SUPERPACK: Dishonest Majority MPC with Constant Online Communication

Daniel Escudero,[1] Vipul Goyal,[2][3] Antigoni Polychroniadou,[1] Yifan Song[4] and Chenkai Weng[5]

[1] J.P. Morgan AI Research & J.P. Morgan AlgoCRYPT CoE, NY, USA
[2] NTT Research, CA, USA
[3] Carnegie Mellon University, PA, USA
[4] Tsinghua University, Beijing, China
[5] Northwestern University, IL, USA

Abstract. In this work we present a novel actively secure dishonest majority MPC protocol, SUPERPACK, whose efficiency improves as the number of *honest* parties increases. Concretely, let $0 < \epsilon < 1/2$ and consider an adversary that corrupts $t < n(1-\epsilon)$ out of n parties. SUPERPACK requires $6/\epsilon$ field elements of online communication per multiplication gate across all parties, assuming circuit-dependent preprocessing, and $10/\epsilon$ assuming circuit-independent preprocessing. In contrast, most of the previous works such as SPDZ (Damgård *et al*, ESORICS 2013) and its derivatives perform the same regardless of whether there is only one honest party or a constant (non-majority) fraction of honest parties. A notable exception is due to Goyal *et al* (CRYPTO 2022), which achieves $58/\epsilon + 96/\epsilon^2$ field elements assuming circuit-independent preprocessing. Our work improves this result substantially by a factor of at least 25 in the circuit-independent preprocessing model.
Practically, we also compare our work with the best concretely efficient online protocol Turbospeedz (Ben-Efraim *et al*, ACNS 2019), which achieves $2(1-\epsilon)n$ field elements per multiplication gate among all parties. Our online protocol improves over Turbospeedz as n grows, and as ϵ approaches $1/2$. For example, if there are 90% corruptions ($\epsilon = 0.1$), with $n = 50$ our online protocol is $1.5\times$ better than Turbospeedz and with $n = 100$ this factor is $3\times$, but for 70% corruptions ($\epsilon = 0.3$) with $n = 50$ our online protocol is $3.5\times$ better, and for $n = 100$ this factor is $7\times$.
Our circuit-dependent preprocessing can be instantiated from OLE/VOLE. The amount of OLE/VOLE correlations required in our work is a factor of $\approx \epsilon n/2$ smaller than these required by Le Mans (Rachuri and Scholl, CRYPTO 2022) leveraged to instantiate the preprocessing of Turbospeedz.
Our dishonest majority protocol relies on packed secret-sharing and leverages ideas from the honest majority TURBOPACK (Escudero *et al*, CCS 2022) protocol to achieve concrete efficiency for any circuit topology, not only SIMD. We implement both SUPERPACK and Turbospeedz and verify with experimental results that our approach indeed leads to more competitive runtimes in distributed environments with a moderately large number of parties.

1 Introduction

Secure multiparty computation (MPC) protocols enable a set of parties $P_1, \ldots, P_n$ to securely compute a function on their private inputs while leaking only the final output. MPC protocols remain secure even if t out of the n parties are corrupted. There are honest majority protocols, which are designed to tolerate at most a minority of corruptions, or in other words, they assume that $t < n/2$. On the other hand, protocols in the dishonest majority setting accommodate $t \geq n/2$. Honest majority MPC protocols can offer information-theoretic security (that is, they do not need to depend on computational assumptions, which also makes them more efficient), or guaranteed output delivery (that is, all honest parties are guaranteed to receive the output of the computation). However, dishonest majority protocols tolerate a larger number of corruptions at the expense of relying on computational assumptions and sacrificing fairness, and guaranteeing output delivery.

Communication complexity is a key measure of efficiency for MPC. Over the last few decades, great progress has been made in the design of communication-efficient honest majority protocols [BGW88,DN07,GIP+14,LN17,CGH+18,GS20,BGIN20,GLO+21,EGPS22]. In particular, the recent work [EGPS22] shows that it is possible to achieve *constant* communication complexity among all parties (i.e., $O(1)$) per multiplication gate in the online phase while maintaining *linear* communication complexity in the number of parties (i.e., $O(n)$) per multiplication gate in the offline phase — which is independent of the private inputs.

