Minimising Communication in Honest-Majority MPC by Batchwise Multiplication Verification

Peter Sebastian Nordholt[1] and Meilof Veeningen[2]

[1] Alexandra Institute
peter.s.nordholt@alexandra.dk
[2] Philips Research (formerly)
meilof@gmail.com

Abstract. In this paper, we present two new and very communication-efficient protocols for maliciously secure multi-party computation over fields in the honest-majority setting with abort. Our first protocol improves a recent protocol by Lindell and Nof. Using the so far overlooked tool of batchwise multiplication verification, we speed up their technique for checking correctness of multiplications (with some other improvements), reducing communication by $2\times$ to $7\times$. In particular, in the 3PC setting, each party sends only two field elements per multiplication. We also show how to achieve fairness, which Lindell and Nof left as an open problem. Our second protocol again applies batchwise multiplication verification, this time to perform 3PC by letting two parties perform the SPDZ protocol using triples generated by a third party and verified batchwise. In this protocol, each party sends only $\frac{4}{3}$ field elements during the online phase and $\frac{5}{3}$ field elements during the preprocessing phase.

Full version (with appendix) of paper published at ACNS 2018.

1 Introduction

Multi-party computation (MPC) allows a number of parties to compute a function on their respective sensitive inputs without leaking anything but the computation result. Recently, there has been a lot of interest in concretely efficient actively secure MPC in the honest-majority setting with abort, in which fewer than $n/2$ out of n parties may be corrupted. In this setting, very efficient solutions are known and it is also possible to achieve *fairness*, i.e., either all parties learn the result or none do, which is not possible without a honest majority.

A number of recent works have achieved particularly striking performance numbers. Binary circuits can be evaluated at a cost of sending 10 bits per AND gate for three parties due to [FLNW17], and arithmetic circuits can be evaluated at a cost of sending 4 (for $n = 3$), $5(n-1)$, or 42 field elements per multiplication due to [LN17]. However, this still leaves at least a factor four communication increase compared to passive security. Moreover, these best known protocols unfortunately do not satisfy fairness (unlike other honest-majority protocols).

In this work, we improve on the state-of-the-art of concretely efficient honest-majority MPC by further decreasing communication complexity, while also supporting fairness. Concerning communication complexity, we decrease communication in the three main variants of the protocol of Lindel and Nof by factors of approximately 2, 5, and 7, respectively. In all cases, the gap between passive and active security becomes only a factor 2. Moreover, in the three-party setting, the best protocol now requires sending just two messages per party per multiplication. Some of this improvement comes from better use of PRNGs; a more significant improvement comes from applying the tool of batchwise multiplication verification [BFO12], a technique that allows to check that many multiplications have been performed correctly by essentially checking a single multiplication.

We additional provide a novel three-party protocol, based on the SPDZ protocol [DPSZ12], that reduces *online* communication from 2 in our protocol described above to $\frac{4}{3}$ messages per party per multiplication. This comes at the expense of requiring a *preprocessing* phase with $\frac{5}{3}$ messages per party per multiplication. Our SPDZ-based protocol also makes heavy use of PRNGs and batchwise multiplication verification, but additionally incorporates the idea of taking a two-party protocol in the preprocessing model, and replacing the distributed preprocessing protocol by in-the-plain preprocessing by a third party. This idea was known before but, as far as we know, has never been applied; we extend this idea by allowing the preprocessing to be spread evenly between the three parties. By way of comparison, in the two-party dishonest majority setting, a recent SPDZ variant by Keller *et. al* [KPR17] requires the equivalent of around 130 field elements to be sent per party, highlighting the communication gap between the honest- and dishonest-majority settings.

In both our Lindell-Nof and our SPDZ based protocol, the decrease in communication implies an increase in computation, but we show that in many practical settings, communication is still the bottleneck.

Finally, we show how to add fairness both of our constructions. We employ general principles to achieve fairness such as using signature-based broadcast for agreement and MACs or signatures to prevent output manipulation. Our solutions are especially crafted to ensure that they add as little practical overhead as possible; in particular, they do not affect the above communication complexity results. This means that communication-efficient, actively secure MPC is possible in practice without having to sacrifice fairness.

1.1 Outline

We discuss preliminaries in Sections 2, before presenting our Lindell-Nof-based and SPDZ-based constructions in Sections 3 and 4, respectively. We give a brief performance analysis in Section 5.

1.2 Related Work

Several recent works are closely related to this paper. Concerning efficient honest-majority MPC, the most relevant work is the framework for communication-

efficient MPC from [LN17] that forms the basis of our first protocol. It is also the closest competitor in terms of overall communication complexity that we are aware of. Another recent honest-majority MPC framework is due to [DOS17]. Although their construction is quite a bit less communication-efficient than ours, it does work for arbitrary rings as opposed to just fields. They also provide a (less efficient) construction for fairness largely based on the same principles as ours.

Concerning the technique of batchwise multiplication verification, the groundwork was laid out in several earlier works. Ben-Sasson *et al.* [BFO12] first proposed batchwise multiplication verification. As discussed below, there it was used to get an asymptotic result; we are not aware of works using it to improve practical performance. Works such as the Pinocchio verifiable computation system [PHGR13] and the Trinocchio protocol that combines it with MPC [SVdV16] were a main inspiration to start seeing batchwise multiplication verification also as a tool that may deliver practical efficiency. Corrigan-Gibbs and Boneh [CB17] first proposed to use batchwise multiplication verification where one party provides data and a number of other parties verify it, as in our SPDZ-based protocol; but there it is not for performing the MPC but for checking its inputs.

2 Preliminaries

In this section, we present our notation and the security model for honest-majority MPC with abort, and the main technique we will use to minimise its communication: batchwise multiplication verification.

2.1 Notation and Security Model

The protocols in this paper are for n parties $\mathcal{P} = \{\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n\}$, where an adversary may statically corrupt a minority of up to t parties, i.e., $2t < n$. We generally work in the field $\mathbb{Z}_p$ for some prime $p > 2^\sigma$, where σ is a statistical security parameter. We use $[x]$ to denote a Shamir secret sharing of x; $[\![x]\!]$ to denote an additive sharing; and $\langle x \rangle = ([\![x]\!], [\![\alpha x]\!])$ to denote a SPDZ sharing consisting of an additive sharing of the value and its MAC. $[x]_i$, $[\![x]\!]_i$, $\langle x \rangle_i$ refer to shares held by party $\mathcal{P}_i$. We heavily use pseudorandom number generators (PRNGs) to sample random data. For a pseudorandom number generator prng we use the notation $r \leftarrow \mathsf{prng}$ to indicate sampling r uniformly at random (from the relevant domain). $[a, b]$ denotes the interval $[a, a+1, \ldots, b]$.

We define security in the traditional standalone security model from [Can00] as adapted in [LN17]. Security in this model is captured by demanding indistinguishability of the real-world protocol execution to an ideal-world execution with a trusted third party. In the real-world model, the protocol is run between honest parties in the presence of a non-uniform probabilistic polynomial time adversary $\mathcal{A}$ that acts on behalf of the corrupted parties. We assume a synchronous network with pairwise private channels and a rushing adversary (that receives its messages in each round before it sends them). A party may *abort*, meaning

it sends a special abort message to all parties, who abort in the next round. An execution of a protocol π in this model with inputs $x_1, \ldots, x_n$, adversarial auxiliary input z and security parameter κ is denoted $\text{Real}_{\pi,\mathcal{A}(z),\mathcal{C}}(x_1, \ldots, x_n, \kappa)$. This is a tuple containing the outputs of the honest parties and an arbitrary output chosen by the adversary.

The ideal-world model defines how an idealised protocol execution looks like in which the computation is performed by an incorruptible trusted party executing a certain *functionality*. The functionality defines the exact security guarantees; we will define variants with and without fairness. In the ideal-world model, the trusted party executes the functionality in the presence of the honest parties and a non-uniform probabilistic polynomial time adversary $\mathcal{S}$.

The functionalities for fair and non-fair MPC both start with each party $\mathcal{P}_i$ sending its input x_i to the trusted party. The adversary may choose an arbitrary input for corrupted parties and may also provide $\perp$ to indicate an abort. The trusted party computes output y as specified by f, or sets $y = \perp$ if the adversary supplied $\perp$. In the *fair variant*, the trusted party sends the outputs to all of the parties. In the *non-fair variant*, the trusted party sends y to the adversary who returns $c \in \{\top, \perp\}^n$. For each party $\mathcal{P}_i$, if $c_i = \top$ the trusted party sends y to $\mathcal{P}_i$, otherwise it sends $\perp$. Ideal-world executions with these functionalities are denoted $\text{Ideal}_{f,\mathcal{S}(z),\mathcal{C}}(x_1, \ldots, x_n, \kappa)$ or $\text{Ideal}^{\text{fair}}_{f,\ldots}(\ldots)$: a tuple containing the outputs of the honest parties and an arbitrary output chosen by the adversary.

Security is defined as indistinguishability between real-world and ideal-world executions. Precisely, we say that a protocol π *securely computes f with statistical security parameter σ for honest majority* if, for every adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that, for all $x_i, z, \mathcal{C}$ with $|\mathcal{C}| \leq t$, the distinguishing probability between $\text{Ideal}_{f,\mathcal{S}(z),\mathcal{C}}(x_1, \ldots, x_n, \kappa)$ and $\text{Real}_{\pi,\mathcal{A}(z),\mathcal{C}}(x_1, \ldots, x_n, \kappa)$ is at most $2^{-\sigma} + \mu(\kappa)$ for some μ negligible in κ. Protocol π *fairly computes f with statistical security parameter σ for honest majority* if the same holds with respect to $\text{Ideal}^{\text{fair}}_{f,\ldots}(\ldots)$. Security can also be defined more generally for any functionality $\mathcal{F}$. As is well-known, we can design protocols containing calls to an ideal functionality $\mathcal{F}$ (in the so-called *$\mathcal{F}$-hybrid model*) and then replace the ideal functionality by a secure protocol implementing it [Can00].

The above model describes standalone executions with synchronous communication, but we believe that neither limitation is inherent to our protocol. In asynchronous models, unlike above, there is no global round clock. In general, synchronous protocols can be made asynchronous by having each party confirm to all other parties that it has received all messages for round t, and only proceeding to send messages for round $t + 1$ after receiving all confirmations [KLR06], but this is of course costly. We expect that such confirmations are only necessary at a few points in our protocol. In composable models, unlike the standalone model above, protocols are proven secure also in the presence of simultaneous other protocols and protocol instances. Here, we note that we use only black-box non-rewinding simulators, so adding "start synchronisation" should be enough to achieve composability [KLR06]. We leave details for future work.

2.2 Batchwise Multiplication Verification

Batchwise multiplication verification was introduced in [BFO12] to improve the asymptotic complexity of verifying preprocessed multiplication triples over small fields. Standard multiplications checks, e.g. based on sacrificing, scale with the security parameter (which is larger than the field size), but using batchwise multiplication verification, these costs can be spread over a batch.

In particular, given secret-shared values $[a_1], \ldots, [a_N]$, $[b_1], \ldots, [b_N]$, $[c_1], \ldots, [c_N]$, the goal is to verify that $c_i = a_i \cdot b_i$ for all i. This is done by translating these N equalities of field elements into a single equality of polynomials, and verifying this equality based on the Schwartz-Zippel lemma [Zip79,Sch80]. Fix nonzero $\omega_1, \ldots, \omega_{2N-1}$, and let $A(x), B(x)$ be of degree $\leq N-1$ such that for $i \in [1, N]$, $A(\omega_i) = a_i$ and $B(\omega_i) = b_i$. If we let $C(x) = A(x)B(x)$, then obviously $C(\omega_i) = c_i$ for $i \in [1, N]$, but the converse is also true: if there exists a polynomial $C(x)$ of degree $\leq 2N-1$ such that $C(x) = A(x)B(x)$ and $C(\omega_i) = c_i$ for $i \in [1, N]$, this implies $c_i = a_i \cdot b_i$.

In batchwise multiplication verification, first, $C(x)$ is constructed by computing $C(\omega_j) = A(\omega_j) \cdot B(\omega_j)$, $j \in [N+1, 2N-1]$ using passively secure MPC and deriving its coefficients by interpolation. Then, A, B, and C are evaluated in a random point $s \notin \{\omega_1, \ldots, \omega_{2N-1}\}$. This can be done with local linear operations given shares of the a_i, b_i, c_i, and $C(\omega_j)$. Finally, a multiplication check protocol is run to check that $A(s) \cdot B(s) = C(s)$. The Schwartz-Zippel lemma, states that for a non-zero degree d polynomial, P, over field $\mathcal{F}$ of and a random $r \in S$ for a finite $S \subseteq \mathcal{F}$ the probability that $P(r) = 0$ is at most $d/|S|$. Thus if $A(s) \cdot B(s) = C(s)$ then with high probability, $A(x) \cdot B(x) = C(x)$ as polynomials and hence $a_i \cdot b_i = c_i$. Note that for each triple, an additional passively secure multiplication is needed, but the multiplication check is performed only once per batch, giving the asymptotic advantage.

In [CB17], the above idea is used in a different setting: some party provides inputs to MPC, and we want to verify that inputs satisfy a certain property. This property is phrased in terms of a number of multiplications of linear combinations of inputs, and the multiplications are checked similarly to above. In this case, the inputter determines and provides the "witness" values $C(\omega_j)$ proving that the multiplications are correct, and the computing parties again use a simple protocol to check that $A(s) \cdot B(s) = C(s)$. It is also shown there that the various polynomial computations can be performed efficiently using FFTs.

3 Lindell-Nof with Fewer Messages and More Fairness

In this section, we show how to reduce the communication complexity of the Lindell-Nof protocol for honest-majority MPC [LN17] and how to add fairness. We outline their construction (Section 3.1); plug in batchwise multiplication verification (Section 3.2); analyse and further reduce communication complexity (Section 3.3); finally, we show how to achieve fairness and discuss two other improvements (Section 3.4).

3.1 The Lindell-Nof Construction

Lindell and Nof present a framework for efficient actively secure MPC with a honest majority [LN17]. The basic observation underlying this framework is that many passively secure MPC protocols are "actively secret", essentially meaning that an active attack can break correctness of the computation, but not privacy. Hence, to perform a computation in an actively secure way, one can simply perform the computation using a passively secure protocol and, prior to opening the result, retrospectively check that all multiplications, as these are the only operations that require interaction, have been performed correctly.

