|\mathbb{F}|$.

Extending the check to $\mathbb{Z}_{2^k}$ The check described above does not work over $\mathbb{Z}_{2^k}$ for multiple reasons. First, it is not possible to interpolate a polynomial of the required degree. Second, even if this was possible, it does not hold anymore that, for two polynomials that are not equal to each other, the probability that the evaluation at a random point yields the same results is small, which was crucial for our argument above.

We solve these problems by using Galois ring extensions, as in [ACD$^+$19]. Let $\phi(X)$ be a monic polynomial over $\mathbb{Z}_{2^k}$ of degree d, such that its reduction modulo 2 is irreducible over $\mathbb{F}_2$. Consider the quotient ring $R = \mathbb{Z}_{2^k}[X]/(\phi(X))$, which is called a Galois ring of degree d. These rings can be consider to be the analogue of field extensions, and elements in R can be seen as polynomials of degree strictly less than d. We can embed elements of $\mathbb{Z}_{2^k}$ into R by seeing them as polynomials of degree 0. The main property of these rings is the following, which is proven in [ACD$^+$19]

Theorem 1. *If $p^d > N$, then there exist $\lambda_1, \dots, \lambda_N$ such that, for every sequence $(\alpha_1, \dots, \alpha_N) \in R^N$, there exist a unique polynomial $f(X)$ over R of degree at most $N-1$ such that $f(\lambda_i) = \alpha_i$ for all $i = 1, \dots, N$*

In the same work, the following is proved:

Theorem 2 (Lemma 2 in [ACD$^+$19]). *Let $f(X)$ be a polynomial over R of degree $\ell > 0$. Then, the probability that $f(r) = 0$ for a uniformly random $r \in R$, is upper bounded by ℓ/p^d.*

With these two facts at hand, we are ready to extend the protocol for checking multiplication triples from $\mathbb{F}$ to $\mathbb{Z}_{2^k}$. This is done as follows. Let R be a Galois ring extension of degree $d > \log_2(2M-1)$. First, observe that one can extend the secret sharing scheme $[\cdot]$, which is defined over $\mathbb{Z}_{2^k}$, to R by simply secret-sharing each of the coefficients of a given polynomial in R. Also, we can multiply shared elements of R (with security up to additive attacks) by making use of the underlying multiplication over $\mathbb{Z}_{2^k}$. Given this, it is easy to see that the protocol presented over fields also works over $\mathbb{Z}_{2^k}$ if the parties interpret the shared triples as sharings of constant polynomials over R: The necessary sequence $\lambda_1, \dots, \lambda_{2M-1} \in R$ for interpolation exists, which follows from Theorem 1 and the fact that $2^d > 2M-1$, and the final check $h(r) \overset{?}{=} f(r)g(r)$ is sound, from Theorem 2.[11]

5.2 Random Linear Combinations over $\mathbb{Z}_{2^k}$

In this context, a set of shared values $[\eta_1], \dots, [\eta_M]$ have been opened to $\eta'_1, \dots, \eta'_M$, and the parties would like to check that $\eta_i = \eta'_i$ for all $i = 1, \dots, M$. To achieve this over $\mathbb{Z}_{2^k}$ we make use of a Galois ring extension R of degree κ, as follows. Assume for simplicity that κ divides M, say $M = \kappa \cdot \ell$. Intuitively, our method consists of "packing" the elements of $\mathbb{Z}_{2^k}$ into elements of R, and performing the same check we did over $\mathbb{F}$, but over R. This is described in detail below.

[11] Here we would take d such that $(2M-1)/2^d$ is negligibly small. If one wants to keep the weaker requirement $d > \log_2(2M-1)$ only, one can make use of the packing techniques presented in [ACD$^+$19].

Checking correctness of openings over $\mathbb{Z}_{2^k}$

1. For $j = 1, \ldots, \ell$, the parties define the following shared polynomial $\phi_j \in R$ and the following public polynomial $\phi'_j \in R$:
$$[\phi_j] = \left[\eta_{1+(j-1)\cdot\kappa}\right] + X\left[\eta_{2+(j-1)\cdot\kappa}\right] + \cdots + X^{\kappa-1}\left[\eta_{\kappa+(j-1)\cdot\kappa}\right],$$
$$\phi'_j = \eta'_{1+(j-1)\cdot\kappa} + X\cdot\eta'_{2+(j-1)\cdot\kappa} + \cdots + X^{\kappa-1}\cdot\eta'_{\kappa+(j-1)\cdot\kappa}.$$
2. The parties call $\mathcal{F}_{\mathsf{Coin}}$ to sample $\alpha_1, \ldots, \alpha_\ell \in_R R$.
3. The parties compute locally $[\phi] = \sum_{j=1}^{\ell} \alpha_j \cdot [\phi_j]$ and $\phi' = \sum_{j=1}^{\ell} \alpha_j \cdot \phi'_j$.
4. The parties call $z \leftarrow \mathsf{rec}(\phi' - [\phi])$. If $z \neq 0$ then the parties abort.

Security follows a similar argument as the one provided in Section 3.3, mixed with the properties of Galois rings. More precisely, suppose that $\eta'_i = \eta_i + \delta_i$, where the adversary knows $\delta'_i \in \mathbb{Z}_{2^k}$. It is easy to see that this allows us to write, $\phi'_j = \phi_j + \epsilon_j$, for $j = 1, \ldots, \ell$, and where $\epsilon_j \in R$ is known by the adversary. With this notation, one can verify that the check holds if and only if $\sum_{j=1}^{\ell} \alpha_j \cdot \epsilon_j = 0$: Suppose that some δ_{i_0} is not zero. This implies some ϵ_{j_0} not being zero, and so the equality above, once we fix α_j and δ_j for $j \neq j_0$, can be seen as a the evaluation of a polynomial of degree 1 over R producing 0 when evaluated at a random point $\alpha_{j_0} \in R$. From Theorem 2, this can happen with probability at most $1/2^\kappa$.