Dishonest majority protocols provide the best security guarantees in terms of collusion sizes since security will be ensured even if all parties but one jointly collude against the remaining honest party. It is known that in this setting public key cryptography tools are needed. In the seminar work of Beaver [Bea92] it was shown how to push most of the "heavy crypto machinery" to an offline phase, hence allowing for a more efficient online phase that can even be information-theoretically secure, or at least use simpler cryptographic tools such as PRGs and hash functions for efficiency. This approach eventually led to the seminal works of BeDOZa [BDOZ11] and SPDZ [DKL⁺13,DPSZ12], which leveraged the Beaver triple technique from [Bea92] together with message authentication codes to achieve a concretely efficient online phase with linear communication complexity in the number of parties per gate. The online phase in SPDZ has been very influential, and there is a large body of research that has focused solely on improving the offline phase, leaving the SPDZ online phase almost intact.

Despite the progress of designing MPC in the dishonest majority setting, it remains unclear whether we can achieve a sub-linear communication complexity in the number of parties per multiplication gate without substantially sacrificing the offline phase[6]. This motivates us to study the following question:

"If a small constant fraction of parties are honest, can we build concretely efficient dishonest majority MPC protocols that achieve constant online communication among all parties per multiplication gate with comparable efficiency as the state-of-the-art in the honest majority setting?"

To be concretely efficient, we refer to protocols that do not rely on heavy Cryptographic tools such as FHE. In particular, we restrict the online phase to be almost information-theoretic except for the black-box use of PRGs or hash functions. Perhaps surprisingly, it is not clear what benefits can be achieved if assume instead of all-but-one corruption, but a constant fraction of parties are honest. In fact, in the case that there are $n - t > 1$ honest parties — unless these constitute a majority — the best one can do to optimize communication is removing $(n - t - 1)$ parties so that, in the new set, there is at least *one* honest party, which is the only requirement for dishonest majority protocols to guarantee security. To the best of our knowledge, the only[7] exception to this is [GPS22], which considers the corruption threshold $t = n(1 - \epsilon)$ for a constant ϵ in the circuit-independent preprocessing model and achieves $58/\epsilon + 96/\epsilon^2$ elements per multiplication gate among all parties in the malicious security setting[8]. Despite the constant communication complexity per multiplication gate achieved in [GPS22], it requires hundreds or even thousands of parties to outperform SPDZ [DPSZ12].

Given the above state-of-affairs, we see that existing dishonest majority protocols are either not very flexible in terms of the amount of corruptions — 50% corruptions are as good as 99%, and having more honest parties do not provide any substantial benefit — or not concretely efficient at all.

1.1 Our Contribution

In this work, we answer the above question affirmatively: we design the first concretely efficient dishonest majority MPC protocol SuperPack that achieves constant online communication among all parties per multiplication gate with comparable efficiency as the state-of-the-art in the honest

[6] An example is [Cou19] which achieves slightly sub-linear communication complexity in the *circuit size* at the cost of increasing the preprocessed data size to be quadratic in the circuit size.

[7] There are also the works of [HOSS18b,HOSS18a], which also explore the benefits of having more than one honest party in the dishonest majority setting. However, their focus is the case of boolean circuits, where they exploit the additional honest parties in order to allow for shorter authentication keys, which in turn reduces the cost of generating these keys. Our work focuses on arithmetic circuits over moderately large fields.

[8] The work [GPS22] does not analyze the concrete cost of their malicious protocol. We obtain this number by counting the amount of communication in their construction. We note that the protocol in [GPS22] also needs to interact for addition gates. Our reported number assumes that the amount of addition gates is the same as the number of multiplication gates.

majority setting [EGPS22]. SuperPack tolerates any number of corruptions and becomes more efficient as the number of honest parties increases, or put differently, it becomes more efficient as the percentage of corrupted parties decreases.

More concretely, we show the following theorem.