In slightly more detail, the Lindell-Nof construction uses of t-out-of-n secret sharing, such as Shamir secret sharing or replicated secret sharing. The protocol starts with all parties secret-sharing their inputs, and checking whether they are "correct", in the sense that the shares of all honest parties reconstruct to a unique value. Next, a passively secure MPC is executed, with linear operations performed locally on shares and multiplication using known protocols for Shamir by Gennaro *et al.* [GRR98], Damgård and Nielsen [DN07] and for replicated secret sharing by Araki *et. al* [AFL+16]. We will refer to these three multiplication methods as GRR, DN and AFL+ respectively. Finally, the correctness of the performed multiplications is checked using one of two possible methods, and if this check passes, the secret shares of the output are reconstructed to obtain the output. Overall, this gives active security without fairness with relatively little communication.

3.2 Plugging in Batchwise Multiplication Verification

We now show how batchwise multiplication verification can be used to efficiently implement the multiplication check in the Lindell-Nof protocol. As discussed above, the multiplication check is called at the end of the protocol to check correctness of a number of passively secure multiplications performed before.

Our protocol performing this multiplication check is shown in Fig. 1. The protocol uses functionalities $\mathcal{F}_{\text{RAND}}$ for generating $\mathsf{share}\, r$ for random $r \in \mathbb{Z}_p$ and $\mathcal{F}_{\text{COIN}}$ for generating a public field element $r \in \mathbb{Z}_p \setminus \{0\}$ known to all parties as described in [LN17]. Moreover, it uses a passively secure multiplication protocol that, as described by Lindell and Nof [LN17], needs to be "secure up to additive attacks", meaning that the adversary can manipulate its result only by adding an additive offset to its result. The GRR, DN and AFL+ protocols mentioned above all meet this requirement.

Our multiplication protocol follows the basic idea of [BFO12], but avoids its actively secure $A(s) \cdot B(s) = C(s)$ check. We add a random multiplication triple (a_N, b_N, c_N) to the batch of triples and choose s uniformly at random from $\mathbb{Z}_p$. Then, the values of $A(s), B(s), C(s)$ are uniformly random and can be opened so that the check $A(s) \cdot B(s) = C(s)$ can be performed in the plain. (Note that this option was not available to the authors of [BFO12] since they need s from an extension field so $A(s), B(s), C(s)$ are not uniform.)

Protocol: Batchwise multiplication check for Lindell-Nof (batch size N):

Inputs: The parties hold a list of triples $([a_i], [b_i], [c_i])_{i=1}^{N-1}$ they want to verify.

1. Generate random $[a_N], [b_N]$ with $\mathcal{F}_{\text{RAND}}$ and together compute $[c_N] \leftarrow [a_N] \cdot [b_N]$
2. Let $A(x), B(x)$ be of degree $\leq N-1$ such that $A(\omega_i) = a_i$; $B(\omega_i) = b_i$ for $i \in [1, N]$. Using $[a_i]$ and $[b_i]$, locally compute $[a_j] = [A(\omega_j)]$, $[b_j] = [B(\omega_j)]$ for $j \in [N+1, 2N-1]$
3. Together compute $[c_j] \leftarrow [a_j] \cdot [b_j]$ for $j \in [N+1, 2N-1]$
4. Generate random s with $\mathcal{F}_{\text{COIN}}$. Repeat until $s \notin \{0, \omega_1, \ldots, \omega_{2N-1}\}$.
5. Let $C(x)$ be of degree $\leq 2N-2$ such that $C(\omega_i) = c_i$ for $i \in [1, 2N-1]$. Locally compute $[A(s)]$, $[B(s)]$ and $[C(s)]$ as linear combinations of $([a_i])_{i=1}^{N}$, $([b_i])_{i=1}^{N}$ and $([c_i])_{i=1}^{2N-1}$ respectively
6. Exchange secret shares $[A(s)]$, $[B(s)]$ and $[C(s)]$ between all parties. Output `accept` if the shares are correct and $A(s)B(s) = C(s)$.

Fig. 1. Batchwise multiplication check for Lindell-Nof

We now prove correctness of our multiplication check. In Lindell-Nof, correctness of their multiplication check is shown in [LN17, Lemma 3.9]. We prove that the same result holds for our multiplication check, implying that it can be used as a drop-in replacement in their protocol. Actually, our result is slightly more complete since we do not just prove correctness but also privacy of the multiplication check. In the appendix, we use this result for a self-contained proof of an optimised version of the Lindell-Nof protocol.

Proposition 1. *Suppose shares $([a_i], [b_i])_{i=1}^{N-1}$ are correct and $([c_i])_{i=1}^{N-1}$ are valid, and that $[\cdot] \leftarrow [\cdot]\cdot[\cdot]$ is a multiplication protocol secure up to additive attack. There exists a simulator that, on input $\Delta_i := c_i - (a_i \cdot b_i)$ and the shares held by the corrupted parties, simulates an execution of the protocol from Fig. 1 with respect to an active adversary corrupting a minority of parties with statistical distance at most negligibly greater than $(2N-2)/(|\mathbb{Z}_p| - 2N)$. In particular, if any $\Delta_k \neq 0$, then the honest parties output `accept` with at most this probability; if all $\Delta_k = 0$ then honest parties fail or succeed at the will of the adversary.*

Proof. The simulator proceeds as follows. The simulator first simulates the generation of random $[a_N]$ and $[b_N]$ and the computation of $[c_N]$, $[a_{N+1}], \ldots, [a_{2N-1}]$, $[b_{N+1}], \ldots, [b_{2N-1}]$, $[c_{N+1}], \ldots, [c_{2N-1}]$, learning the errors $\Delta_N, \ldots, \Delta_{2N-1}$ to the c_i introduced by the adversary (which is possible since the protocol is secure up to additive attack). Simulate the generation of s and the computation of $[A(s)]$, $[B(s)]$, and $[C(s)]$. Let $D(x)$ be of degree $\leq 2N-2$ such that $D(\omega_1) = \Delta_1, \ldots, D(\omega_{2N-1}) = \Delta_{2N-1}$. If $(\Delta_1, \ldots, \Delta_{2N-1}) \neq \mathbf{0}$ but $D(s) = 0$, abort. Generates random $A(s)$ and $B(s)$, and let $C'(s) = A(s) \cdot B(s)$ and $C(s) = C'(s) + D(s)$. Simulate the opening of $[A(s)]$ to $A(s)$, $[B(s)]$ to $B(s)$, and $[C(s)]$ to $C(s)$. Let the honest parties output `success` if $D(s) = 0$ and the adversary provides the correct shares of $[A(s)]$, $[B(s)]$, $[C(s)]$ and fail otherwise.

We argue that this simulation is indeed indistinguishable. For this, we need to check that the view of the adversary and the outputs of the honest parties

in the simulation are indistinguishable from a real execution. Concerning the view of the adversary, note that the values $A(s)$ and $B(s)$ that are opened are uniformly random because of the inclusion of the random $[a_N]$, $[b_N]$. Given these values $A(s)$ and $B(s)$, $C'(s) = A(s) \cdot B(s)$ is the value that is opened for $[C(s)]$ if all multiplications are correct. By linearity of the computation of $C(s)$, given $A(s)$ and $B(s)$ the value the adversary expects for $[C(s)]$ is $C'(s) + D(s)$. Hence, the simulation of the multiplication check is indistinguishable to the adversary and its success implies $(\Delta_1, \ldots, \Delta_{c_1}) = \mathbf{0}$, unless $(\Delta_1, \ldots, \Delta_{2N-1}) \neq \mathbf{0}$ and $D(s) = 0$. But $D(s)$ is the evaluation of a polynomial of degree at most $2N - 2$ in a random point from $\mathbb{Z}_p \setminus \{0, \omega_1, \ldots \omega_{2N-1}\}$, so by the Schwartz-Zippel lemma, if $(\Delta_1, \ldots, \Delta_{c_1}) \neq \mathbf{0}$ then $D(s) = 0$ with probability $(2N-2)/(|\mathbb{Z}_p|-2N)$. Hence, except with this probability, the adversary cannot make wrong multiplications pass the check, so also the honest parties' outputs are indistinguishable. □

Corollary 1 (Informal). *The protocol for computing an arithmetic circuit over a finite field from [LN17] with the batchwise multiplication check from Fig. 1 computes any n-party functionality f with computational security in the presence of a malicious adversary controlling up to $t < n/2$ corrupted parties.*

In Appendix A, we present an optimised and slightly simplified version of the Lindell-Nof protocol and prove its security in detail.

3.3 Performance Analysis and Optimisation with PRNGs

Table 1 shows how the amount of communication in the Lindell-Nof protocol is reduced by batchwise multiplication verification, and how it can be further reduced with PRNGs. As mentioned above Lindel and Nof give three concrete instantiations of their protocol based on the GRR, DN and AFL+ multiplication protocols respectively [GRR98,DN07,AFL$^+$16]. (The exact variants of the protocols used for this comparison are given in Appendix A.) They instantiate three core operations, multiplying, opening shared values and generating a random shared value, and use them in two multiplication checks. The first check uses 2 multiplications, 2 random values and 3 openings; the second check uses 6 multiplications and 3 random values. In GRR, the first check is used; in DN, the second check is used; and in AFL+, a slight optimisation of the first check is used, leading to the given performance in Table 1.

As shown, using batchwise multiplication verification, checking a multiplication requires essentially one additional multiplication. As a result, using it instead of either of the Lindell-Nof multiplication checks reduces communication by a factor 2 to 3.5. The constant cost of the check (hidden behind the $\gtrsim$ symbol in the table) is spread over the triples in a batch but pretty small: e.g., for ≤ 10 parties the batch size needed to make the overhead less than one is always less than 50 and to make it less than 0.1 it is less than 500. As shown in Section 5, this is possible without affecting computational complexity too much.

Using PRNGs, we can reduce communication in the GRR and DN constructions even further. For instance, consider the re-sharing of values that takes place

Operation	GRR	GRR-PRNG	DN	DN-PRNG	AFL+
Random value	0	0	$\lesssim 2$	$\lesssim 1$	0
Opening	$n-1$	$n-1$	$n-1$	$n-1$	1
Passive mult.	$n-1$	$n-t-1$	$\lesssim 6$	$\lesssim 3$	1
LN mul + check	$5(n-1)$	$6(n-t-1)$	$\lesssim 42$	$\lesssim 18$	4
Batch mul + check	$\gtrsim 2(n-1)$	$\gtrsim 2(n-t-1)$	$\gtrsim 12$	$\gtrsim 6$	$\gtrsim 2$

Table 1. Field elements sent per party for the Lindell-Nof protocol instantiated with GRR, DN (both with or without PRNG optimizations) and AFL+ (with PRNG optimization). The number of parties and the threshold is denoted by n and t respectively (generally $n \approx 2t$). Grey areas are our results

in GRR multiplication: instead of sending shares to each party, the dealing party can simply set the shares of t parties by pairwise PRNGs between him and the recipients so that he only needs to send $n-t-1$ shares, halving communication if $n = 2t+1$. This idea is of course not new, but it is still important for us since applying it reduces communication in the Shamir constructions by an additional factor of at least two. In particular, using PRNGs, the Shamir-based construction with GRR becomes as communication-efficient as the PRNG-based construction. Details appear in Appendix A.

3.4 Further Improvements

Adding Fairness To achieve fairness, we first let the parties reach agreement on whether to produce an output. Once there is agreement, we let the parties derive the output in such a way that the adversary cannot force a failure anymore.

To reach agreement, we use detectable broadcast [FGMvR02]. Detectable broadcast lets a party send a message to all parties so that either all parties receive the same message, or all parties agree that the broadcast has failed. In our case, the adversary may cause this failure after seeing the value to be broadcast. Unlike full broadcast, it can be achieved over private channels without set-up assumptions. Essentially, [FGMvR02] achieves detectable broadcast by letting each party once pick and distribute a public key, and performing a pairwise check if all parties consistently sent out their keys. After this setup, broadcasts are performed with the standard Dolev-Strong protocol [DS83]. In our protocol, parties detectably broadcast their shares of $A(s)$, $B(s)$, and $C(s)$ in the last round of the multiplication check; the parties decide to produce an output only if all parties have successfully broadcast a value; all shares consistently reconstruct to some values $A(s)$, $B(s)$, and $C(s)$; and $A(s) \cdot B(s) = C(s)$.

To derive the output, we need to ensure that honest parties can detect wrong values sent by corrupted parties. If there are only few parties, each party $\mathcal{P}_i$ can input a random information-theoretic MAC key α_i, β_i into the MPC (with PRSS, this requires no communication) and the parties compute MAC $\alpha_i \cdot x + \beta_i$ on output x. After the multiplication check, all parties send their shares of x and $\alpha_i x + \beta_i$ to $\mathcal{P}_i$, who selects whichever reconstructed x has a correct MAC. For

many parties, this technique is not secure since it costs $\log((t+1)\binom{n-1}{t}) \approx n$ bits security; for that case see Appendix A.3.

Efficient inner products One particularly appealing property of MPC based on secret sharing schemes like Shamir and replicated secret sharing, is that they allow inner products $[c] = \sum_{i=1}^{l} [a_i] \cdot [b_i]$ to be computed at the cost of a single multiplication. Such multiplication protocols first locally perform the multiplication (turning t-out-of-n shared inputs into a $2t$-out-of-n sharing of the product) and then re-share the result (turning the product from a $2t$-out-of-n sharing back into a t-out-of-n sharing). To compute an inner product, several local multiplications are first summed up and then the result is re-shared.

We can make such inner product computations actively secure by generalising batchwise multiplication verification to verify many inner products of the same length. Instead of generating two random values and computing their product, we generate $2l$ random values and compute their inner product. We then define polynomials $(A_i(x))_{i=1}^{l}$, $(B_i(x))_{i=1}^{l}$, $C(x)$ in the natural way; exchange shares of $(A_i(s))_{i=1}^{l}$, $(B_i(s))_{i=1}^{l}$, $C(s)$; and check whether $\sum_{i=1}^{l} A_i(s)B_i(s) = C(s)$. This gives the same security guarantees as batchwise multiplication verification.

Smaller fields Because of the false positive rate of the Schwartz-Zippel lemma, our construction requires a field of size at least $2N \cdot 2^{\sigma}$, where σ is the statistical security parameter. When working over a smaller field, the multiplication check can be performed repeatedly. This way, statistical security can be boosted arbitrarily: repeating the check k times increases statistical security from $\log((|\mathbb{Z}_p| - 2N)/(2N-2)$ to $\log(\binom{|\mathbb{Z}_p|-2N}{k}/\binom{2N-2}{k})$ bits. Note that repeated checking can be done more efficiently than by just repeating the full check as follows. Instead of adding one random triple to a batch of multiplications, we add k of them; and instead of generating one random challenge s, we generate k challenges s_i and evaluate $A(s_i)$, $B(s_i)$, and $C(s_i)$ for $i = 1, \ldots, k$. (These can be opened because of the inclusion of the k random triples.)