6 A Unified View of Other Works

So far we have presented two honest-majority protocols with abort, $\Pi_{\mathsf{MPCLowOnline}}$ and $\Pi_{\mathsf{MPCLowChannels}}$, that aim at simplifying and improving the online phase specifically. These protocols are conceptually very simple and general, and their main building blocks, namely the masked secret-sharing scheme from Section 3.1 and the MACs from Section 4, were already present in previous works.

Nevertheless, the main potential of our protocols, specially the one from Section 3, is that it provides a general paradigm for protocols with efficient online phase, and it enables a unified view of the design behind some other protocols in the literature. Here we review some of the protocols that fall within our design. However, we need to describe first replicated secret-sharing for the three-party case.

Replicated Secret Sharing Consider $n = 3$. To secret-share a value $x \in \mathbb{F}$ among 3 parties P_1, P_2, P_3 using replicated secret-sharing, the dealer samples $r_1, r_2, r_3 \in_R \mathbb{F}$ subject to $x = r_1 + r_2 + r_3$, and sends (r_i, r_{i+1}) to P_i, where the sub-indexes wrap modulo 3. The necessary building blocks for this scheme, like multiplication protocols that are secure up to additive attacks, can be found for example in [CGH+18]. Our protocol from Section 3 has a communication complexity in the online phase per multiplication gate of $1.5(3-1) = 3$ elements, which amounts to an average 1 element per party.

6.1 Trident

Trident [CRS20] is a four-party protocol over $\mathbb{Z}_{2^k}$ that tolerates one active corruption. Three types of sharings are defined in [CRS20]:

1. ($[\cdot]$ in [CRS20]) A value $v \in \mathbb{Z}_{2^k}$ is shared among P_1, P_2, P_3 by sending $v_1 \in_R \mathbb{Z}_{2^k}$ to P_1, $v_2 \in_R \mathbb{Z}_{2^k}$ to P_2 and $v_3 = v - v_1 - v_2$ to P_3.
2. ($\langle\cdot\rangle$ in [CRS20]) A value $v \in \mathbb{Z}_{2^k}$ is shared among P_1, P_2, P_3 by sampling $v_1, v_2 \in_R \mathbb{Z}_{2^k}$, setting $v_3 = v - v_1 - v_2$, and sending (v_2, v_3) to P_1, (v_1, v_3) to P_2 and (v_1, v_2) to P_3.
3. ($[\![\cdot]\!]$ in [CRS20]) A value $v \in \mathbb{Z}_{2^k}$ is shared among P_0, P_1, P_2, P_3 if
   – There is a random $\lambda_v \in_R \mathbb{Z}_{2^k}$ shared among P_1, P_2, P_3 according to the scheme from item 2 above.
   – P_0 knows all these shares.
   – P_1, P_2, P_3 all know the value $m_v = v + \lambda_v$.

   sampling $v_1, v_2 \in_R \mathbb{Z}_{2^k}$, setting $v_3 = v - v_1 - v_2$, and sending (v_2, v_3) to P_1, (v_1, v_3) to P_2 and (v_1, v_2) to P_3.

For the purpose of this section we will denote the first two secret-sharing schemes by $[\cdot]_1$ and $[\cdot]_2$, respectively. Note that these are simply additive secret sharing and replicated secret sharing. Our main observation is that the third secret sharing scheme above corresponds (up to some difference in signs) to our secret sharing scheme from Section 3.1 using replicated secret sharing $[\cdot]_1$ among P_1, P_2, P_3 as the base sharing, with the additional property that the "extra party" P_0 knows the replicated shares of the mask.

It is now not hard to verify that the multiplication protocol from [CRS20] (Protocol Π_{Mult} in [CRS20]) corresponds—quite literally—to our protocol from Section 3 without the loose opening optimization (that is, parties P_1, P_2, P_3 open values robustly leveraging the redundancy of replicated secret sharing). The only difference lies in the preprocessing of the mask triples $([\lambda_x]_2, [\lambda_y]_2, [\lambda_x\lambda_y]_2)$: Our protocol from Section 3 uses generic honest-majority correct triple generation methods since we only have three parties, but in [CRS20] the existence of an extra party P_0 is leveraged in order to make the preprocessing more efficient. In a bit more detail, the preprocessing in [CRS20] proceeds as follows:

1. The parties sample shares $[\lambda_x]_2, [\lambda_y]_2$, where P_0 knows all these shares.
2. Parties P_1, P_2, P_3 run the passive multiplication protocol from [AFL⁺16] to multiply these shares. In this protocol P_1 sends one ring element to P_3, P_2 sends one ring element to P_1, and P_3 sends one ring element to P_2. However, active corruption may cause an additive attach in the output, which renders the resulting product incorrect. To make sure that each party sends the correct value, [CRS20] makes use of the fact that P_0 knows all these shares, so P_0 also sends the corresponding messages[12] alongside P_1, P_2 and P_3.

Remark 3. The idea of having extra parties helping in the preprocessing can be generalized to more than 4 parties using Shamir secret-sharing with threshold t.

[12] In fact, hashes of these suffice.

This is done as follows: Consider our protocol from Section 3. The protocol is intended to be run by $2t+1$ parties, which requires some multiplication triples to be preprocessed. As we have seen already, most of the complexities arise from the fact that preprocessing these triples in an honest-majority setting is not simple, given that an active adversary may cause triples to be incorrect, which must be checked. However, if there are t extra parties to assist in this preprocessing (without modifying the adversary threshold, that is, at most t parties out of the new $3t+1$ parties are corrupt), we can apply very efficient protocols to preprocess correct triples, like [DN07,BTH08]. To see why this would be much simpler, observe that the security bottleneck in these protocols is when the parties need to open degree-$2t$ sharings, but with $3t+1$ parties this can be done robustly. With $2t+1$ parties these openings may lead to additive attacks on the output, which requires an extra phase to check the triples.

6.2 ASTRA

This protocol, proposed by Chaudhari et al. [CCPS19], is set in the 3-party setting with active security over the ring $\mathbb{Z}_{2^k}$. The protocol from [CCPS19] achieves a communication complexity of 4 elements per multiplication in the online phase, whereas our protocol instantiated with replicated secret-sharing achieves a communication complexity of 3 elements in the online phase.