Theorem 1 (Informal). *Let n be a positive integer, $\epsilon \in (0, 1/2)$ be a constant, and κ be the security parameter. For an arithmetic circuit C that computes an n-ary functionality $\mathcal{F}$, there exists an n-party protocol that computes C with computational security against a fully malicious adversary who can control at most $t = n(1-\epsilon)$ corrupted parties. The protocol has total communication $O(6|C|n+45|C|/\epsilon)$ elements (ignoring the terms that are independent of the circuit size or only related to the circuit depth[9]) with splitting cost:*

- *Online Phase: $6/\epsilon$ per multiplication gate across all parties.*
- *Circuit-Dependent Preoprocessing Phase: $4/\epsilon$ per multiplication gate across all parties.*
- *Circuit-Independent Preprocessing Phase: $6n + 35/\epsilon$ per multiplication gate across all parties.*

Our construction has the following features:

Online phase (Section 4). The online phase requires *circuit-dependent* preprocessing (meaning, this preprocessing does not depend on the inputs but depends on the topology of the underlying circuit). It relies on information-theoretic tools and as it is typical we also introduce PRGs to further improve efficiency.

Circuit-dependent offline phase (Section 5). The circuit-dependent preprocessing is instantiated using *circuit-independent* preprocessing (meaning, it may depend on the number of certain types of gates of the circuit, but not on its topology) in a simple and efficient manner. Again, the protocol makes use of information-theoretical tools together with PRGs to further improve efficiency.

Circuit-independent offline phase (Section 6). The circuit-independent preprocessing is instantiated by a vector oblivious linear evaluation (VOLE) functionality and an oblivious linear evaluation (OLE) functionality. These two functionalities are realized by protocols in Le Mans [RS22], which can achieve sub-linear communication complexity in the amount of preprocessed data. In addition, we manage to reduce the amount of preprocessed data by a factor of $\epsilon n/2$ compared with that in [RS22]. More discussion can be found in Section 2.

Comparison to Best Previous Works. When comparing with [GPS22], which achieves $58/\epsilon + 96/\epsilon^2$ elements per multiplication gate among all parties in the circuit-independent preprocessing phase, our protocol achieves a factor of at least 25 improvement in the same setting, and a factor of at least 40 improvement in the circuit-dependent preprocessing phase. Since [GPS22] does not realize the circuit-independent preprocessing phase, we do not compare the cost in the circuit-independent preprocessing phase.

Since our goal is to optimize the online phase of dishonest majority protocols where there is a constant fraction of honest parties, we take as a baseline for comparison the existing dishonest majority protocol with the best concrete efficiency in the online phase. This corresponds to the Turbospeedz protocol [BNO19], which is set in the circuit-dependent preprocessing model. To instantiate the preprocessing, we utilize the state-of-the-art [RS22]. Details on this protocol are given in Section E in the Supplementary Material. The resulting protocol has the following communication complexity: $2(1-\epsilon)n$ in the online phase, $4(1-\epsilon)n$ in the circuit-dependent offline phase, and $6(1-\epsilon)n$ in the circuit-independent offline phase when instantiated using Le Mans [RS22] (ignoring the calls to the VOLE and OLE functionalities). Again, the VOLE and OLE functionalities can be properly instantiated with sub-linear communication complexity in the preprocessed data size. And our protocol even reduces this size by a factor of $\epsilon n/2$.

The communication complexity of our protocol and its comparison with respect to Turbospeedz is given in Table 1. We also measure the number of OLE/VOLE correlations required by our protocol and by Turbospeedz. We see that our online phase is better than Turbospeedz by a factor of $(n\epsilon(1-\epsilon))/3$. Some observations about this expression:

[9] The only term that is related to the circuit depth is in the form of $O(n \cdot \texttt{Depth})$. This is because of the use of packed secret sharing which requires to evaluate at least $O(n)$ gates per layer. A similar term also occurs in previous works that use packed secret sharings [DIK10,GIP15,BGJK21,GPS21,GPS22,EGPS22].