4 SPDZ with an Untrusted Dealer

In this section, we present a protocol for honest-majority 3PC. The main contribution is a communication efficient protocol implementing the preprocessing phase for the 2PC SPDZ protocol using batchwise multiplication verification to check the correctness of Beaver triples generated locally by a third party dealer $\mathcal{P}_3$. In the online phase two parties $\mathcal{P}_1, \mathcal{P}_2$ use the preprocessed values in the regular two party SPDZ[3] to compute the desired function. Using a small addition to the online SPDZ protocol, based on ideas from [JNO14], we can allow the dealer to provide input to and receive output from the 2PC protocol, thus giving an actively secure 3PC protocol in the honest-majority setting. We leave these modifications as an exercise.

[3] The version by Damgård *et. al* refered to as SPDZ-2 [DKL+13]

We note that, the resulting 3PC protocol is highly asymmetric; in the preprocessing phase the $\mathcal{P}_3$ is doing most of the work while in the online phase $\mathcal{P}_1, \mathcal{P}_2$ do all the work. To better utilise resources across all three parties, we also develop a load balanced version of the protocol. This works by letting each of party play the role of the dealer in separate runs of the preprocessing phase. In the online phase, we then partition the multiplications to be performed into three sets to be evaluated by each pair of parties in a 2PC fashion. The overall communication per multiplication required in both versions is 5 field elements for the preprocessing phase and 4 field elements in the online phase (as per the regular 2PC SPDZ protocol). Thus using the load balanced version of the protocol we get 4/3 and 5/3 fields elements an average per party in the preprocessing and online phases respectively. We defer the load balancing version of the protocol to the appendix, and in this section we focus on our protocol for the SPDZ preprocessing phase.

We note that, compared to our Lindell-Nof based protocol, the protocol presented in this section does communicate three additional field elements per multiplication. However, the *online* phase communicates two field elements less than the Lindell-Nof based protocol. Thus the setting were preprocessing is available our SPDZ-based protocol is preferable.

4.1 Data Needed for the Online Phase

Before we describe our protocol for the preprocessing phase we here first summarise the data that should be generated: We use $\langle a \rangle = (\llbracket a \rrbracket, \llbracket \alpha a \rrbracket)$ to denote a *SPDZ sharing* of $a \in \mathbb{Z}_p$, where the sharing is between the parties $\mathcal{P}_1, \mathcal{P}_2$. Here $\alpha \in \mathbb{Z}_p$ is a random *MAC key* fixed at initialisation and unknown to both $\mathcal{P}_1, \mathcal{P}_2$, but which they share additively. The shared value αa of is an *information theoretic MAC* on a, which is used in the online phase to ensure active security.

The online phase of SPDZ needs preprocessed *multiplication triples* and *input masks*. A multiplication triple is SPDZ sharings $(\langle a \rangle, \langle b \rangle, \langle c \rangle)$ where $a, b \in \mathbb{Z}_p$ are random values and $c = ab$. In the online phase each multiplication will consume one triple. An input mask is a pair $(r, \langle r \rangle)$ for a random value $r \in \mathbb{Z}_p$ known to, say, $\mathcal{P}_1$. In the online phase each input provided by $\mathcal{P}_1$ consumes one such mask. For security in the online phase we require that the preprocessed data should be *correct* in the sense that the shared values and their MACs should obey the correllations described above. Furthermore, the shared values should be unknown and random in the view of any corrupt party participating in the online phase (i.e., either $\mathcal{P}_1$ or $\mathcal{P}_2$). We more formally describe the ideal functionality in $\mathcal{F}_{\text{DEAL}}$ in Section B.1

4.2 Preprocessing Phase

The basic idea of our protocol is to let $\mathcal{P}_3$ generate the all the preprocessed data locally, and send the appropriate shares to $\mathcal{P}_1, \mathcal{P}_2$. Batchwise multiplication verification is then used to check that $\mathcal{P}_3$ generated the multiplication triples correctly, and a separate check is used to check that the MACs are correct. To

save communication our protocol heavily relies on joint PRNGs $\mathsf{prng}_{i,j}$ between each pair of parties $\mathcal{P}_i, \mathcal{P}_j$ in order to non-interactively share values.

Our protocol Π_{DEAL} implementing the preprocessing phase is described in detail in Fig. 2 and Fig. 3. In Fig. 2 we show how the protocol is initialised by using the joint PRNGs to sample a random MAC key α in such a way that α is unknown to all parties but is additively secret shared between each pair of parties $\mathcal{P}_i, \mathcal{P}_j$, denoted $[\![\alpha]\!]_i^{i,j}$, $[\![\alpha]\!]_j^{i,j}$. Additionally, $\mathcal{P}_1$ and $\mathcal{P}_2$ use $\mathsf{prng}_{1,2}$ to sample a challenge $s_{1,2}$ used for multiplication checks.

In Fig. 2 we also describe two subprotocols which will be used through out the Π_{DEAL} protocol. These protocols use the PRNGs to non-interactively generate a random additive sharing $[\![r]\!]$ between $\mathcal{P}_1, \mathcal{P}_2$, where r is known to $\mathcal{P}_3$ (4a of Fig. 2), and given any such shared r an additive sharing of $[\![\alpha r]\!]$ between *all* the parties (4b of Fig. 2). Note, that this means that by sending $\mathcal{P}_3$'s share $[\![\alpha r]\!]_3$ of αr to, say, $\mathcal{P}_1$ we can trivially compute a SPDZ sharing $\langle r \rangle$ by adding $[\![\alpha r]\!]_3$ to $[\![\alpha r]\!]_1$. In the protocol we slightly abuse notation in this case by saying that $\mathcal{P}_1$ *updates* her share $[\![\alpha r]\!]_1 = [\![\alpha r]\!]_1 + [\![\alpha r]\!]_3$. Note that this requires $\mathcal{P}_3$ so send exactly one field element per SPDZ sharing.

In Fig. 3 we describe how to generate and verify the actually preprocessed data to be used in the online phase. Multiplication triples are generated by first using the 4a and 4b subprotocols to generate $\langle a \rangle$ and $\langle b \rangle$ as described above. $\mathcal{P}_3$ then computes $c = ab$ and additively shares it among the parties, using 4b on c we get its MAC. This requires $\mathcal{P}_3$ to send four field elements.

For a batch of triples $(\langle a_i \rangle, \langle b_i \rangle, \langle c_i \rangle)_{i=1}^{N-1}$ the multiplicative property $a_i b_i = c_i$ is verified using batch multiplication verification similar to the Lindell-Nof case above. In this case we let the dealer $\mathcal{P}_3$ compute and additively share (without MACs) the values $c_{N+1} = C(\omega_{N+1}), \ldots, c_{2N-1} = C(\omega_{2N-1})$, as in [CB17]. $\mathcal{P}_1, \mathcal{P}_2$ verify the multiplications by checking the polynomials evaluated in the challenge point s generated at initialisation. Again we can open $A(s), B(s), C(s)$ by sacrificing one triple. The check requires a single field element sent per triple and an additional element per batch of $N-1$ triples. Overall, a total of 5 field elements are sent to generate each multiplication triples and verify the multiplicative property plus one additional field element per batch.

Input masks are simply generated by first using the 4a and 4b subprotocols to generate $\langle r \rangle$, and then letting $\mathcal{P}_3$ send the value r to the party using the input mask. This requires sending two field elements for each input mask.

Finally, $\mathcal{P}_1, \mathcal{P}_2$ must check that all the MACs resulting from invocations of the 4b subprotocol are correct. We do this using protocol similar to the MAC check subprotocol of the regular SPDZ protocol. Essentially, the parties take a pseudorandom linear combination of all the shared values generated, and check that the MACs a consistent with the result. This takes constant communication.

The intuition for security of the protocol goes as follows. Consider first a corrupt $\mathcal{P}_i$ for $i \in \{1, 2\}$, i.e., one of the parties that will run the online phase. In this case, the dealer $\mathcal{P}_3$ is honest, and only deals correct random additive shares, which does not reveal information on the shared values. Furthermore, since $\mathcal{P}_i$ only sends messages in the protocols checking correctness of the dealt shares, $\mathcal{P}_i$

Protocol Π_{DEAL}

Inputs: The amount of multiplication triples M, random input masks I_1, I_2 and a batch size N.

1. *(PRNG setup)* Each pair of parties $\mathcal{P}_i, \mathcal{P}_j$ sets up joint PRNG $\mathsf{prng}_{i,j}$ (one party generates it and sends it to the other)
2. *(MAC key generation)* The parties generate a random secret MAC key α, additively shared between each pair of parties:
   (a) Let $\alpha_1, \alpha_2 \leftarrow \mathsf{prng}_{1,3}$; $\alpha_3, \alpha_4 \leftarrow \mathsf{prng}_{1,2}$; $\alpha_5, \alpha_6 \leftarrow \mathsf{prng}_{2,3}$
   (b) Parties $\mathcal{P}_1, \mathcal{P}_2$ set $[\![\alpha]\!]_1^{1,2} = \alpha_1 + \alpha_2 + \alpha_3$, $[\![\alpha]\!]_2^{1,2} = \alpha_4 + \alpha_5 + \alpha_6$
   (c) Parties $\mathcal{P}_1, \mathcal{P}_3$ set $[\![\alpha]\!]_1^{1,3} = \alpha_3 + \alpha_4 + \alpha_1$, $[\![\alpha]\!]_3^{1,3} = \alpha_5 + \alpha_6 + \alpha_2$
   (d) Parties $\mathcal{P}_2, \mathcal{P}_3$ set $[\![\alpha]\!]_2^{2,3} = \alpha_3 + \alpha_4 + \alpha_5$, $[\![\alpha]\!]_3^{2,3} = \alpha_1 + \alpha_2 + \alpha_6$
3. *(Sample Challenge)* $\mathcal{P}_1, \mathcal{P}_2$ sample $s_{1,2} \in \mathbb{Z}_p \setminus \{0, \omega_1, \ldots, \omega_{2N-1}\}$ using $\mathsf{prng}_{1,2}$
4. *(Subprotocols)* Throughout the parties use two subprotocols to non-interactively generate random value r known by $\mathcal{P}_3$ and secret-shared between $\mathcal{P}_1, \mathcal{P}_2$ and a corresponding MAC secret shared among all three parties:
   (a) *(Random)* Let $[\![r]\!]_1 \leftarrow \mathsf{prng}_{1,3}$; $[\![r]\!]_2 \leftarrow \mathsf{prng}_{2,3}$. $\mathcal{P}_3$ sets $r = [\![r]\!]_1 + [\![r]\!]_2$.
   (b) *(Additive MAC shares)* Let $\delta_{1,3} \leftarrow \mathsf{prng}_{1,3}$; $\delta_{2,3} \leftarrow \mathsf{prng}_{2,3}$; $\delta_{1,2} \leftarrow \mathsf{prng}_{1,2}$. For an additively shared $[\![r]\!]$ as above
   $\mathcal{P}_1$ sets $[\![\alpha r]\!]_1 = [\![\alpha]\!]_1^{1,3} \cdot [\![r]\!]_1 + \delta_{1,2} - \delta_{1,3}$.
   $\mathcal{P}_2$ sets $[\![\alpha r]\!]_2 = [\![\alpha]\!]_2^{2,3} \cdot [\![r]\!]_2 + \delta_{2,3} - \delta_{1,2}$.
   $\mathcal{P}_3$ sets $[\![\alpha r]\!]_3 = [\![\alpha]\!]_3^{1,3} \cdot [\![r]\!]_1 + [\![\alpha]\!]_3^{2,3} \cdot [\![r]\!]_2 + \delta_{1,3} - \delta_{2,3}$.

(continued in Fig. 3)

Fig. 2. Protocol Π_{DEAL}

can only influence the protocol by making it abort (which we allow anyway), but cannot influence the values of any of the shared values. Thus the preprocessed data will be correct and $\mathcal{P}_i$ will not get information on the shared values. Consider then a corrupt dealer $\mathcal{P}_3$. By the security of the multiplication verification and MAC check, if the protocol does not abort, then with overwhelming probability the preprocessed data will be correct. $\mathcal{P}_3$ will learn all values shared in the preprocessing phase, but since these are independent of the parties' input to the online phase and since $\mathcal{P}_3$ does not directly participate in the online phase of the protocol, this does not leak any private information.