Although it is not obvious at first glance due to the presentation in [CCPS19], the ASTRA protocol is closely related to ours. We begin by reviewing their protocol for passive security.

Passive Security The ASTRA protocol defines new types of sharings:

1. ($[\cdot]$ in [CCPS19]) The dealer shares v among P_1, P_2 by sending v_1 to P_1 and $v_2 = v - v_1$ to P_2, where $v_1 \in \mathbb{Z}_{2^k}$ is uniformly random.
2. ($[\![\cdot]\!]$ in [CCPS19]) The dealer shares v among P_0, P_1, P_2 by sampling $\lambda_v, \lambda_{v,1} \in_R \mathbb{Z}_{2^k}$ and sending $(\lambda_{v,1}, \lambda_{v,2} = \lambda_v - \lambda_{v,1})$ to P_0, $(m_v = v + \lambda_v, \lambda_{v,1})$ to P_1 and $(m_v, \lambda_{v,2})$ to P_2.

For the purpose of this write-up, we denote these secret-sharing schemes by $[\cdot]_1$ and $[\cdot]_2$. There are two ways in which the secret-sharing scheme $[\cdot]_2$ can be interpreted. First, it can be seen as parties P_1, P_2 having additive shares $[\cdot]_1$ of a mask $[\lambda_v]_1$, together with the masked value $m_v = v + \lambda_v$, with party P_0 knowing the shares of λ_v. This can be interpreted (up to changes in the sign) as our masked secret-sharing scheme from Section 3.1 applied to additive secret sharing among P_1, P_2, with the additional property that P_0 knows the shares of the masks. This is analogous of what we saw in Trident in Section 6.1. Abusing notation slightly, we denote this as $[v]_2 = ([\lambda_v]_1, v + \lambda_v)$, which resembles the notation from Section 3.1, except that there is an extra party, P_0, who knows the shares of λ_v.

A second interpretation is that, although not observed in [CCPS19], the scheme $[\cdot]_2$ is just a simple variant of replicated secret-sharing in which some of the signs are flipped, which can be seen from the fact that $v = m_v - \lambda_{v,1} - \lambda_{v,2}$, and the fact that each party has two of these summands. This is an interesting observation on its own: Three-party replicated secret-sharing is equivalent to our masked secret sharing from Section 3.1 between two parties, where the third party knows the shares of the mask.

With these interpretations it is easy to see how passive multiplication is handled in [CCPS19] (protocol $\Pi^{\mathsf{s}}_{\mathsf{Mul}}$ in [CCPS19]): Their protocol follows the template of our protocol from Section 3.3, that is, parties P_1, P_2 preprocess $[\lambda_x\lambda_y]_1$ (in fact, P_0 preprocesses these for them) and they reconstruct an appropriate linear combination of the shared values. More precisely, using the notation in [CCPS19], parties P_1, P_2 reconstruct $m_z = m_x m_y - m_x [\lambda_y]_1 - m_y [\lambda_x]_1 + [\lambda_z]_1 + [\lambda_x\lambda_y]_1$, which is, up to changes in the sign, what is done in Section 3.3.

We conclude that, for passive security, ASTRA can be regarded as a simple variant of our protocol from Section 3.3 applied to the 2-party setting where another party knows the shared mask and also helps with the preprocessing.

Active Security The protocol above suffers from two major attack vectors in the active setting: A corrupt P_0 may generate the preprocessing incorrectly, or a corrupt P_1 or P_2 may lie in the reconstruction of $[m_z]_1$. The following expose describes how [CCPS19] handles these issues, and also serves to establish a relation with our methods.

To handle corruption in the multiplication protocol, the authors in [CCPS19] add several steps to the basic semi-honest protocol (this is presented in protocol $\Pi^{\mathsf{m}}_{\mathsf{Mul}}$ in [CCPS19]). We present these steps below, trying to maintain their notation but also establishing their relation with our more general protocol from Section 3.

1. P_1 and P_2 sample common shares $\delta_x, \delta_y, \delta_z$ and define $[\delta_x - \lambda_x]_2 = ([\lambda_x]_1, \delta_x)$ and $[\delta_y - \lambda_y]_2 = ([\lambda_y]_1, \delta_y)$
2. The parties compute $[c]_2 = ([\chi]_1, c+\chi)$ where $c = ab$, $a = \delta_x - \lambda_x$, $b = \delta_y - \lambda_y$ and χ is uniformly random. The parties do this as in our protocol from Section 3.3: P_1 and P_2 first locally compute $[c+\chi]_1 = \delta_x\delta_y - \delta_x [\lambda_y]_1 - \delta_y [\lambda_x]_1 + [\lambda_x\lambda_y]_1 + [\chi]_1$, they send these shares to each other to reconstruct $c+\chi$. They also send the shares of $[\chi]_1$ to P_0 to preserve the invariant. Let $\delta_z = (c+\chi) + \delta_x\delta_y$. Since χ is uniformly random, but the parties P_1 and P_2 ultimately learn $c+\chi$, it is equivalent if we P_1 and P_2 sample δ_z and define $[\chi]_1 := \delta_z - \delta_x\delta_y - [c]_1$. This is what is actually done in [CCPS19]. This has the advantage that they do not need to communicate to obtain $c+\chi$.
3. The parties may have cheated when sending the shares above, but this can be ruled out by verifying that $([a]_2, [b]_2, [c]_2)$ is a correct multiplication triple. This is done by sampling another correct random triple and performing a sacrifice step.
4. In the online phase the parties P_1 and P_2 opened $m_z = x \cdot y + \lambda_z$ as per the passive protocol from the previous section, but due to cheating this value may

be incorrect. P_1 and P_2 send $m_x^\star = m_x + \delta_x$ and $m_y^\star = m_y + \delta_y$ to P_0,[13] which leads to masked shares $\langle x \rangle = ([a]_2, m_x^\star = x + a)$ and $\langle y \rangle = ([b]_2, m_y^\star = y + b)$, reusing the notation from our masked secret-sharing scheme from Section 3.1, with minor changes in the signs.