- *(Fixing the ratio ϵ)* Given a factor ϵ, meaning there is an $\epsilon \times 100\%$ percentage of honest parties and a $(1-\epsilon) \times 100\%$ percentage of corrupt parties, our online phase is better as long as the total number of parties n is at least the *constant* term $3/(\epsilon(1-\epsilon))$, with the improvement factor increasing as n grows past this threshold. Furthermore, this term goes down as ϵ approaches $1/2$, meaning that the more honest parties/fewer corruptions, the smaller n needs to be for our online phase to be better. For example, if $\epsilon = 0.1$ (90% corruptions) we see improvements with $n \geq 34$; if $\epsilon = 0.2$ (80% corruptions) then $n \geq 19$; and if $\epsilon = 0.3$ (70% corruptions) then $n \geq 15$.
- *(Fixing the number of honest parties)* Given a *fixed* number of *honest* parties h, our online protocol is $\left(\frac{h}{4}\right) \times$ better than prior work *regardless of the total number of parties* n, as long as $n \geq 4h$. This is proven in Section F in the Supplementary Material. This motivates the use of our protocol over prior solutions for any number of parties, as long as minimal support of honest parties can be assumed.

Regarding the complete offline phase, our complexity is $6n + 39/\epsilon$, while in Turbospeedz it is $10(1-\epsilon)n$. This ignores the calls to the OLE and VOLE functionalities, but we remark that this cost is *sublinear* in the size of the circuit when instantiating these with pseudorandom correlator generators, as in [RS22].[10] In the limit as $n \to \infty$, our offline protocol is approximately a factor of $10(1-\epsilon)/6$ times better than Turbospeedz/Le Mans, which ranges between $10/6 \approx 1.6$ for $\epsilon = 0$, to $5/6 \approx 0.83$ for $\epsilon = 1/2$. As a result, *in the limit*, our offline phase is only $1/0.83 = 1.2\times$ less efficient than that of Turbospeedz (and for ϵ close to zero it can be even up to 1.6 better), which is a reasonable cost taking into account the benefits in the online phase. A more thorough discussion on the communication complexity and its implications is given in Section F in the Supplementary Material.

	Online	**CD Offline**	**CI Offline**
SuperPack	$6/\epsilon$	$4/\epsilon$	$6n + 35/\epsilon$
Turbospeedz*	$2(1-\epsilon)n$	$4(1-\epsilon)n$	$6(1-\epsilon)n$

Table 1. Communication complexity in terms of field elements per multiplication gate of SuperPack, and comparison to the previous work with the best concrete efficiency in the online phase, which is Turbospeedz [BNO19] (with its offline phase instantiated by Le Mans [RS22]), referred to as Turbospeedz*. We ignore the calls to $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ and $\mathcal{F}_{\mathsf{nVOLE}}$, which can be instantiated with sublinear communication using PCGs.

Passive-to-active costs. When we strip out SuperPack from the different components needed for active security, like MACs and correctness checks, we obtain a passively secure protocol that also improves over previous works in this setting. Interestingly, the cost of active security is minor: the online and circuit-dependent preprocessing phases do not change between the passive and the active version, and the only overhead appears in the circuit-independent preprocessing phase, where the active protocol requires approximately only a factor of $\times 3$ more communication, on top of calls to a Vector OLE functionality that are needed for authentication.

Implementation and experimental results. Finally, we implement our protocol entirely—except for the OLE/VOLE functionalities as they fall outside the scope of our work—and verify that, experimentally, our protocol outperforms Turbospeedz by the expected amount based on the communication measures when the runtimes are not computation bound. For example, in a 100 Mbps network, our online phase is more than $\approx 4.5\times$ better than that of Turbospeedz for 80 parties, where 60% of them are corrupted. If the network is too fast, then computation becomes a more noticeable bottleneck, and our improvements are less noticeable. This is discussed in Section 7.

[10] Furthermore, our protocol requires fewer OLE/VOLE correlations than previous works, which also makes computation lighter. We expand on this in Remark 1 in page 21.