In Appendix B we prove security more formally, giving this result:

Corollary 2 (Informal). *Combining the Π_{DEAL} with the 2PC online phase of SPDZ and the outsourced MPC additions of [JNO14] leads to an over all protocol that computes any 3-party functionality f with computational security in the presence of a malicious adversary controlling at most one corrupted party.*

4.3 Variants and Extensions

Fairness Fairness is easily achieved in the load-balanced variant of the protocol described in the appendix, similarly to the Lindell-Nof case. Essentially, each

Protocol Π_{DEAL} *(continued from Fig. 2)*

5. *(Triple generation)* Generate M multiplication triples in $M/(N-1)$ batches $(\langle a_i \rangle, \langle b_i \rangle, \langle c_i \rangle)_{i=1}^{N-1}$ of size $N-1$:
   (a) Generate N multiplication triples by doing the following for each $i \in [1, \ldots, N]$:
      i. *(Shares of a_i, b_i)* Repeat 4a twice to get $[\![a_i]\!]$, $[\![b_i]\!]$.
      ii. *(Shares of c_i)* Let $\delta_{2,3} \leftarrow \mathsf{prng}_{2,3}$, $\delta_{1,2} \leftarrow \mathsf{prng}_{1,2}$. $\mathcal{P}_3$ sets $c_i \leftarrow a_i \cdot b_i$, and sends $c_i - \delta_{2,3}$ to $\mathcal{P}_1$. $\mathcal{P}_1, \mathcal{P}_2$ set $[\![c_i]\!]_1 = (c_i - \delta_{2,3}) + \delta_{1,2}$, $[\![c_i]\!]_2 = \delta_{2,3} - \delta_{1,2}$ respectively.
      iii. *(MAC shares)* Repeat 4b to get $[\![\alpha a_i]\!], [\![\alpha b_i]\!], [\![\alpha c_i]\!]$ shared between the three parties. $\mathcal{P}_3$ sends $[\![\alpha a_i]\!]_3$ to $\mathcal{P}_1$, who updates his MAC share $[\![\alpha a_i]\!]_1 = [\![\alpha a_i]\!]_1 + [\![\alpha a_i]\!]_3$; and similarly for $[\![\alpha b_i]\!]_k$, $[\![\alpha c_i]\!]_k$ sent to $\mathcal{P}_2$.
   (b) Check correctness of $(\langle a_i \rangle, \langle b_i \rangle, \langle c_i \rangle)_{i=1}^{N-1}$ by sacrificing $(\langle a_N \rangle, \langle b_N \rangle, \langle c_N \rangle)$:
      i. $\mathcal{P}_3$ computes $c_j = C(\omega_j)$ for $j \in [N+1, 2N-1]$ where $C(x) = A(x)B(x)$; $A(x), B(x)$ of degree $\leq N-1$ s.t. $A(\omega_i) = a_i, B(\omega_i) = b_i$
      ii. $\mathcal{P}_3$ secret-shares $(c_j)_{j=N+1}^{2N-1}$ by sampling $[\![c_j]\!]_2 \leftarrow \mathsf{prng}_{2,3}$ and sending $[\![c_j]\!]_1 = c_j - [\![c_j]\!]_2$ to $\mathcal{P}_1$
      iii. $\mathcal{P}_1, \mathcal{P}_2$ compute $[\![A(s_{1,2})]\!]$, $[\![B(s_{1,2})]\!]$, $[\![C(s_{1,2})]\!]$ linearly from $([\![a_i]\!], [\![b_i]\!], [\![c_i]\!])_{i=1}^{N}$ and $([\![c_i]\!])_{i=N+1}^{2N-1}$
      iv. $\mathcal{P}_1$ sends $[\![A(s_{1,2})]\!]_1$, $[\![B(s_{1,2})]\!]_1$, $[\![C(s_{1,2})]\!]_1$ to $\mathcal{P}_2$. $\mathcal{P}_2$ reconstructs $A(s_{1,2})$, $B(s_{1,2})$, and $C(s_{1,2})$, and aborts if $A(s_{1,2}) \cdot B(s_{1,2}) \neq C(s_{1,2})$
6. *(Input generation)* To generate an input mask $(r, \langle r \rangle)$ for party $\mathcal{P}_i$ the parties run 4a and 4b. Without loss of generality assume $i = 1$. $\mathcal{P}_3$ sends r and $[\![\alpha r]\!]_3$ to $\mathcal{P}_1$ who updates his MAC share $[\![\alpha r]\!]_1 = [\![\alpha r]\!]_1 + [\![\alpha r]\!]_3$.
7. *(MAC check)* The parties check all the MACs on all the generated sharings (input masks and triples). Denote these $([\![a_i]\!])_{i=1}^{L}$ for $L = I_1 + I_2 + 3M$:
   (a) Repeat 4a,4b to get random $\langle a_{L+1} \rangle$ known by $\mathcal{P}_3$ and shared between $\mathcal{P}_1, \mathcal{P}_2$. $\mathcal{P}_3$ sends $[\![\alpha a_{L+1}]\!]_3$ to $\mathcal{P}_1$, who updates his MAC share $[\![\alpha a_{L+1}]\!]_1 = [\![\alpha a_{L+1}]\!]_1 + [\![\alpha a_{L+1}]\!]_3$
   (b) $\mathcal{P}_1, \mathcal{P}_2$ sample PRNG seed $s \leftarrow \mathsf{prng}_{1,2}$. Both send s to $\mathcal{P}_3$, who aborts if inconsistent. All three generate $r_1, \ldots, r_{L+1}$ from the PRNG with seed s.
   (c) $\mathcal{P}_3$ computes $S = \sum_{i=1}^{L+1} r_i a_i$ and sends S to $\mathcal{P}_1$
   (d) $\mathcal{P}_1$ computes $[\![\sigma]\!]_1 \leftarrow (\sum_{i=1}^{L+1} r_i [\![\alpha a_i]\!]_1) - S \cdot [\![\alpha]\!]_1^{1,2}$ and sends $S, [\![\sigma]\!]_1$ to $\mathcal{P}_2$
   (e) $\mathcal{P}_2$ computes $[\![\sigma]\!]_2 \leftarrow (\sum_{i=1}^{L+1} r_i [\![\alpha a_i]\!]_2) - S \cdot [\![\alpha]\!]_2^{1,2}$, aborts if $[\![\sigma]\!]_1 + [\![\sigma]\!]_2 \neq 0$
8. Finally, each party returns its preprocessed MAC key, masks, and triples.

Fig. 3. Protocol Π_{DEAL}

party $\mathcal{P}_i$ inputs MAC key α_i, β_i and mask δ_i (for which we can use input masks). Then, $\alpha_i x + \beta_i$ and $x + \delta_i$ are opened to the other two parties. These values are checked with the SPDZ MAC check and then provided to $\mathcal{P}_i$. The SPDZ MAC check needs to be performed such that everybody agrees on its result, which essentially means that we need to compute a sum $\sum [\![\sigma]\!]_1 + [\![\sigma]\!]_2 + [\![\sigma]\!]_3$ in a fair way. This can be done by letting each party secret-share its summand in a digitally signed way and the other parties forwarding these secret shares, similarly to Dolev-Strong broadcast. We omit the details because of space.

Preprocessing other material Apart from multiplication triples, other random data can be preprocessed in order to speed up specific computations in the SPDZ online phase. For example, Damgård *et. al* [DKL+13] show how to preprocess random square pairs $\langle a \rangle, \langle a^2 \rangle$ for random a. In the online phase $\langle z \rangle = \langle x^2 \rangle$ can be computed from $\langle x \rangle$ by revealing $\varepsilon = x - a$ and setting $\langle z \rangle = 2\varepsilon \langle x \rangle + \langle a^2 \rangle - \varepsilon^2$, which requires only half the communication of regular online multiplications. Our dealer based protocol allows such material to be generated very efficiently.

To preprocess $N-1$ pairs of squares $(\langle a_i \rangle, \langle a_i^2 \rangle)_{i=1}^{N-1}$, we run the protocol for generating multiplication triples as above, except the dealer sets all $b_i = a_i$ (including b_N in the triple to be sacrificed). Note that in this case $B(s) = A(s)$ does not need to be computed or exchanged separately.

Damgård *et. al* also preprocess random bits, i.e., values $\langle x \rangle$ so that $x \in \{0, 1\}$. Such preprocessed values are useful to speed up the online computation of e.g. comparisons [LT13]. To preprocessed random bits $\langle x_1 \rangle, \ldots, \langle x_{N-1} \rangle$, we run the protocol for generating multiplication triples as above, except the dealer sets all $a_i = x_i$ and $b_i = 1 - x_i$ (implying $c_i = 0$). If we use $(\langle a_N \rangle, (1 - \langle a_N \rangle), \langle a_N \rangle (1 - \langle a_N \rangle))$ for random a_N as the extra multiplication triple to be sacrificed, we have $B(x) = 1 - A(x)$ so $B(s)$ does not need to be computed or exchanged. Thus the preprocessing of both a square pair and a bit requires communicating one less field element than a multiplication.

Similarly, we can compute other useful preprocessed material by having the dealer prove the appropriate multiplicative relations using the batchwise multiplication check. For example, random values with their negative powers $\langle r \rangle, \langle r^{-1} \rangle, \ldots, \langle r^{-k} \rangle$ are useful to compute $\langle x^2 \rangle, \ldots, \langle x^k \rangle$ from $\langle x \rangle$ by opening (rx) and taking $\langle x^i \rangle = (rx)^i \langle r^{-i} \rangle$ (e.g., for secure equality [LT13]). Correctness is verified from triples $\langle a_1 \rangle = \langle r \rangle, \langle b_1 \rangle = r^{-1}, \langle c_1 \rangle = 1$, $\langle a_i \rangle = \langle r^{-1} \rangle$, $\langle b_i \rangle = \langle r^{-i+1} \rangle$, $\langle c_i \rangle = \langle r^{-i} \rangle$, $i = 2, \ldots, k$.

Secret-shared random matrix products can be used to efficiently compute matrix products [MZ17]: given random matrices $\langle \mathbf{U} \rangle$, $\langle \mathbf{V} \rangle$, and $\langle \mathbf{W} \rangle = \langle \mathbf{U} \cdot \mathbf{V} \rangle$ of the correct size, matrix product $\langle \mathbf{Z} \rangle = \langle \mathbf{X} \cdot \mathbf{Y} \rangle$ is computed by opening $\langle \mathbf{X} - \mathbf{U} \rangle$ and $\langle \mathbf{Y} - \mathbf{V} \rangle$ and letting

$$\langle \mathbf{Z} \rangle = (\mathbf{X} - \mathbf{U}) \cdot (\mathbf{Y} - \mathbf{V}) + (\mathbf{X} - \mathbf{U}) \cdot \langle \mathbf{V} \rangle + (\mathbf{Y} - \mathbf{V}) \cdot \langle \mathbf{U} \rangle + \langle \mathbf{W} \rangle .$$

To preprocess a random matrix product, the dealer provides secret shares of all $U_{i,j}$, $V_{j,k}$ and products $U_{i,j} \cdot V_{j,k}$, and proves their correctness. The elements of W are computed as linear combinations of these products. Preprocessing in this case reduces *overall* communication, e.g., by a factor 1.5 for 2×2 matrices or a factor 2.5 for 10×10 matrices. Similarly, in the common case of multiplying value (i.e., 1-by-1 matrix) $\langle x \rangle$ with each element in vector (i.e., 1-by-n matrix) $\langle \boldsymbol{y} \rangle$, online communication halves and overall communication decreases by 33%.

Smaller fields As in the Lindell-Nof case, we need a field of size at least $2N \cdot 2^\sigma$, but as there, we can enhance the statistical security of Π_{DEAL} by repeating the multiplication check. Of course, for an overall secure protocol for fields smaller than 2^σ, also modifications to the SPDZ online phase are needed, cf. [DPSZ12].

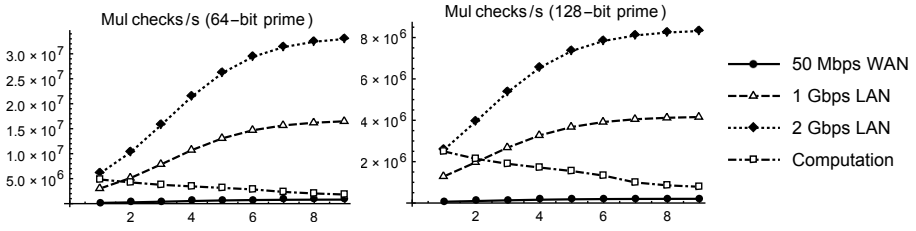

Fig. 4. Number of Lindell-Nof multiplications that can be checked for correctness per second based on the given network capacity or computation effort, with batches of size $2^1, \ldots, 2^9$ for a 64-bit prime (left) or 128-bit prime (right)

5 Performance Evaluation

In this section we present performance estimates suggesting that, despite the computations in our protocols, communication is often still the main bottleneck.

5.1 Implementation Details

To estimate the computation effort of our protocol, we have implemented batch-wise multiplication verification both in the Lindell-Nof and the SPDZ setting. In both cases, we implemented only computation (including PRNG evaluation, secret sharing, reconstruction, and the MAC check) and not communication. For the PRNG, we used the SPDZ-2 implementation based on AES-NI[4].

We implemented the batch check in batch sizes of 2^k using fields $\mathbb{Z}_p$ that allow efficient modular arithmetic and efficient FFTs for those batch sizes (batches do not need to be completely filled up). Batch verification relies heavily on performing FFTs of the size of the batch for performing interpolation; with batch size 2^k, we can use the efficient Cooley-Tukey FFT algorithm. This requires a (2^k)th root of unity in $\mathbb{Z}_p$, or equivalently, $2^k | p-1$. To have fast modular arithmetic, we use pseudo-Mersenne primes $p = 2^s - 2^l + 1$; note that if $k \leq l$ then $2^k | 2^l | p-1$. (We cannot use regular Mersenne primes $2^s - 1$ since $2^k \nmid 2^s - 1$.) In particular, we use our own modular arithmetic/FFT implementation for primes $2^{64} - 2^{10} + 1$ and $2^{128} - 2^{54} + 1$, allowing batches up to 2^9, and 2^{53}, respectively.

To estimate communication complexity, we compute the number of bits that each party needs to send to check correctness of one multiplication. For Lindell-Nof, this is the same for each party; for SPDZ, we use load-balancing so that communication is also evenly spread. The number of multiplications per second is computed as the bandwidth divided by that amount of bits.

5.2 Evaluation Results

Fig. 4, estimates the number of multiplications that can be checked in the Lindell-Nof protocol using Shamir secret sharing, GRR multiplication, and our batchwise

[4] `https://github.com/bristolcrypto/SPDZ-2/`

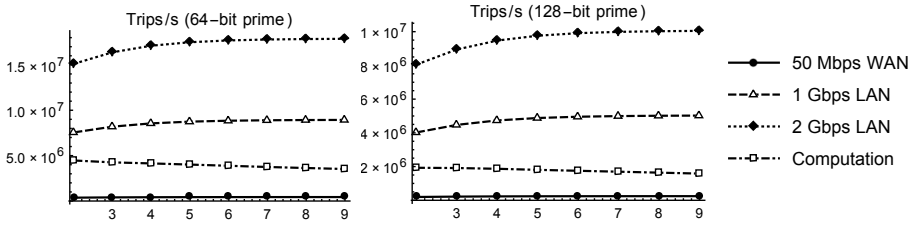

Fig. 5. Number of SPDZ multiplication triples that can be preprocessed per second based on the given network capacity or computation effort (excluding online phase), with batches of size $2^2, \ldots, 2^9$ for a 64-bit prime (left) or 128-bit prime. (right).

check. (Note that this does not include the multiplication to be checked itself.) We show, for different batch sizes 2^k, how many checks are allowed by the bandwidth of a 50 Mbps WAN, a 1 Gbps LAN, and a 2 Gbps LAN. We also show, on a single core of a Amazon M4.large machine (a 2.3/2.4 GHz Intel Xeon E5), how many checks can be handled by the processor. As expected, larger batches are good for communication complexity but bad for computation complexity. With a 1 Gbps LAN and a single core, computation quickly becomes the bottleneck, but still it is possible to process check around 5 million multiplications per second for 64-bit primes and 2 million for 128-bit primes. Note that batchwise verification is trivially parallelizable by checking each batch on a different core, so the number of checks per second can easily be increased by increasing the number of cores. With less than 1 Gbps available, communication quickly becomes the bottleneck rather than computation. We did not run experiments for more than three parties, but in general, the amount of computation should stay roughly the same (since it is dominated by the FFTs) whereas the amount of communication increases as shown in Table 1.