5. The parties compute locally $[xy + \lambda_z - m_z]_2 = m_x^\star m_y^\star - m_x^\star [b]_2 - m_y^\star [a]_2 + [c]_2 + [\lambda_z]_2 - m_z$, and they check that this shared value is equal to 0 as follows: Let α_1, α_2 be the shares held by P_0, and let β, α_i be the shares held by P_i for $i = 1, 2$. These are supposed to be shares of 0, so they should satisfy $\beta = \alpha_1 + \alpha_2$. P_0 sends a hash of $\alpha_1 + \alpha_2$, and P_1 and P_2 check that the hash of β is the same as the one received from P_0.[14] The parties abort if this is not the case.

We see then that, to compute an actively secure multiplication in ASTRA, the parties begin by performing the passive multiplication. However, it must be checked that the value $m_z = xy + \lambda_z$ was opened correctly. To do this, the parties first obtain replicated shares $[xy]_2$ and then compute $[\alpha]_2 = [xy]_2 + [\lambda_z]_2 - m_z$. The goal now is to check that $\alpha = 0$, but this can be done very efficiently since replicated secret sharing is a particular instance of honest-majority secret sharing, which has enough redundancy.

The major complexity in [CCPS19] lies in computing $[xy]_2$. This is done by first computing masked shares $\langle x \rangle = ([a]_2, m_x^\star)$ and $\langle y \rangle = ([b]_2, m_y^\star)$ and using what can be regarded as our multiplication protocol from Section 3.3 (which as we stressed was already present in [BNO19]) in order to compute $[xy]_2$ efficiently. This requires to preprocess a correct random triple $([a], [b], [c])$, which, as we showed above, is produced in [CCPS19] by cleverly exploiting the correlations that the parties already have from $[\lambda_x]_1, [\lambda_y]_1, [\lambda_x \lambda_y]_1$.

Comparison to our protocol. Now that we have described ASTRA's multiplication protocol in a setting that is as close to ours as possible, we can easily compare our work with theirs.

Both of these works make use of the masked secret sharing scheme from Section 3.1 in some form: ASTRA uses it on top of additive secret sharing among P_1 and P_2, using P_0 as a helper, in a similar way as done in Trident (see Section 6.1). Ours uses it on top of replicated secret sharing among the three parties.

To ensure active security, both protocols must ensure that the opening of certain shared value on the base secret-sharing scheme is done correctly. In our work the base secret-sharing scheme is replicated secret-sharing, so we could simply make use of the redundancy of the scheme, but instead we do this in a more efficient way by open non-robustly using only two parties and verifying correctness at the end of the protocol. In ASTRA, however, the base secret-sharing scheme is additive secret sharing, which is not robust by default. To overcome

[13] Actually, only P_1 needs to send the actual values. The other party P_2 can send hashes of these and P_0 verifies that these match.

[14] The checks for all multiplication gates can be concatenated so that only one hash is sent at the end of the protocol.

this issue, like in our protocol, replicated shares of the value to be opened are computed, but instead of opening these directly (which would undermine the whole idea of using additive secret sharing), the parties exploit the fact that the (potentially incorrect) value was already opened, and that checking equality to 0 with replicated secret sharing is much cheaper than directly opening the value.

We see then that both protocols use similar building blocks and have a similar design: Open cheaply but non-robustly and then check at the final stage that these openings were correct, using communication which is independent of the amount of openings being checked. Both protocols have a communication pattern in which only two parties need to be online most of the time. However, our protocol uses a smaller amount of communication overall.

6.3 BLAZE

After unfolding the ASTRA protocol in the previous section, it is very easy to understand the BLAZE protocol from [PS20], as these are closely related. To keep the presentation here as concise as possible we will not introduce any additional notation.

In the ASTRA protocol as sketched above, P_1 and P_2 must send $m_x^\star = m_x + \delta_x$ and $m_y^\star = m_y + \delta_y$ in step 4 to P_1 so that the masked shares $\langle x \rangle = ([a]_2, m_x^\star = x + a)$ and $\langle y \rangle = ([b]_2, m_y^\star = y + b)$ can be obtained. The main observation in [PS20] is that the values $m_x^\star$ and $m_y^\star$ (denoted by $\beta_x^\star$ and $\beta_y^\star$ in [PS20]) could have been already sent to P_0 in the gates having x and y as output wires. Hence, when processing the multiplication gate with inputs x and y, $m_z^\star$ is sent to P_0. This lowers the communication of the consistency check by half since instead of sending two values to P_0, namely $m_x^\star$ and $m_y^\star$, only one value $m_z^\star$ must be sent.

7 Security Proofs

We consider a standard ideal functionality for secure function evaluation of a function f. This functionality, denoted $\mathcal{F}_{\mathsf{SFE}}^f$, is shown below.

$\mathcal{F}_{\mathsf{SFE}}^f$

Let n be the number of parties, $f : R^n \to R$ the function to be computed, and $\mathcal{S}$ the ideal world adversary.
Init: Initialize a set I as $\emptyset$ as well as $z = \bot$.
Input: On $(\mathtt{input}, P_i, x_i)$ from P_i and if $(i, \cdot) \notin I$, set $I \leftarrow I \cup \{(i, x_i)\}$.
Eval: On $(\mathtt{eval})$ from $\mathcal{S}$ and if $(i, \cdot) \in I$ for all $i = 1, \ldots, n$; Compute and set $z = f(x_i, \ldots, x_n)$ and give z to $\mathcal{S}$.
Output: On $(\mathtt{output}, a)$ from $\mathcal{S}$ and if $z \neq \bot$. If $a = 1$, output $\bot$ to all parties and halt. Otherwise output z to all parties and halt.

For simplicity this functionality only considers functions of arity n and with 1 output. Extending $\mathcal{F}_{\mathsf{SFE}}^f$ to handle functions of different arity or with more than one output is a straightforward extension and is therefore left out for the sake of simplifying the below proofs.

7.1 Security of $\Pi_{\mathsf{MPCLowOnline}}$

Theorem 3. *Protocol $\Pi_{\mathsf{MPCLowOnline}}$ securely realizes $\mathcal{F}^f_{\mathsf{SFE}}$ for any $f : \mathbb{F}^n \to \mathbb{F}$ against a static malicious adversary corrupting $t = (n-1)/2$ parties in a $(\mathcal{F}_{\mathsf{Rand}}, \mathcal{F}_{\mathsf{CorrectMult}}, \mathcal{F}_{\mathsf{Coin}})$-hybrid model, unless with probability $1/|\mathbb{F}|$.*

Proof. Let $\mathcal{A}$ be the real world adversary and $\mathcal{Z}$ the environment. Let C denote the set of corrupted parties and H its complement. The ideal adversary $\mathcal{S}$ has control over the $t = (n-1)/2$ parties in C that $\mathcal{Z}$ corrupts in the ideal world and emulates the remaining $t+1 = (n-1)/2 + 1$ honest parties. The simulator $\mathcal{S}$ in addition emulates all calls to $\mathcal{F}_{\mathsf{Rand}}$, $\mathcal{F}_{\mathsf{CorrectMult}}$ and $\mathcal{F}_{\mathsf{Coin}}$.

If at any point during the simulated run of $\Pi_{\mathsf{MPCLowOnline}}$ an abort happens, the simulator invokes $(\texttt{input}, P_i, 0)$ on $\mathcal{F}^f_{\mathsf{SFE}}$ for all $P_i \in C$, followed by $(\texttt{eval})$ and finally $(\texttt{output}, 1)$, after which $\mathcal{S}$ halts. When we say that the simulator aborts, it is implied that it first takes these actions before stopping.

Moving on, the simulator performs the following actions when simulating the execution of $\Pi_{\mathsf{MPCLowOnline}}$:

1. **Offline phase.**
   - For every wire on the circuit, emulate the calls to $\mathcal{F}_{\mathsf{Rand}}$ by sampling λ_x and secret-sharing these.
   - For every input gate x corresponding to a corrupt party emulate the reconstruction of $[\lambda_x]$ to this party.
   - For multiplication gates with wires u and v, the simulator receives shares $[\lambda_v]^C$ and $[\lambda_u]^C$ from the corrupt parties. Then, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{CorrectMult}}$ by checking if these are the same shares than the ones $\mathcal{S}$ sent above. If not, then $\mathcal{S}$ aborts. Else, $\mathcal{S}$ computes $\lambda_w = \lambda_v \lambda_u$, and distributes $[\lambda_w]$.
2. **Input phase.**
   - For a wire v owned by an honest party, the simulator samples a uniformly random μ_x and broadcasts this value to all parties.
   - For a wire v owned by a corrupt party P_i, the simulator waits to receive μ_v after which it computes $x_i = \mu_v + \lambda_v$. Finally, the simulator invokes $(\texttt{input}, P_i, x_i)$ on $\mathcal{F}^f_{\mathsf{SFE}}$.
3. **Computation phase.**
   - When all parties have given their input (and in particular, when $\mathcal{S}$ have extracted all the inputs of corrupt parties), the simulator invokes $(\texttt{eval})$ on $\mathcal{F}^f_{\mathsf{SFE}}$. Let z denote the output.
   - The simulator now runs the computation phase as in $\Pi_{\mathsf{MPCLowOnline}}$: Addition gates require no interaction, and for multiplication gates, $\mathcal{S}$ runs Π_{LooseRec} as in the real execution.
4. **Checking phase.** Let $[\eta_1], \dots, [\eta_M]$ be shares of values that were loosely opened to $\eta'_1, \dots, \eta'_M$ during the computation phase.
   - $\mathcal{S}$ distributes α_i for $i = 1, \dots, M$ to all parties, as part of the emulation of $\mathcal{F}_{\mathsf{Coin}}$.

- Compute $[\eta] = \sum_{i=1}^{M} \alpha_i [\eta_i]$, $\eta' = \sum_{i=1}^{M} \alpha_i \eta_i'$ and reconstruct $w = \Pi_{\mathsf{Rec}} [\eta' - \eta]$ towards all corrupt parties. If reconstruction fails, or if $w \neq 0$, abort.

5. **Output phase.** Let v be the output wire and $([\lambda_v], \nu_v)$ the associated mask.
   - Using knowledge of $[\lambda_v]^C$, the simulator computes a sharing $[\lambda_{v'}]$ such that (1) $[\lambda_{v'}]^C = [\lambda_v]^C$ and (2) $\lambda_{v'} = z - \nu_v$. (Recall that z was the output of the computation.) $\lambda_{v'}$ is then reconstructed towards the corrupt parties.
   - Finally, $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs and halts.

We now argue that $\mathcal{Z}$ cannot distinguish between $\mathtt{EXEC}_{\mathcal{Z},\mathcal{A},\Pi_{\mathsf{MPCLowOnline}}}^{\mathcal{F}_{\mathsf{Rand}},\mathcal{F}_{\mathsf{CorrectMult}},\mathcal{F}_{\mathsf{Coin}}}$ and $\mathtt{EXEC}_{\mathcal{Z},\mathcal{S},\mathcal{F}_{\mathsf{SFE}}^{f}}$. As a first observation, notice that up until the checking phase, the simulator is perfectly simulating the execution of $\Pi_{\mathsf{MPCLowOnline}}$ and so no advantage can be gained. The only avenue for distinguishing is if the adversary can cheat in the checking phase. Indeed, at step 4 (checking phase) in the simulation, the ideal functionality $\mathcal{F}_{\mathsf{SFE}}^{f}$ have already performed the computation of f, and so, if the adversary can cheat in this step, the resulting output of the hybrid execution would be potentially different from the one of the ideal execution. However, cheating in step 4 is not possible unless with negligible probability, as we will now argue:

Observe that passing the check in step (4) corresponds to satisfying the equation $0 = \sum_{i=1}^{M} \alpha_i(\eta_i - \eta_i') = \sum_{i=1}^{M} \alpha_i \delta_i$, where $\eta_i' = \eta_i + \delta_i$ and δ_i is an error introduced by $\mathcal{A}$ when η_i was loosely reconstructed. Cheating implies that some δ_i are non-zero, and thus that the above equation can be rewritten as $\alpha_i \delta_i = -\sum_{j \neq i} \alpha_j \delta_j$. However, this implies that $\alpha_i = -\delta_i^{-1} \sum_{j \neq i} \alpha_j \delta_j$ which clearly cannot be the case except with negligible probability as the α_i's were chosen uniformly at random from $\mathbb{F}$. In particular, this holds with probability $1/|\mathbb{F}|$. □

7.2 Security of $\Pi_{\mathsf{MPCLowChannels}}$

For the following security proof, we will make use of the following ideal commitment functionality

$\mathcal{F}_{\mathsf{Commit}}$

Commit. On $(\mathtt{commit}, P_i, P_j, x)$ from P_i, record (P_i, P_j, x) if no such tuple was recorded before.
Open. On $(\mathtt{open}, P_i, P_j)$ from P_i and if a tuple (P_i, P_j, x) was previously recorded, send x to P_j.

Theorem 4. *Protocol $\Pi_{\mathsf{MPCLowChannels}}$ securely realizes $\mathcal{F}_{\mathsf{SFE}}^{f}$ for any $f : \mathbb{F}^n \to \mathbb{F}$ against a static malicious adversary corrupting $t = (n-1)/2$ parties in a $(\mathcal{F}_{\mathsf{Rand}}, \mathcal{F}_{\mathsf{CorrectMult}}, \mathcal{F}_{\mathsf{Coin}}, \mathcal{F}_{\mathsf{Commit}})$-hybrid model, unless with probability $1/|\mathbb{F}|$.*

Proof. Let $\mathcal{A}$ be the real world adversary and $\mathcal{Z}$ the environment. Let C denote the set of corrupted parties and H its complement. The ideal adversary $\mathcal{S}$ runs $\mathcal{A}$ internally, giving it control over the $t = (n-1)/2$ parties in C that $\mathcal{Z}$ corrupts and simulates the remaining $t+1 = (n-1)/2+1$ parties. The simulator $\mathcal{S}$ in addition emulates all calls to $\mathcal{F}_{\mathsf{Rand}}$, $\mathcal{F}_{\mathsf{CorrectMult}}$, $\mathcal{F}_{\mathsf{Coin}}$ and $\mathcal{F}_{\mathsf{Commit}}$. For the sake of simplicity, assume parties $P_1, \ldots, P_t$ are corrupt. Similar to above, an abort of $\mathcal{S}$ implies that $\mathcal{F}^f_{\mathsf{SFE}}$ is invoked with 0 inputs, then evaluated and finally aborted before $\mathcal{S}$ itself halts.

Simulation proceeds much as in the previous proof, and as follows:

1. **Offline phase.**
   – As in the protocol from the previous section, the simulator emulates $\mathcal{F}_{\mathsf{Rand}}$ and $\mathcal{F}_{\mathsf{CorrectMult}}$ as in the real execution. Also, for an input wire x owned by P_i, private reconstruct a random value λ_x to P_i, and abort if this this fails for an honest receiver.
2. **Input phase.**
   – For wires owned by an honest party P_i, the simulator broadcasts a random value μ_x to all parties.
   – For wires owned by a corrupt party P_i, the simulator waits to receive μ_x from P_i (aborting if the broadcast is inconsistent) after which $\mathcal{S}$ invokes $(\texttt{input}, P_i, \mu_x + \lambda_x)$ on $\mathcal{F}^f_{\mathsf{SFE}}$.
3. **Computation phase.** At this point, only parties $P_1, \ldots, P_{t+1}$ are participating in the protocol, and all broadcasts happen with this subset in mind.
   – The simulator invokes $(\texttt{eval})$ on $\mathcal{F}^f_{\mathsf{SFE}}$. Let z be the output of the computation.
   – Run $\Pi_{\mathsf{MPCLowChannels}}$ where $\mathcal{S}$ plays the role of the single remaining honest party P_{t+1}. When handling multiplication gates, P_{t+1} uses a random share for the loose opening.
4. **Checking phase.** Let $[\eta_1], \ldots, [\eta_M]$ be shares of values that were loosely opened to $\eta'_1, \ldots, \eta'_M$ during the previous step. Recall that each party also holds $[r \cdot \eta_i]$ for $i = 1, \ldots, M$. $\mathcal{S}$ proceeds as follows.
   – Emulate $\mathcal{F}_{\mathsf{Coin}}$ by selecting uniformly at random values α_i for $i = 1, \ldots, M$, and send these to all parties.
   – Compute $[r \cdot \eta] = \sum_{i=1}^{M} \alpha_i [r \cdot \eta_i]$, $\eta' = \sum_{i=1}^{M} \alpha_i \eta'_i$ and $[\beta] = [r \cdot \eta] - \eta' [r]$. Wait for all parties to commit to their share of β.
   – When all parties have committed as well as opened their shares, compute $\beta' = \mathsf{rec}([\beta])$ and abort if $\beta' \neq 0$.
5. **Output phase.**
   – For the output wire z, the simulator waits for all parties to commit to their share of λ_z.
   – The simulator changes the shares of $[\lambda_z]$ and $[r \cdot \lambda_z]$ corresponding to P_{t+1} so that they reconstruct to $[\lambda'_z]$ and $[r \cdot \lambda'_z]$ respectively, where $\lambda'_z = z - \mu_z$.
   – $\mathcal{S}$ participates in the opening of λ'_z, emulating $\mathcal{F}_{\mathsf{Commit}}$. The simulator waits for all parties to be committed to their share of λ'_z, and then their openings are broadcast. The resulting opened value is equal to λ''_z, which may be different to λ'_z.

– Compute $[\rho] = [r \cdot \lambda'_z] - \lambda''_z [r]$ and wait for all parties to commit to their share of ρ.
– When all parties are committed to their share of ρ, their openings are broadcast. If the reconstructed value is not 0, the parties abort.
– Finally, and if no aborts happened in the previous steps, the simulator invokes $(\texttt{output}, 0)$ on $\mathcal{F}^f_{\mathsf{SFE}}$ and then halts.