Fig. 5 similarly estimates the number of multiplication triples per second of our SPDZ preprocessing, load-balanced between the three parties. As above, for different batch sizes 2^k, we plot the number of triples that can be generated on a 50 Mbps WAN, a 1 Gbps LAN, and a 2 Gbps LAN; and a single Amazon M4.large core. Note that SPDZ has less communication than Lindell-Nof for small batch sizes; this is because the constant overhead of the SPDZ batch check is very small (just a few field elements). However, for larger batches, Lindell-Nof has less communication (each party sends one field element per check vs. the dealer sends five field elements for one third of the checks). In SPDZ, on a 1Gbps network with a single core, computation is the bottleneck, and around 5 million triples per second are possible for a 64-bit primes or around 2 million triples for a 128-bit prime; with two to four cores, it is possible to reach around 10 million triples for a 64-bit prime or 5 million triples for a 128-bit prime.

Acknowledgements We thank the anonymous reviewers for their useful suggestions. This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement #731583 (SODA).

References

AFL+16. T. Araki, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In *Proceedings of CCS '16*. ACM, 2016.

BFO12. E. Ben-Sasson, S. Fehr, and R. Ostrovsky. Near-linear unconditionally-secure multiparty computation with a dishonest minority. In *Proceedings of CRYPTO*, 2012.

Can00. R. Canetti. Security and Composition of Multi-Party Cryptographic Protocols. *Journal of Cryptology*, 13(1):143–202, 2000.

CB17. H. Corrigan-Gibbs and D. Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *Proceedings of NSDI*, 2017.

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

DN07. I. Damgård and J. B. Nielsen. Scalable and unconditionally secure multiparty computation. In *Proceedings of CRYPTO*, 2007.

DOS17. I. Damgård, C. Orlandi, and M. Simkin. Yet another compiler for active security or: Efficient MPC over arbitrary rings. Cryptology ePrint Archive, Report 2017/908, 2017. `http://eprint.iacr.org/2017/908`.

DPSZ12. I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Proceedings of CRYPTO*, 2012.

DS83. D. Dolev and H. R. Strong. Authenticated algorithms for byzantine agreement. *SIAM J. Comput.*, 12(4):656–666, 1983.

FGMvR02. M. Fitzi, N. Gisin, U. M. Maurer, and O. von Rotz. Unconditional byzantine agreement and multi-party computation secure against dishonest minorities from scratch. In *Proceedings of EUROCRYPT*, 2002.

FLNW17. J. Furukawa, Y. Lindell, A. Nof, and O. Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In *Proceedings of EUROCRYPT*, 2017.

GRR98. R. Gennaro, M. O. Rabin, and T. Rabin. Simplified VSS and fact-track multiparty computations with applications to threshold cryptography. In *Proceedings of PODC*, 1998.

JNO14. T. P. Jakobsen, J. B. Nielsen, and C. Orlandi. A framework for outsourcing of secure computation. In *Proceedings of CCSW '14*, 2014.

KLR06. E. Kushilevitz, Y. Lindell, and T. Rabin. Information-theoretically secure protocols and security under composition. In *Proceedings of STOC '06*, 2006.

KPR17. M. Keller, V. Pastro, and D. Rotaru. Overdrive: Making SPDZ great again. Cryptology ePrint Archive, Report 2017/1230, 2017. `https://eprint.iacr.org/2017/1230`.

LN17. Y. Lindell and A. Nof. A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In *Proceedings of CCS'17*. ACM, 2017.

LT13. H. Lipmaa and T. Toft. Secure equality and greater-than tests with sublinear online complexity. In *Proceeings of ICALP*, 2013.

MZ17. P. Mohassel and Y. Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *Proceedings of S&P*, 2017.

PHGR13. B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *Proceedings of S&P*, 2013.
Sch80. J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27(4):701–717, 1980.
SSW17. P. Scholl, N. P. Smart, and T. Wood. When it's all just too much: Outsourcing MPC preprocessing. Cryptology ePrint Archive, Report 2017/262, 2017. `http://eprint.iacr.org/2017/262`.
SVdV16. B. Schoenmakers, M. Veeningen, and N. de Vreede. Trinocchio: Privacy-preserving outsourcing by distributed verifiable computation. In *Proceedings of ACNS*, 2016.
Zip79. R. Zippel. Probabilistic algorithms for sparse polynomials. In *Proceedings of EUROSAM '79*, 1979.

A Security Proof for our Lindell-Nof Instantiation

In this section, we provide a full specification and security proof for the main protocol from Section 3: our optimised version of the Lindell-Nof protocol with Shamir secret-sharing, our PRNG optimisations, and our batchwise technique for multiplication verification, where we assume the number of parties is $n = 2t+1$[5]. Note that, while Lindell and Nof do prove some important facts about their construction, such a comprehensive proof is not given in [LN17]. By focussing on this particular case, we can provide such a proof reasonably easily (in particular, using Shamir with $n = 2t + 1$ will allow us to deal easily with incorrect shares); moreover, considering this particular instantiation is sufficient since, as shown in Table 1, it covers all our important performance improvements. Due to space constraints, we omit the security proof for our fair protocol variants.

A.1 The protocol

As discussed, the protocol achieves security by combining well-known passively secure implementations of MPC operations with an (in our case, batchwise) multiplication check. For these subroutines, we define few-party variants Fig. 6 and many-party variants in Fig. 7. In any case, we use PRSS, which requires a Setup step (see Fig. 8) that sets up PRNG keys between each pair of parties and between each size-$(t+1)$ subset of parties (only for the few-parties variant). The operations are along the lines of Lindell-Nof, except that we use PRNGs to save on communication for inputting (and so also few-party multiplication and many-party random shared value generation) and many-party multiplication.

The complete actively secure MPC protocol based on the Lindell-Nof framework and our batchwise multiplication check is shown in Fig. 8. Note that we can skip the Lindell-Nof input correctness check essentially because $n = 2t + 1$ (as we show below). To get active security, computations are performed using the above passively secure building blocks; and the batchwise multiplication check is used to verify that the result was correct, as discussed in Section 3.2.

[5] Note that this is also the optimal case because choosing t to be lower will only lessen the privacy and correctness guarantees of the protocol.

Basic subroutines for Shamir-based t-out-of-n MPC (few parties)

Input: Party $\mathcal{P}_i$ inputs value x into the computation as follows:

1. Parties $\mathcal{P}_j$, $j \in [i+1, i+t]$ set $[x]_j \leftarrow \mathsf{prng}_{\{\mathcal{P}_i,\mathcal{P}_j\}}$
2. Party $\mathcal{P}_i$ computes $[x]_j \leftarrow \mathsf{prng}_{\{\mathcal{P}_i,\mathcal{P}_j\}}$ for $j \in [i+1, i+t]$ using its PRNGs and $[x]_i, [x]_{i+t+1}, \ldots, [x]_{i+2t}$ such that they form a consistent sharing of x. Party $\mathcal{P}_i$ sends $[x]_j$ to $\mathcal{P}_j$ for $j \in [i+t+1, i+2t]$

Add/multiply by constant: To compute $[z] = [x] + y$, $[z] = [x] + [y]$, or $[z] = \alpha[x]$, the parties proceed as follows:

1. $\mathcal{P}_i$ sets $[z]_i = [x]_i + y$, $[z]_i = [x]_i + [y]_i$, or $[z]_i = \alpha \cdot [x]_i$

Random shared value: To generate random $[r]$, the parties proceed as follows:

1. Party $\mathcal{P}_i$ computes $r_{\mathcal{S}} \leftarrow \mathsf{prng}_{\mathcal{S}}$ for each size-$(t+1)$ subset $\mathcal{P}_i \in \mathcal{S} \subset \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$, lets $[r_{\mathcal{S}}]_i$ be the unique share of $r_{\mathcal{S}}$ consistent with $[r_{\mathcal{S}}]_j = 0$ for all $j \notin \mathcal{S}$, and lets $[r]_i = \sum_{\mathcal{S}} [r_{\mathcal{S}}]_i$.

Multiply: To compute $[z] = [x] \cdot [y]$, the parties proceed as follows:

1. $\mathcal{P}_i$ computes $z_i = [x]_i \cdot [y]_i$ and inputs z_i to the computation
2. Each party sets $[z]_i = \sum_{j=1}^{n} \lambda_j [z_j]_i$ for degree-$2t$ Lagrange coefficients λ_j

Open: To open a secret-shared value $[x]$ to party $\mathcal{P}_j$, the parties proceed as follows:

- Each party $\mathcal{P}_i$ sends its share $[x]_i$ to $\mathcal{P}_j$
- $\mathcal{P}_j$ reconstructs x based on shares $[x]_1, \ldots, [x]_{t+1}$. It checks if any share $[x]_{t+2}, [x]_n$ is inconsistent with this. If so, it aborts; otherwise it returns x.

Fig. 6. Basic subroutines for Shamir-based t-out-of-n MPC (few parties)

A.2 Security proof

Theorem 1. *Let N be an integer, and let f be a function given by an arithmetic circuit over $\mathbb{Z}_p$, where $p > 2N \cdot 2^{\sigma}$. Assuming secure PRNGs, the protocol in Fig. 8 with batch size N securely computes f with statistical security parameter σ for honest majority.*

Proof. Given adversary $\mathcal{A}$ we need to construct simulator $\mathcal{S}$ that simulates a real-world execution of the protocol with respect to $\mathcal{A}$ in the ideal-world model. For simplicity, let us assume that parties $\mathcal{P}_1, \ldots, \mathcal{P}_d$ are corrupted, where $d \leq t < \frac{n}{2}$. We divide the parties into *corrupted parties* $\mathcal{P}_1, \ldots, \mathcal{P}_d$, *redundant parties* $\mathcal{P}_{d+1}, \ldots, \mathcal{P}_t$, and *essential parties* $\mathcal{P}_{t+1}, \ldots, \mathcal{P}_n$. It is easy to generalize this since the protocol is completely symmetric apart from the use of PRNGs in the input phase, for which it is easy to see that our proof generalises.

The simulator simulates an execution of the protocol with respect to the adversary (controlling the corrupted parties $\mathcal{P}_1, \ldots, \mathcal{P}_d$), as well as with respect to the redundant parties $\mathcal{P}_{d+1}, \ldots, \mathcal{P}_t$ (whose shares are redundant with respect to the shares of the essential parties $\mathcal{P}_{t+1}, \ldots, \mathcal{P}_n$). For each share $[s]$, it keeps track of shares of the first t parties $[s]_1, \ldots, [s]_t$ that are consistent with the shares of essential parties $[s]_{t+1}, \ldots, [s]_n$ (without needing to know the values of these latter shares). For redundant parties $\mathcal{P}_{d+1}, \ldots, \mathcal{P}_t$, it also keeps track of the shares $[s]'_{d+1}, \ldots, [s]'_t$ that they actually hold during the simulated protocol execution (which may not be the same in case the adversary distributed

Basic subroutines for Shamir-based t-out-of-n MPC (many parties)

All steps as in the few-party variant, except:

Random shared value: To generate a batch of $t+1$ random values $[r_i]$, the parties proceed as follows:

1. Party $\mathcal{P}_i$ generates random r'_i and inputs it into the multi-party computation
2. Each party sets $([r_1], \ldots, [r_{t+1}]) = M \cdot ([r'_1], \ldots, [r'_n])^T$, with M a $(t+1) \times n$ Vandermonde matrix.

Multiply: To compute $[z] = [x] \cdot [y]$, the parties proceed as follows:

1. The parties generate random doubly shared value $([r], [\![r]\!])$. This is done in batches of $t+1$ with the following subroutine:
   (a) Party $\mathcal{P}_i$ generates random r'_i and inputs it into the multi-party computation. Parties $\mathcal{P}_i$, $\mathcal{P}_j$ $(i \neq j)$ set $[\![r'_i]\!]_j \leftarrow \mathsf{prng}_{\{\mathcal{P}_i, \mathcal{P}_j\}}$ and $\mathcal{P}_i$ turns it into an additive sharing of r'_i by setting $[\![r'_i]\!]_i = r'_i - \sum_{j \neq i} [\![r'_i]\!]_j$.
   (b) Set $([r_1], \ldots, [r_{t+1}]) = M \cdot ([r'_1], \ldots, [r'_n])^T$ and $([\![r_1]\!], \ldots, [\![r_{t+1}]\!]) = M \cdot ([\![r'_1]\!], \ldots, [\![r'_n]\!])^T$ with M a $(t+1) \times n$ Vandermonde matrix.
2. All $\mathcal{P}_i$ compute $[\![s]\!]_i = \lambda_i \cdot [x]_i \cdot [y]_i - [\![r]\!]_i$ and send it to designated $\mathcal{P}_j$
3. Party $\mathcal{P}_j$ computes $s = \sum_i [\![s]\!]_i$ and inputs it into the computation
4. Set $[z] = [s] + [r]$

Fig. 7. Basic subroutines for Shamir-based t-out-of-n MPC (many parties)

Protocol for efficient Shamir-Based actively secure t-out-of-n MPC

Inputs: each party has a respective number of inputs to the MPC

1. *(Setup)* Set up joint PRNGs between the parties as follows:
   (a) Each size-2 subset $\mathcal{S} \subset \{\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n\}$ sets up a joint PRNG $\mathsf{prng}_{\mathcal{S}}$: one party generates the key and sends it to the other party
   (b) (Only for few parties) Each size-$(t+1)$ subset $\mathcal{S} \subset \{\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n\}$ sets up a joint PRNG $\mathsf{prng}_{\mathcal{S}}$: one party proposes a key, the others echo and check
2. *(Function evaluation, including inputs)* The parties evaluate the function using the subroutines from Fig. 6 or Fig. 7 to input values, apply linear operations, multiply values, and generate secret random values
3. *(Verification stage)* The parties check that all multiplications $[c] = [a] \cdot [b]$ in the computation have been performed correctly, in batches $([a_i], [b_i], [c_i])_{i=1}^{N-1}$, as shown in Fig. 1.
4. *(Output reconstruction)* The parties open output $[y]$ by opening $[y]$ to each of the parties as in Fig. 6 (in particular, aborting in case of inconsistencies)

Fig. 8. Efficient Shamir-Based actively secure t-out-of-n MPC

inconsistent sharings). Finally, it keeps track of the additive error Δ_s made to essential shares $[s]_{t+1}, \ldots, [s]_n$ due to the last layer of multiplications used to compute it (again, without needing to know the value of s itself).