We now argue indistinguishability between $\texttt{EXEC}^{\mathcal{F}_{\mathsf{Rand}},\mathcal{F}_{\mathsf{CorrectMult}},\mathcal{F}_{\mathsf{Coin}},\mathcal{F}_{\mathsf{Commit}}}_{\mathcal{Z},\mathcal{A},\Pi_{\mathsf{MPCLowChannels}}}$ and $\texttt{EXEC}_{\mathcal{Z},\mathcal{S},\mathcal{F}^f_{\mathsf{SFE}}}$.

As in the proof for $\Pi_{\mathsf{MPCLowOnline}}$, simulation proceeds perfectly up until the checking phase, and we have to argue that errors introduced during computation will be caught in the output phase. Towards this goal, suppose that errors were introduced, i.e., $\eta'_i = \eta_i + \delta_i$ with $\delta_{i_0} \neq 0$ for some i_0. Let $\beta + \epsilon$ be the value that was opened. Successfully cheating implies that $0 = \beta + \epsilon$ and thus that:

$$\epsilon = -\beta = -r \sum_{i=1}^{M} \alpha_i(\eta_i - \eta'_i) = -r \sum_{i=1}^{M} \alpha_i \delta_i \implies r = -\epsilon \left(\sum_{i=1}^{M} \alpha_i \delta_i \right)^{-1},$$

assuming that $\sum_{i=1}^{M} \alpha_i \delta_i$ is not zero. This is the case indeed with overwhelming probability given that this is the inner product between a non-zero vector $(\delta_1, \ldots, \delta_M)$ and a random vector. Furthermore, the equation above cannot be satisfied with high probability since r was picked uniformly at random and not revealed to $\mathcal{A}$. Note that this argument relies on ϵ being independent of β (indeed, if the adversary could pick ϵ dependent on β, then $0 = \beta + \epsilon$ would be trivial to satisfy). This is achieved thanks to the $\mathcal{F}_{\mathsf{Commit}}$ functionality.

This takes care of the checking phase, which leaves us with the output phase. First notice that the equivocation of the honest party's share guarantees that $\mu_z + \lambda'_z$ results in the same output as the ideal execution. Furthermore, this is acceptable since the adversary only controls t shares, which carry no information about the underlying secrets.

Now, suppose that $\lambda''_z = \lambda'_z + \delta$, where $\delta \neq 0$. Furthermore, the adversary can add an error in the opening of ρ, resulting in $\rho + \gamma$. If the final check passes, then $\rho + \gamma = 0$, which means that $r \cdot \lambda'_z - r \cdot (\lambda'_z + \delta) + \gamma = 0$, or $r = \gamma/\delta$, which as before cannot happen with high probability since r is chosen at random and independently of γ and δ. This concludes the proof.

8 Conclusion

We presented two conceptually simple, yet novel, actively secure protocols that both perform very well in terms of amortized communication complexity. Both protocols achieve an asymptotic complexity of *only* 1.5 shares per party, which for a popular instantiation like Shamir secret-sharing translates to only 1.5 field elements per multiplication per party. In addition, our second protocol $\Pi_{\mathsf{MPCLowChannels}}$ shows how techniques used in $\Pi_{\mathsf{MPCLowOnline}}$ together with techniques from dishonest majority MPC can be used to obtain a protocol where

$n - t$ parties do not even have to be online in during the computation phase of the protocol. Finally, we show how the framework that we establish along the way, can be used to frame several recent and highly efficient protocols in a clear and unified manner.

References

ACD$^+$19. Mark Abspoel, Ronald Cramer, Ivan Damgård, Daniel Escudero, and Chen Yuan. Efficient information-theoretic secure multiparty computation over $\mathbb{Z}/p^k\mathbb{Z}$ via galois rings. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019: 17th Theory of Cryptography Conference, Part I*, volume 11891 of *Lecture Notes in Computer Science*, pages 471–501, Nuremberg, Germany, December 1–5, 2019. Springer, Heidelberg, Germany.

ADEN19. Mark Abspoel, Anders Dalskov, Daniel Escudero, and Ariel Nof. An efficient passive-to-active compiler for honest-majority MPC over rings. Cryptology ePrint Archive, Report 2019/1298, 2019. `https://eprint.iacr.org/2019/1298`.

AEV21. Mark Abspoel, Daniel Escudero, and Nikolaj Volgushev. Secure training of decision trees with continuous attributes. To appear at PoPETs 2021, 2021.

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: 23rd Conference on Computer and Communications Security*, pages 805–817, Vienna, Austria, October 24–28, 2016. ACM Press.

BCS19. Carsten Baum, Daniele Cozzo, and Nigel P. Smart. Using TopGear in overdrive: A more efficient ZKPoK for SPDZ. In Kenneth G. Paterson and Douglas Stebila, editors, *SAC 2019: 26th Annual International Workshop on Selected Areas in Cryptography*, volume 11959 of *Lecture Notes in Computer Science*, pages 274–302, Waterloo, ON, Canada, August 12–16, 2019. Springer, Heidelberg, Germany.

Bea92. Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *Advances in Cryptology – CRYPTO'91*, volume 576 of *Lecture Notes in Computer Science*, pages 420–432, Santa Barbara, CA, USA, August 11–15, 1992. Springer, Heidelberg, Germany.

BFO12. Eli Ben-Sasson, Serge Fehr, and Rafail Ostrovsky. Near-linear unconditionally-secure multiparty computation with a dishonest minority. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 663–680, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Heidelberg, Germany.

BGIN19. Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. Practical fully secure three-party computation via sublinear distributed zero-knowledge proofs. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019: 26th Conference on Computer and Communications Security*, pages 869–886. ACM Press, November 11–15, 2019.