For instance, to simulate *input* by an honest party: for each corrupted party, if its share is defined by the PRNG, set $[x]_i$ accordingly; otherwise, generate $[x]_i$ randomly and send it to the adversary. For each redundant party, set $[x]_i = [x]'_i$ randomly. Set $\Delta_x = 0$. In case a corrupted party provides input, for each honest party (essential or redundant), determine share $[x]'_i$ according to the protocol, i.e., either by generating it from a PRNG or by receiving it from the adversary. Compute x by reconstructing it from $[x]'_{t+1}, \ldots, [x]'_n$, and from this, compute shares $[x]_{d+1}, \ldots, [x]_t$ such that together with the essential shares $[x]'_{t+1}, \ldots, [x]'_n$ they consistently reconstruct to x. Set $\Delta_x = 0$ and provide x to the trusted party.

To simulate *multiplication in the few-party variant*, simulate the input by each party as above. After this simulation, we have shares $[z_j]_i$ consistent with essential shares ($i \in [1,t]$, $j \in [1,n]$); shares $[z_j]'_i$ held by simulated redundant parties ($i \in [d+1,t]$, $j \in [1,n]$); and values z_j input by corrupted parties ($j \in [1,d]$). Set $[z]_i = \sum_j [z_j]_i$; $[z]'_i = \sum_j [z_j]'_i$; and $\Delta_z = \sum_j (z_j - [x]_j[y]_j) + \sum_j ([x]'_j[y]'_j - [x]_j[y]_j)$. Other operations are simulated along similar lines.

To simulate *verification*, use the simulator from Proposition 1. To simulate output reconstruction, receive output y from the trusted party. Simulate opening y to all parties. Send $(c_{t+1}, \ldots, c_n)$ to the trusted party, where $c_i = \perp$ iff the simulated $\mathcal{P}_i$ has aborted. Return what the adversary returns.

We need to show that an ideal-world execution with the above simulator is indistinguishable from a real-world execution with the protocol. As a first step, we can ignore the use of PRNGs between the honest parties (otherwise we could distinguish PRNG-generated data from random), as we have been doing implicitly. Next, one checks that the basic subroutines are simulated in a statistically indistinguishable way and preserve the invariants claimed before: namely, that $[x]_1, \ldots, [x]_d$ are the shares that the adversary would get from following the computation steps; Δ_x is the error in the value that would be reconstructed by parties $\mathcal{P}_{t+1}, \ldots, \mathcal{P}_n$; and $[x]'_{d+1} - [x]_{d+1}, \ldots, [x]'_t - [x]_t$ is the deviation from the shares of $\mathcal{P}_{d+1}, \ldots, \mathcal{P}_t$ that would be consistent with the shares from $\mathcal{P}_{t+1}, \ldots, \mathcal{P}_n$. For the multiplication check, this follows from Proposition 1.

Now, for outputting, if the multiplication check succeeded, we simulate opening the computation result to the value y received from the trusted party. As argued above, success in the multiplication check means $\Delta_v = 0$ for all multiplications performed in the protocol, so that in the corresponding real-world execution, $[y]_{t+1}, \ldots, [y]_n$ are a sharing of the actual function result y. Hence, both in the real-world and ideal-world executions, the honest parties either abort or return y. Since we have simulated the adversary in a statistically indistinguishable way, also the adversary's output is statistically indistinguishable between the real- and ideal-world executions. This completes the proof. □

A.3 Fairness for Many-Party Lindell-Nof

As remarked, the approach for fairness presented before does not work if there are many parties. This is because both the advantage of the adversary is exponential in the number of parties (and so is its computation complexity). As an alternative, inside the multi-party computation, we can compute an "inner"

Shamir sharing $[x_1], \ldots, [x_n]$ of the value x, similarly to [DOS17]. Then, each party sends it share of $[x_i]$, digitally signed, to party $\mathcal{P}_i$. Party $\mathcal{P}_i$ checks that all received shares consistently recombine to one single value (and if this is the case, it can prove this fact to the others), and aborts if they do not. All of this is done before the multiplication check so that, in particular, the multiplication check will not succeed (and hence output reconstruction will not happen) if not all parties have successfully reconstructed their inner share.

B Security Proof of the SPDZ Preprocessing Protocol

In this section, we define ideal functionality $\mathcal{F}_{\text{DEAL}}$ for SPDZ preprocessing (Section B.1), and prove that Π_{DEAL} protocol in Fig. 2 and Fig. 3 securely implements it (Section B.2). Actually, since we want to use the $\mathcal{F}_{\text{DEAL}}$ in our load-balanced version of the SPDZ protocol, we describe a generalized $\mathcal{F}_{\text{DEAL}}$ where each party $\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3$ can play the role of the dealer.

We then show that $\mathcal{F}_{\text{DEAL}}$ is implemented by a "load-balanced version" of the Π_{DEAL} protocol. In this version, the PRNG setup and MAC key generation are performed once, but the other steps are performed three times: once for each pair of parties. Input masks in this setting do not require communication: it is sufficient to just use a the random mask as generated by steps 4a and 4b from Fig. 2 as an input mask for $\mathcal{P}_3$, and similarly for the other parties (where the MAC is additively shared between the three parties). Apart from this, the protocol without load balancing is just a special case of the load-balanced protocol.

B.1 The $\mathcal{F}_{\text{Deal}}$ Functionality

The $\mathcal{F}_{\text{DEAL}}$ ideal functionality is shown in Fig. 9. This model is an adaptation of the normal SPDZ dealer functionality (e.g., [DKL$^+$13,SSW17]) to our setting where one party knows each generated triple. In the normal functionality, the trusted party generates MAC keys, multiplication triples, and masks in such a way the adversary chooses the shares of the corrupted parties, but does not learn the values. In our case, as usual an adversary controlling one of the recipients of a triple can choose its *shares* of preprocessed values; but an adversary controlling its dealer can choose the *values* of multiplication triples and masks as long as they are correct, i.e., the triples contain products, the MACs are correct with respect to the MAC key, and the mask values correspond to their secret sharings.

B.2 Security Proof for the Preprocessing Protocol

We now show security of the load-balanced version of Π_{DEAL} in Fig. 2 and Fig. 3. We provide a simulator $\mathcal{S}_{\text{DEAL}}$ that simulates a real-world execution of Π_{DEAL} with a given adversary $\mathcal{A}$ in the ideal-world model with access to the $\mathcal{F}_{\text{DEAL}}$ functionality. For concreteness, assume that $\mathcal{P}_1$ is corrupted by $\mathcal{A}$; as before, replace the use of PRNGs between $\mathcal{P}_2, \mathcal{P}_3$ by real randomness in a computationally indistinguishable way.

Multiplication triples (dealer $\mathcal{P}_k$ corrupted): To generate multiplication triple $(\langle a\rangle = (\llbracket a\rrbracket, \llbracket \alpha a\rrbracket), \langle b\rangle = (\llbracket b\rrbracket, \llbracket \alpha b\rrbracket), \langle c\rangle = (\llbracket c\rrbracket, \llbracket \alpha c\rrbracket))$:

- Get $\llbracket a\rrbracket_i, \llbracket a\rrbracket_j, \llbracket b\rrbracket_i, \llbracket b\rrbracket_j, \llbracket c\rrbracket_i$ from $\mathcal{A}$
- Compute $a, b, c = a \cdot b$, $\llbracket c\rrbracket_j = c - \llbracket c\rrbracket_i$; randomly share $\llbracket \alpha a\rrbracket, \llbracket \alpha b\rrbracket, \llbracket \alpha c\rrbracket$

Multiplication triples (dealer $\mathcal{P}_k$ honest): To generate multiplication triple $(\langle a\rangle = (\llbracket a\rrbracket, \llbracket \alpha a\rrbracket), \langle b\rangle = (\llbracket b\rrbracket, \llbracket \alpha b\rrbracket), \langle c\rangle = (\llbracket c\rrbracket, \llbracket \alpha c\rrbracket))$:

- Generate random a, b and compute $c = a \cdot b$. For each sharing $\llbracket a\rrbracket$, $\llbracket \alpha a\rrbracket$, $\llbracket b\rrbracket$, $\llbracket \alpha b\rrbracket$, $\llbracket c\rrbracket$, $\llbracket \alpha c\rrbracket$: if $\mathcal{P}_i, \mathcal{P}_j$ are honest, generate random sharing; otherwise, get corrupted share from $\mathcal{A}$ and compute other share

Random mask (dealer corrupted): To generate random mask r known by corrupted $\mathcal{P}_k$ and additively shared between $\mathcal{P}_i, \mathcal{P}_j$:

- Get $r, \llbracket \alpha r\rrbracket_k$ from $\mathcal{A}$
- Generate random shares $\llbracket r\rrbracket_.$, $\llbracket \alpha r\rrbracket_i$, $\llbracket \alpha r\rrbracket_j$ reconstructing to $r, \alpha r$

Random mask (dealer honest): To generate random mask r known by honest $\mathcal{P}_k$ and additively shared between $\mathcal{P}_i, \mathcal{P}_j$:

- Generate random r. For $r, \llbracket \alpha r\rrbracket$: if $\mathcal{P}_i, \mathcal{P}_j$ are honest, generate random sharing, otherwise, get corrupted shares from $\mathcal{A}$ and compute other shares

Functionality $\mathcal{F}_{\mathbf{Deal}}$: To deal a given number of multiplication triples and masks:

1. *(MAC key generation)* Generate random $\alpha \in \mathbb{Z}_p^*$. For each pair of parties $\mathcal{P}_i, \mathcal{P}_j$: if both honest, generate random sharing $\llbracket \alpha\rrbracket$; otherwise, get corrupted share $\llbracket \alpha\rrbracket^{i,j}$ from $\mathcal{A}$ and compute other one
2. Generate the requested number of multiplication triples and random masks
3. For each honest $\mathcal{P}_i$, receive v_i from the adversary. If $v_i = \top$, send $\mathcal{P}_i$'s material (MAC shares $\llbracket \alpha\rrbracket_i$; triple shares $\llbracket \cdot\rrbracket_i$, $\llbracket \alpha \cdot\rrbracket_i$ of obtained triples and $\llbracket \cdot\rrbracket_j$, $\llbracket \cdot\rrbracket_k$ of dealt triples; random mask shares $\llbracket r\rrbracket_i$, $\llbracket \alpha r\rrbracket_i$ and random masks r) to $\mathcal{P}_i$, else send "abort"

Fig. 9. The $\mathcal{F}_{\text{DEAL}}$ functionality

Simulate Setup by computing shares $\llbracket \alpha\rrbracket_1^{1,2}$, $\llbracket \alpha\rrbracket_1^{1,3}$ that $\mathcal{P}_1$ gets and providing these to the trusted party. Also simulate the sampling of $s_{1,2}, s_{1,3}$.

Simulate the generation of a batch of multiplication triples dealt by corrupted $\mathcal{P}_1$, where $\mathcal{P}_i = \mathcal{P}_2$ and $\mathcal{P}_j = \mathcal{P}_3$, as follows. For each triple, get $\llbracket a_l\rrbracket_2$, $\llbracket a_l\rrbracket_3$, $\llbracket b_l\rrbracket_2$, $\llbracket b_l\rrbracket_3$, $\llbracket \alpha a_l\rrbracket_1$, $\llbracket \alpha b_l\rrbracket_1$ from simulating the 4a and 4b subprotocols. Reconstruct a_l, b_l and set $c_l = a_l \cdot b_l$. Compute $\llbracket c_l\rrbracket_3$ using the PRNG, receive $\llbracket c_l\rrbracket_2$ from the adversary, and reconstruct these values to obtain c_l'. Send the relevant values to the trusted party. Set $\delta_{c_l} = c_l' - c_l$. Compute the values $\llbracket \alpha a_l\rrbracket_1, \llbracket \alpha b_l\rrbracket_1, \llbracket \alpha c_l\rrbracket_1$ that $\mathcal{P}_1$ is supposed to send, and set $\delta_{\alpha a_l}, \delta_{\alpha b_l}, \delta_{\alpha c_l}$ based on the values that $\mathcal{P}_1$ actually sends. Simulate batchwise multiplication verification by computing the values $c_{N+1}, \ldots, c_{2N-1}$ that $\mathcal{P}_1$ is supposed to provide, receiving $\llbracket c_{N+1}\rrbracket_2, \ldots, \llbracket c_{2N-1}\rrbracket_2$ and setting $\delta_{c_{N+1}}, \ldots, \delta_{c_{2N-1}}$ accordingly. Simulate an abort if any of $\delta_{c_1}, \ldots, \delta_{c_{2N-1}}$ is nonzero.

Simulate the generation of a batch of multiplication triples dealt by honest $\mathcal{P}_2$ where $\mathcal{P}_i = \mathcal{P}_1$ and $\mathcal{P}_j = \mathcal{P}_3$, as follows. For each triple, simulate generation of $\llbracket a_l\rrbracket, \llbracket b_l\rrbracket$ (subprotocol 4a), and send $\llbracket a_l\rrbracket_1$, $\llbracket \alpha a_l\rrbracket_1$, $\llbracket b_l\rrbracket_1$, $\llbracket \alpha b_l\rrbracket_1$ to the trusted party. Generate a random $\llbracket c_l\rrbracket_1$, send $\llbracket c_l\rrbracket_1 - \delta_{1,2}$ to $\mathcal{P}_1$, and send

$[\![c_l]\!]_1$ to the trusted party. Simulate computing $[\![\alpha c_l]\!]$ to get $[\![\alpha c_l]\!]_1$ (subprotocol 4b); send random $[\![\alpha a_l]\!]_2$ to the adversary and $[\![\alpha c_l]\!]_1 = [\![\alpha c_l]\!]_1 + [\![\alpha a_l]\!]_2$ to the trusted party. Simulate batchwise multiplication verification by sending random $[\![c_{N+1}]\!]_1, \ldots, [\![c_{2N-1}]\!]_1$ to the adversary, and receiving $[\![A(s_{1,3})]\!]_1$, $[\![B(s_{1,3})]\!]_1$, $[\![C(s_{1,3})]\!]_1$. Recompute the values that the adversary should have sent and simulate abort in a mismatch. The case when $\mathcal{P}_i = \mathcal{P}_3$ and $\mathcal{P}_j = \mathcal{P}_1$ is similar.

Simulate the generation of random masks similarly to above, namely, by recomputing the shares that $\mathcal{P}_1$ computes using the PRNG and by simulating the sending of shares to $\mathcal{P}_1$ by sending random values. Send the relevant values to the trusted party.

Simulate the MAC check in case $\mathcal{P}_1$ is the dealer by receiving $[\![\alpha a_M]\!]_k$ and setting $\delta_{\alpha a_M}$ based on the value that $\mathcal{P}_1$ should have sent; and by receiving value S and setting δ_S based on the value that $\mathcal{P}_1$ should have sent. Simulate an abort if any $\delta_{\alpha a_1}, \ldots, \delta_{\alpha a_M}, \delta_S$ is nonzero. Simulate the MAC check in case $\mathcal{P}_1$ acts as $\mathcal{P}_i$ by providing random $[\![\alpha a_M]\!]_k$ to the adversary; computing $[\![\sigma]\!]_1$ and S that $\mathcal{P}_1$ should have sent; and setting δ_σ and δ_S accordingly. Simulate an abort if δ_σ or δ_S is nonzero. Simulate the MAC check in case $\mathcal{P}_1$ acts as $\mathcal{P}_j$ by recomputing the value $[\![\sigma]\!]_1$ that the adversary should compute and sending $[\![\sigma]\!]_. = -[\![\sigma]\!]_1$ Simulate the overall execution by following the steps of Π_{DEAL}, checking which honest parties have aborted, and for them sending $v_i = \perp$ to the trusted party; and returning as adversary's output whatever is returned by the simulated adversary.

Theorem 2. *Let N be an integer and let $p > 2N \cdot 2^\sigma$. Assuming secure PRNGs, the load-balanced variant of the protocol in Fig. 2, Fig. 3 securely implements $\mathcal{F}_{\text{DEAL}}$ with statistical security parameter σ for honest majority.*

Proof. In checking that the above simulator simulates a real-world execution in an indistinguishable way, the most difficult part is to check that a simulated dealer cannot provide wrong products or MACs for multiplication triples. Indeed, the other parts follow by inspection. For multiplication triples dealt by $\mathcal{P}_1$, note that the simulator simulates an abort at the first attempt of $\mathcal{P}_1$ to deal a batch of triples with $(\delta_{c_1}, \ldots, \delta_{c_{2N-1}}) \neq \mathbf{0}$. In this first attempt, in the real-world execution $\mathcal{P}_2, \mathcal{P}_3$ compute $A(s_{2,3})$, $B(s_{2,3})$, and $C(s_{2,3}) + D(s_{2,3})$, where $C(x) = A(x)B(x)$ and $D(x)$ is such that $D(\omega_i) = \delta_{c_i}$ for $i = 1, \ldots, 2N-1$. Equality holds if and only if $D(s_{2,3}) = 0$, and since $s_{2,3}$ is chosen randomly and independently from the actions of $\mathcal{P}_1$, this happens with probability $(2n-2)/(|\mathbb{Z}_p| - 2n)$. So except with this negligible probability, the real-world leads to an abort by the honest parties as well. So, if there is a first attempt at cheating, simulation is indistinguishable (and if there is no attempt at cheating at all then simulation is of course indistinguishable, too).

If $\mathcal{P}_1$ acts as party $\mathcal{P}_i$ in triple generation and it provides values $[\![A(s_{1,3})]\!]_1 + \delta_1$, $[\![B(s_{1,3})]\!]_1 + \delta_2$, $[\![C(s_{1,3})]\!]_1 + \delta_3$ with $(\delta_1, \delta_2, \delta_3) \neq \mathbf{0}$, then the ideal-world simulator aborts. In the real world, the adversary chooses $\delta_1, \delta_2, \delta_3$ independently from the random values $A(s), B(s)$, and this leads to an abort if the polynomial

map $(A(s), B(s)) \mapsto (A(s) + \delta_1) \cdot (B(s) + \delta_2) - (A(s)B(s) + \delta_3)$ happens to map to zero. By the Schwartz-Zippel lemma, this happens with at most the negligible probability $2/|\mathbb{Z}_p|$, and otherwise the simulation is indistinguishable. If $\mathcal{P}_1$ acts as party $\mathcal{P}_j$, then indistinguishability is also clear since the adversary receives correct random shares $[\![A(s_{3,1})]\!]_1, [\![B(s_{3,1})]\!]_1, [\![C(s_{3,1})]\!]_1$, and in both cases the adversary gets to choose whether or not to make the multiplication check fail.

For the MAC check, first consider the case when $\mathcal{P}_1$ is the dealer. In this case, the value that is opened by the MAC check is $\sigma = \sum r_l(\alpha a_l + \delta_{\alpha a_l}) - ((\sum r_l a_l) + \delta_S) \cdot \alpha = \sum r_l \delta_{\alpha a_l} - \alpha\delta_S$. In the ideal world, the MAC check fails whenever one of the δ's is nonzero. In the real world, the MAC check fails whenever $\sigma \neq 0$. Now, suppose some δ_{a_l} is nonzero. The adversary has chosen the δ_{a_l}'s independently from the random coefficients r_l used (recall that we replaced the PRNG by random values before), so by the Schwartz-Zippel lemma, the probability that $\sum r_l \delta_{\alpha a_l}$ is zero is negligible; and if it is nonzero, the probability that it matches $\alpha\delta_S$ is negligible as well (since the adversary chooses δ_S independently from the α used). On the other hand, if all δ_{a_l} and δ_S are nonzero, then clearly $\sigma \neq 0$ as well. This settles the indistinguishability in this case. Now, consider the case when $\mathcal{P}_1$ acts as $\mathcal{P}_i$. In this case, $\mathcal{P}_j$ computes $\sigma = \sum r_l \alpha a_l - S \cdot [\![\alpha]\!]_i^{i,j} - (S + \delta_S) \cdot [\![\alpha]\!]_j^{i,j} + \delta_\sigma = \delta_\sigma - \delta_S \cdot [\![\alpha]\!]_j^{i,j}$. In the ideal world, the MAC check fails whenever one of δ_σ, δ_S is nonzero. In the real world, note that the adversary chooses the δ's independently from the $[\![\alpha]\!]_j^{i,j}$ used. So, except with negligible probability, the MAC check fails here as well, again establishing indistinguishability. Finally, if $\mathcal{P}_1$ acts as $\mathcal{P}_j$ then indistinguishability is clear since the adversary just receives the share it excepts and can then choose whether or not to let the MAC check succeed.

Overall, we have given a simulator such that an ideal-world execution with the simulator gives an indistinguishable result from a real-world execution with the adversary. Hence, the protocol implements $\mathcal{F}_{\text{DEAL}}$, as we wanted to show. □

C The Load-Balanced SPDZ Online Phase

We now provide details of our load-balanced SPDZ online phase, where the multiplications in the computation are evenly divided between the pairs of parties in order to evenly divide the preprocessing and online work.

Similarly to the normal SPDZ protocol, we make use of additive secret sharing to share values. However, unlike in SPDZ, a value x can either be additively shared either between all three parties (i.e., $\mathcal{P}_1$ has $[\![x]\!]_1$, $\mathcal{P}_2$ has $[\![x]\!]_2$, and $\mathcal{P}_3$ has $[\![x]\!]_3$ such that $[\![x]\!]_1 + [\![x]\!]_2 + [\![x]\!]_3 = x$); or between any pair of them (in which case the third share, e.g., $[\![x]\!]_3$, is $\perp$ and $[\![x]\!]_1 + [\![x]\!]_2 = x$. This reflects the "load-balanced" nature of the protocol, e.g., multiplication using a multiplication triple from $\mathcal{P}_1$ results in a value shared between $\mathcal{P}_2$ and $\mathcal{P}_3$. Apart from the value x itself, also an information-theoretic MAC $[\![\alpha \cdot x]\!]$ is secret-shared between the three parties (this value is always shared between all parties). As a result of our preprocessing, no individual party knows the MAC key α but this key itself is additively shared between $\mathcal{P}_1, \mathcal{P}_2$ as $[\![\alpha]\!]_1^{1,2}, [\![\alpha]\!]_2^{1,2}$, and similarly between $\mathcal{P}_1, \mathcal{P}_3$ and between $\mathcal{P}_2, \mathcal{P}_3$. We write $\langle x \rangle = ([\![x]\!], [\![\alpha x]\!])$ for a shared value with its MAC.

C.1 Primitive operations

Operations on such secret-shared values such as addition and multiplication are performed similarly to SPDZ, except that we take into account that different operands may be secret-shared between different sets of parties.

To let a party $\mathcal{P}_i$ **input** a value x into the multi-party computation, we consume a random mask from the preprocessing phase: a random value r that is known by $\mathcal{P}_i$ and secret shared as $\langle r \rangle = (\llbracket r \rrbracket, \llbracket \alpha r \rrbracket)$ between the other two parties $\mathcal{P}_j, \mathcal{P}_k$. Party $\mathcal{P}_i$ computes $\varepsilon = x - r$ and sends it to one of the other two parties, say, $\mathcal{P}_j$. This becomes a sharing of x between $\mathcal{P}_i$ and $\mathcal{P}_j$ by setting $\langle x \rangle_i = (r, \llbracket \alpha r \rrbracket_i + \varepsilon \cdot \llbracket \alpha \rrbracket_i^{i,j})$; $\langle x \rangle_j = (\varepsilon, \llbracket \alpha r \rrbracket_j + \varepsilon \cdot \llbracket \alpha \rrbracket_j^{i,j})$; $\langle x \rangle_k = (\bot, \llbracket \alpha r \rrbracket_k)$. Alternatively, it can become a sharing of x between $\mathcal{P}_j$ and $\mathcal{P}_k$ by setting $\langle x \rangle_i = (\bot, \llbracket \alpha r \rrbracket_i + \varepsilon \cdot \llbracket \alpha \rrbracket_i^{i,j})$; $\langle x \rangle_j = (\llbracket r \rrbracket_j + \varepsilon, \llbracket \alpha r \rrbracket_j + \varepsilon \cdot \llbracket \alpha \rrbracket_j^{i,j})$; $\langle x \rangle_k = (\llbracket r \rrbracket_k, \llbracket \alpha r \rrbracket_k)$.

To perform **linear operations** such as addition or multiplication by a constant, apply them directly on the sharings, with the convention that $\alpha \cdot \bot = \bot$ and $x + \bot = x$.

To generate **random value** $\langle r \rangle$, consume three random masks: r_1 known by $\mathcal{P}_1$ and secret-shared between the other two; r_2 known by $\mathcal{P}_2$ and r_3 known by $\mathcal{P}_3$, and set $\langle r \rangle = \langle r_1 \rangle + \langle r_2 \rangle + \langle r_3 \rangle$.

Finally, to perform a **multiplication** $\langle z \rangle \leftarrow \langle x \rangle \cdot \langle y \rangle$ such that the result $\langle z \rangle$ is secret-shared between $\mathcal{P}_i$ and $\mathcal{P}_j$, use a multiplication triple $(\langle a \rangle, \langle b \rangle, \langle c \rangle)$ for $\mathcal{P}_i$ and $\mathcal{P}_j$ from the preprocessing phase: i.e., secret shares of values a, b, c such that $c = a \cdot b$ that are secret-shared between $\mathcal{P}_i$ and $\mathcal{P}_j$. First, $\varepsilon = x - a$ is opened to $\mathcal{P}_i$ and $\mathcal{P}_j$. This costs one round and two messages if x is shared between $\mathcal{P}_i$ and $\mathcal{P}_j$ themselves (they can simply exchange their shares of $\llbracket x - a \rrbracket$); two rounds and two messages if x is shared between one of the parties and $\mathcal{P}_k$ ($\mathcal{P}_k$ sends $\llbracket x \rrbracket_k - \llbracket a \rrbracket_j$ to $\mathcal{P}_j$ who sends $x - a$ to $\mathcal{P}_i$); or two rounds and three messages if x is shared between all three parties (the parties use their PRNGs to re-randomise $\llbracket x \rrbracket$; $\mathcal{P}_k$ sends its share $\llbracket x \rrbracket_k$ to $\mathcal{P}_i$ or $\mathcal{P}_j$ and they exchange their shares of $\llbracket x - a \rrbracket$). Similarly, $\rho = y - b$ is also opened and $\langle z \rangle$ is computed linearly as $\langle z \rangle_i = (\varepsilon\rho + \varepsilon \cdot \llbracket b \rrbracket_i + \rho \cdot \llbracket a \rrbracket_i + \llbracket c \rrbracket_i, \varepsilon\rho \cdot \llbracket \alpha \rrbracket_i^{i,j} + \varepsilon \cdot \llbracket \alpha b \rrbracket_i + \rho \cdot \llbracket \alpha a \rrbracket_i + \llbracket \alpha c \rrbracket_i)$; $\langle z \rangle_j = (\varepsilon\rho + \varepsilon \cdot \llbracket b \rrbracket_j + \rho \cdot \llbracket a \rrbracket_j + \llbracket c \rrbracket_j, \varepsilon\rho \cdot \llbracket \alpha \rrbracket_j^{i,j} + \varepsilon \cdot \llbracket \alpha b \rrbracket_j + \rho \cdot \llbracket \alpha a \rrbracket_i + \llbracket \alpha c \rrbracket_j)$; $\langle z \rangle_k = (\bot, 0)$.

Note that, for multiplication, the number of rounds and messages depends on how the inputs are shared. In the normal SPDZ case, where the inputs are shared between the same two parties, we get the normal SPDZ costs: one round and two messages. We need an extra round for inputs from between different pairs, and also an extra message for inputs shared between all three parties (e.g., random values). We expect that normal computations can be partitioned well into groups of operations that can be assigned to pairs of parties, in which case few extra messages or rounds will be needed.