BHKL18. Assi Barak, Martin Hirt, Lior Koskas, and Yehuda Lindell. An end-to-end system for large scale P2P MPC-as-a-service and low-bandwidth MPC for weak participants. In David Lie, Mohammad Mannan, Michael Backes, and

XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 695–712, Toronto, ON, Canada, October 15–19, 2018. ACM Press.

BNO19. Aner Ben-Efraim, Michael Nielsen, and Eran Omri. Turbospeedz: Double your online SPDZ! Improving SPDZ using function dependent preprocessing. In Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung, editors, *ACNS 19: 17th International Conference on Applied Cryptography and Network Security*, volume 11464 of *Lecture Notes in Computer Science*, pages 530–549, Bogota, Colombia, June 5–7, 2019. Springer, Heidelberg, Germany.

BTH06. Zuzana Beerliová-Trubíniová and Martin Hirt. Efficient multi-party computation with dispute control. In Shai Halevi and Tal Rabin, editors, *TCC 2006: 3rd Theory of Cryptography Conference*, volume 3876 of *Lecture Notes in Computer Science*, pages 305–328, New York, NY, USA, March 4–7, 2006. Springer, Heidelberg, Germany.

BTH08. Zuzana Beerliová-Trubíniová and Martin Hirt. Perfectly-secure MPC with linear communication complexity. In Ran Canetti, editor, *TCC 2008: 5th Theory of Cryptography Conference*, volume 4948 of *Lecture Notes in Computer Science*, pages 213–230, San Francisco, CA, USA, March 19–21, 2008. Springer, Heidelberg, Germany.

Can01. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science*, pages 136–145, Las Vegas, NV, USA, October 14–17, 2001. IEEE Computer Society Press.

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*, pages 81–92, 2019.

CDE$^+$18. Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. SPD $\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for dishonest majority. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part II*, volume 10992 of *Lecture Notes in Computer Science*, pages 769–798, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany.

CGH$^+$18. Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority MPC for malicious adversaries. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part III*, volume 10993 of *Lecture Notes in Computer Science*, pages 34–64, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany.

CRS20. Harsh Chaudhari, Rahul Rachuri, and Ajith Suresh. Trident: Efficient 4pc framework for privacy preserving machine learning. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.

DEF$^+$19. Ivan Damgård, Daniel Escudero, Tore Kasper Frederiksen, Marcel Keller, Peter Scholl, and Nikolaj Volgushev. New primitives for actively-secure MPC over rings with applications to private machine learning. In *2019 IEEE Symposium on Security and Privacy*, pages 1102–1120, San Francisco, CA, USA, May 19–23, 2019. IEEE Computer Society Press.

DEK20. Anders Dalskov, Daniel Escudero, and Marcel Keller. Secure evaluation of quantized neural networks. PoPETs, 2020.

DKL$^+$13. Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS 2013: 18th European Symposium on Research in Computer Security*, volume 8134 of *Lecture Notes in Computer Science*, pages 1–18, Egham, UK, September 9–13, 2013. Springer, Heidelberg, Germany.

DN07. Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In Alfred Menezes, editor, *Advances in Cryptology – CRYPTO 2007*, volume 4622 of *Lecture Notes in Computer Science*, pages 572–590, Santa Barbara, CA, USA, August 19–23, 2007. Springer, Heidelberg, Germany.

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: 1st Conference on Information-Theoretic Cryptography*, pages 5:1–5:24, Boston, MA, USA, June 17–19, 2020. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

FLNW17. Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017, Part II*, volume 10211 of *Lecture Notes in Computer Science*, pages 225–255, Paris, France, April 30 – May 4, 2017. Springer, Heidelberg, Germany.

GIP$^+$14. Daniel Genkin, Yuval Ishai, Manoj Prabhakaran, Amit Sahai, and Eran Tromer. Circuits resilient to additive attacks with applications to secure computation. In David B. Shmoys, editor, *46th Annual ACM Symposium on Theory of Computing*, pages 495–504, New York, NY, USA, May 31 – June 3, 2014. ACM Press.

Gol01. Oded Goldreich. *Foundations of Cryptography: Basic Tools*, volume 1. Cambridge University Press, Cambridge, UK, 2001.

GS20. Vipul Goyal and Yifan Song. Malicious security comes free in honest-majority mpc. Cryptology ePrint Archive, Report 2020/134, 2020. `https://eprint.iacr.org/2020/134`.

KKW18. Jonathan Katz, Vladimir Kolesnikov, and Xiao Wang. Improved non-interactive zero knowledge with applications to post-quantum signatures. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 525–537, Toronto, ON, Canada, October 15–19, 2018. ACM Press.

KOS16. Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, pages 830–842, Vienna, Austria, October 24–28, 2016. ACM Press.

KPR18. Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ great again. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018, Part III*, volume 10822 of *Lecture Notes* *in Computer Science*, pages 158–189, Tel Aviv, Israel, April 29 – May 3, 2018. Springer, Heidelberg, Germany.

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: 24th Conference on Computer and Communications Security*, pages 259–276, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press.

LN18. Yehuda Lindell and Ariel Nof. Fast secure multiparty ECDSA with practical distributed key generation and applications to cryptocurrency custody. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 1837–1854, Toronto, ON, Canada, October 15–19, 2018. ACM Press.

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: 25th Conference on Computer and Communications Security*, pages 35–52, Toronto, ON, Canada, October 15–19, 2018. ACM Press.

NV18. Peter Sebastian Nordholt and Meilof Veeningen. Minimising communication in honest-majority MPC by batchwise multiplication verification. In Bart Preneel and Frederik Vercauteren, editors, *ACNS 18: 16th International Conference on Applied Cryptography and Network Security*, volume 10892 of *Lecture Notes in Computer Science*, pages 321–339, Leuven, Belgium, July 2–4, 2018. Springer, Heidelberg, Germany.

PS20. Arpita Patra and Ajith Suresh. BLAZE: blazing fast privacy-preserving machine learning. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*, 2020.

WGC19. Sameer Wagh, Divya Gupta, and Nishanth Chandran. SecureNN: 3-party secure computation for neural network training. *Proceedings on Privacy Enhancing Technologies*, 2019(3):26–49, July 2019.