C.2 Overall Protocol

The overall online phase of our protocol is given in Fig. 10. First, the preprocessing phase of the protocol (discussed above) is performed to obtain the

Load-Balanced SPDZ-Based 1-out-of-3 MPC

Inputs: each party has a respective number of inputs to the MPC

1. *(Preprocessing)* The parties perform the preprocessing phase of the protocol, obtaining the appropriate number of multiplication triples and random masks
2. *(PRNG setup)* Each pair of parties $\mathcal{P}_i, \mathcal{P}_j$ sets up joint PRNG $\mathsf{prng}_{i,j}$
3. *(Function evaluation, including inputs)* The parties evaluate the function using the described ways to input values, apply linear operations, multiply values, and generate secret random values
4. *(Masked opening)* For each value $\langle y \rangle$ that needs to be output, the parties take a random mask r that is known by one of the parties $\mathcal{P}_k$ and secret-shared between the other two parties $\mathcal{P}_i$ and $\mathcal{P}_j$, and open the value $y + r$ to $\mathcal{P}_i$ and $\mathcal{P}_j$. (If $[\![y]\!]_k \neq \perp$, then $\mathcal{P}_j, \mathcal{P}_k$ generate $\delta \leftarrow \mathsf{prng}_{j,k}$ and $\mathcal{P}_k$ sends $[\![y]\!]_k + \delta$ to $\mathcal{P}_i$; otherwise, set $\delta = 0$. $\mathcal{P}_i$ and $\mathcal{P}_j$ then exchange $[\![y]\!]_i + [\![r]\!]_i + ([\![y]\!]_k + \delta)$ and $[\![y]\!]_j + [\![r]\!]_j - \delta$. Finally, $\mathcal{P}_i, \mathcal{P}_j$ reconstruct $y' := y + r$)
5. *(MAC check)* The parties perform a MAC check on all values $a_1, \ldots, a_M$ that have been opened to a pair of parties in the protocol (including the y's above):
   (a) Jointly generate a random PRNG seed s (see text)
   (b) Sample $r_1, \ldots, r_M$ from the PRNG
   (c) Party $\mathcal{P}_i$ computes $[\![\sigma]\!]_i = \sum_l r_l \cdot [\![\alpha a_l]\!]_i - \sum_l r_l a_l \cdot [\![\alpha]\!]_i^{i,j} - \sum_l r_l a_l \cdot [\![\alpha]\!]_i^{i,k}$, where the first sum is over all opened values; the second sum is over values opened to $\mathcal{P}_i, \mathcal{P}_j$; and the third sum is over values opened to $\mathcal{P}_i, \mathcal{P}_k$
   (d) The parties securely check that $[\![\sigma]\!]_1 + [\![\sigma]\!]_2 + [\![\sigma]\!]_3 = 0$ (see text)
6. *(Output reconstruction)* For value $[\![y]\!]$ to be output that was opened to $\mathcal{P}_i, \mathcal{P}_j$: $\mathcal{P}_i$ and $\mathcal{P}_j$ send $y' = y + r$ to $\mathcal{P}_k$ and $[\![r]\!]$ to each other. $\mathcal{P}_k$ sends r to $\mathcal{P}_i$ and $\mathcal{P}_j$. $\mathcal{P}_i$ and $\mathcal{P}_j$ compute y from y', r and from $y', [\![r]\!]_i, [\![r]\!]_j$. $\mathcal{P}_k$ computes y from r and received y's. Each party accepts y if the values are consistent, else aborts

Fig. 10. Load-Balanced SPDZ-Based 1-out-of-3 MPC

multiplication triples and random masks needed for the computation. Then, the computation is performed using the operations discussed above. Finally, the output of the computation is determined, with a MAC checking procedure being used to guarantee that the output values are correct.

Compared to the original SPDZ protocol, we make two changes to the output procedure. First, making use of our honest majority setting, we can eliminate one of the MAC checks that SPDZ does. In SPDZ, first a MAC check is performed on all intermediate values opened during the protocol; only after this check has succeeded, the output is reconstructed and a final MAC check is performed on it. The reason this is done is that reconstructing the output before checking the intermediate values can leak information. In our honest majority setting, we can avoid this need for two MAC checks by opening a masked output to two parties, with the mask known by the third party (line 4). After applying the MAC check to the masked output, each party determines the unmasked output twice: once with one party and once with the other. If both are the same then, since one party is honest, the output must be correct (line 6).

Our second change to the output procedure is a modified MAC check. In the MAC check, the parties check the correctness of each opened value a essentially by comparing the additive shares $[\![\alpha a]\!]$ of the MAC to a times their additive shares $[\![\alpha]\!]$ of the MAC key. The normal MAC check [DKL+13] is between two parties that both know a number of reconstructed values and have an additive sharing of their MACs; in our case, two of the three parties know the reconstructed values but the MAC is shared between three parties. We modify the procedure so that still, all reconstructed values can be opened in a single check (line 5).

Apart from this modification, we also modify the methods used in the MAC check to jointly sample a random PRNG seed and fairly exchange a number of values that supposedly add up to zero. In the SPDZ protocol, both are done with a commitment scheme, but in our honest majority setting, we can essentially replace this with secret sharing. In particular, to jointly sample a random PRNG seed, each pair of parties $\mathcal{P}_i, \mathcal{P}_j$ generates $s_{i,j} \leftarrow \mathsf{prng}_{i,j}$ and both send it to the third party. All parties abort if they receive any inconsistent values, and otherwise set $s := s_{1,2} \oplus s_{1,3} \oplus s_{2,3}$. To fairly exchange the values $[\![\sigma]\!]_1$, $[\![\sigma]\!]_2$ and $[\![\sigma]\!]_3$, the parties first secret-share their values: party $\mathcal{P}_1$ additively shares $[\![\sigma]\!]_1$ between $\mathcal{P}_2, \mathcal{P}_3$ as shares $[\![\sigma_1]\!]_2$ and $[\![\sigma_1]\!]_3$, and similarly for $\mathcal{P}_2$ and $\mathcal{P}_3$. Then, $\mathcal{P}_1$ sends $(\sigma_1 + [\![\sigma_3]\!]_1)$ to $\mathcal{P}_2$ and $(\sigma_1 + [\![\sigma_2]\!]_1)$ to $\mathcal{P}_3$; $\mathcal{P}_2$ sends $(\sigma_2 + [\![\sigma_3]\!]_2)$ to $\mathcal{P}_1$ and $([\![\sigma_1]\!]_2 + \sigma_2)$ to $\mathcal{P}_3$; and $\mathcal{P}_3$ sends $([\![\sigma_2]\!]_3 + \sigma_3)$ to $\mathcal{P}_1$ and $([\![\sigma_1]\!]_3 + \sigma_3)$ to $\mathcal{P}_2$. Finally, $\mathcal{P}_1$ computes σ as $\sigma_1 + (\sigma_2 + [\![\sigma_3]\!]_2) + [\![\sigma_3]\!]_1$ and $\sigma_1 + [\![\sigma_2]\!]_1 + ([\![\sigma_2]\!]_3 + \sigma_3)$ and aborts if they are different or nonzero. Similarly for $\mathcal{P}_2, \mathcal{P}_3$.

C.3 Security proof

We now prove security of the protocol. We prove that the protocol is secure in the $\mathcal{F}_{\text{DEAL}}$-hybrid model; by applying the [Can00] composition theorem, we obtain that the protocol together with the dealer protocol securely computes f.

We present an ideal-world simulator simulating a real-world protocol execution in the presence of an adversary $\mathcal{A}$ actively corrupting one party, say, $\mathcal{P}_1$. The simulator interacts with the trusted party and simulates calls to $\mathcal{F}_{\text{DEAL}}$ with respect to $\mathcal{A}$. Moreover, for each sharing $[\![x]\!]$ it keeps track of the shares of the corrupted party, and for each opened value x it keeps track of the simulated opened value and the error Δ_x introduced to this value by the adversary.

To simulates the adversary's call to $\mathcal{F}_{\text{DEAL}}$, receive the information that the adversary provides: MAC shares $[\![\alpha]\!]_1$, $[\![\alpha]\!]_1$; dealt multiplication triple shares $[\![a]\!]_2$, $[\![a]\!]_3$, $[\![b]\!]_2$, $[\![b]\!]_3$, $[\![c]\!]_2$; received multiplication triple shares $[\![a]\!]_1$, $[\![b]\!]_1$, $[\![c]\!]_1$, $[\![a]\!]_1$, $[\![b]\!]_1$, $[\![c]\!]_1$, $[\![\alpha a]\!]_1$, $[\![\alpha b]\!]_1$, $[\![\alpha c]\!]_1$; dealt random masks $r, [\![\alpha r]\!]_1$; and received random masks $[\![r]\!]_1, [\![\alpha r]\!]_1$. Also, receive instructions on whether to provide the preprocessed data to the honest parties $\mathcal{P}_2, \mathcal{P}_3$; if not, simulate an abort.

To simulate input x by $\mathcal{P}_1$, receive ε from $\mathcal{A}$, compute $x = \varepsilon + r$ based on pre-dealt value r and provide x to the trusted party. To simulate input x by a honest party for which the corrrupted party is supposed to receive a value ε, generate random ε and send it to $\mathcal{A}$. In both cases, compute share $[\![x]\!]$ of the corrupted party. To simulate linear operations, compute corrupted share $[\![z]\!]$

locally. To simulate multiplication, simulate opening of $\varepsilon = x - a$ and $\rho = y - b$: compute what the corrupted party should have sent, and set errors $\Delta_\varepsilon, \Delta_\rho$ based on the values actually sent. Send random values for the honest parties.

To simulate a MAC check on opened values $a_1, \ldots, a_M$, simulate obtaining a random seed s (details left out because of space). Generate coefficients $r_1, \ldots, r_M$ using the PRNG with seed s. Abort if $\Delta_{a_i} \neq 0$ for some a_i but $\sum r_i a_i = 0$. If $\Delta_{a_i} \neq 0$ for some a_i, generate random nonzero σ, otherwise, set $\sigma = 0$. Simulate the secure check whether $[\![\sigma]\!]_1 + [\![\sigma]\!]_2 + [\![\sigma]\!]_3$ (details left out because of space) and abort the simulation if $\sigma \neq 0$ but $\sigma \neq 0$ but $\sigma + \Delta_\sigma = 0$.

To simulate output where $\mathcal{P}_k = \mathcal{P}_1$, simulate opening y' to $\mathcal{P}_2, \mathcal{P}_3$, giving error $\Delta_{y'}$, and simulate the MAC check as above. Receive output y from the trusted party, compute $y' = y + r$ and send it to the adversary on behalf of $\mathcal{P}_2, \mathcal{P}_3$. Check whether $\mathcal{A}$ provides correct r on behalf of $\mathcal{P}_1$ and, based on this, instruct the trusted party on whether or not to provide output to $\mathcal{P}_2, \mathcal{P}_3$.

To simulate output where $\mathcal{P}_i = \mathcal{P}_1$ and e.g. $\mathcal{P}_j = \mathcal{P}_2$, generate random y' and simulate opening to that value, giving error $\Delta_{y'}$, and simulate the MAC check as above. Receive y from the trusted party, and send values $r, [\![r]\!]$ such that $y + r = y + [\![r]\!]_1 + [\![r]\!]_2 = y'$. Receive values from $\mathcal{A}$ on behalf of the honest parties and instruct the trusted party on whether to provide output to the honest parties based on whether or not the adversary provides the expected values.

Theorem 3. *Let N be an integer, and let f be a function given by an arithmetic circuit over $\mathbb{Z}_p$, where $p > 2^\sigma$. Assuming secure PRNGs, the protocol in Fig. 10 securely computes f with statistical security parameter σ for honest majority in the $\mathcal{F}_{\text{DEAL}}$-hybrid model.*

Proof. We have to show indistinguishability of the joint honest and adversarial outputs between the real-world and ideal-world executions.

For the computation itself (i.e., input, linear operation, multiplication, and output), observe that we can indeed simulate all exchanged values by random values. For input, exchanged value $x - r$ is random to $\mathcal{P}_1$ because the random mask r. For multiplication, when opening $x - a$ and $y - b$, this is by inclusion of the uniformly random value $[\![a]\!]_.$; if the input is additionally shared between the three parties, this is because $[\![x]\!]_k$ is masked by random δ, and the other values contain the uniformly random value $[\![a]\!]_.$. Similarly, for outputting, as a result of masks δ and $[\![r]\!]_i$ we can simulate the exchanged values by random values.

Now, consider the MAC check. Recall that the ideal-world simulator aborts if $\Delta_{a_i} \neq 0$ for some a_i but $\sum r_i a_i = 0$. Suppose the adversary can let this happen with probability at least $1/|\mathbb{Z}_p| + \mu(\kappa)$, where μ is nonnegligible in the security parameter. Note that if the coefficients r_i are generated uniformly at random, the probability that the simulator fails is $1/|\mathbb{Z}_p|$. This is because the mapping $(r_1, \ldots, r_M) \mapsto \sum_i r_i \Delta_{a_i} = 0$ is a nonzero linear mapping which has a null space of dimension $M - 1$, so randomly generated coefficients have probability $|\mathbb{Z}_p|^{M-1}/\mathbb{Z}_p^M = 1/|\mathbb{Z}_p|$ of ending up in the null space. Hence, we can make a distinguisher for the PRNG by receiving coefficients $r_1, \ldots, r_M$ generated either uniformly at random or from the PRNG; simulating the protocol

to obtain $\Delta_{a_1}, \ldots, \Delta_{a_M}$ and outputting whether $\sum_i r_i \Delta_{a_i} = 0$. Since the adversary chooses Δ_{a_i} independently of the seed generated in the protocol, the distinguisher has advantage $\mu(\kappa)$, contradicting security of the PRNG. So from now on, we can assume that $\Delta_{a_i} \neq 0$ for all i if and only if $\sum r_i \Delta_{a_i} = 0$.

Now, in the real-world execution, the value opened by the MAC check is $\sigma = \sum r_l \alpha a_l - \sum r_l (a_l + \Delta_{a_l})\alpha + \Delta_\sigma = \Delta_\sigma - \alpha \sum r_l \Delta_{a_l}$. Since the adversary does not have any information about α, $\alpha \sum r_l \Delta_{a_l}$ can be simulated by a random nonzero value if any $\Delta_{a_i} \neq 0$ or by zero otherwise, as happens in the simulation. Also, in both the real-world and ideal-world simulation, fair opening to $\sigma + \Delta_\sigma$ is simulated. Now, the ideal-world simulator aborts if $\sigma \neq 0$ but $\sigma + \Delta_\sigma = 0$. Note that, since the adversary has to choose Δ_σ completely independently from σ, this happens with negligible probability. Hence also this second failure of the ideal-world simulator happens only with negligible probability, so actually, the simulator provides an indistinguishable simulation of the MAC check.

We have established so far that the simulator simulates protocol execution with respect to the adversary in an indistinguishable way. Moreover, the honest parties return the output given by the real-world protocol (i.e., they do not abort) if and only if they return the output computed by the trusted party in the ideal-world simulation. Now, note that the parties provide an output in the simulation only if $\delta_a = 0$ for all values a opened during the computation (during multiplication and output). But in this case, the adversary has not introduced any errors into the computation, hence the values output by the honest parties in the real-world protocol execution are equal to function result, which is what the trusted party computes and the honest parties return in the ideal-world protocol execution. This concludes the proof. □