2 Overview of the Techniques

In this section we provide an overview of our SuperPack protocol. Recall that in our setting we have $t < n(1-\epsilon)$. Let $\mathbb{F}$ be a finite field with $|\mathbb{F}| \geq 2^\kappa$, where κ is the security parameter. We consider packed Shamir secret sharing, where k secrets $\boldsymbol{x} = (x_1, \ldots, x_k)$ are turned into shares as $[\boldsymbol{x}]_d = (f(1), \ldots, f(n))$, where $f(\mathrm{x})$ is a uniformly random polynomial over $\mathbb{F}$ of degree at most d constrained to $f(0) = x_1, \ldots, f(-(k-1)) = x_k$. It also holds that $[\boldsymbol{x}]_{d_1} * [\boldsymbol{y}]_{d_2} = [\boldsymbol{x} * \boldsymbol{y}]_{d_1+d_2}$, where the operator $*$ denotes point-wise multiplication. In our protocol we would like to be able to multiply degree-d sharings by degree-$(k-1)$ sharings (which corresponds to multiplying by constants), so we would like the sum of these degrees to be at most $n-1$ so that the n parties determine the underlying secrets. For this, we take $d + (k-1) = n-1$. On the other hand, we also want the secrets of a degree-d packed Shamir sharing to be private against t corrupted parties, which requires $d \geq t + k - 1$. Together, these imply $n = t + 2(k-1) + 1 = t + 2k - 1$, and $k = \frac{n-t+1}{2} \geq \frac{\epsilon \cdot n+2}{2}$.

At a high level, our technical contributions can be summarized as two aspects:

1. First, we lift the online protocol of TurboPack [EGPS22] from the honest majority setting to the dishonest majority setting. Our starting point is the observation that the passive version of the online protocol from TurboPack [EGPS22] also works for a dishonest majority by setting the parameters correctly. To achieve malicious security, however, the original techniques do not work. This is because in TurboPack, all parties will prepare a degree-t Shamir sharing for each wire value in the circuit. In the honest majority setting, a degree-t Shamir sharing satisfies that the shares of honest parties can fully determine the secret, and the most that malicious parties can do is to change their local shares and cause the whole sharing inconsistent (in the sense that the shares do not lie on a degree-t polynomial). Malicious parties however cannot change the secret by changing their shares. This property unfortunately does not hold in the dishonest majority setting.
   Instead, in our case, we rely on a different type of redundancy widely used in the dishonest majority setting: We make use of message authentication codes, or MACs, to ensure that corrupted parties cannot change the secrets by changing their local shares without being caught. While a similar technique has also been used in [GPS22], their way of using MACs increases the online communication complexity by a factor of at least 2 compared with their passive protocol. We will show how to use MACs in a way such that the online communication complexity remains the same as our passive protocol.
2. Second, we have to reinvent the circuit-independent preprocessing protocol for SuperPack as the corresponding protocol from TurboPack highly relies on the assumption of honest majority, plus that we also need the preprocessed sharings to be authenticated due to the larger corruption threshold.
   The main preprocessing data we need to prepare is referred to as *Packed Beaver Triples*, which are first introduced in [GPS22]. At a high level, a packed Beaver triple contains three packed Shamir sharings $([\boldsymbol{a}], [\boldsymbol{b}], [\boldsymbol{c}])$ such that $\boldsymbol{a}, \boldsymbol{b}$ are random vectors in $\mathbb{F}^k$ and $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$. To prepare such a packed Beaver triple, a direct approach would be first preparing standard Beaver triples using additive sharings and then transform them to packed Shamir sharings. In this way, we may reuse the previous work of generating standard Beaver triples in a black box way. However, this idea requires us to not only pay the cost of preparing standard Beaver triples, but also pay the cost of doing the sharing transformation. The direct consequence is that the overall efficiency of our protocol will be *worse* than that of the state-of-the-art [RS22] in the dishonest majority setting. (And this is the approach used in TurboPack [EGPS22].)
   We will show how to take the advantage of the constant fraction of honest parties in the circuit-independent preprocessing phase by carefully using the techniques of [RS22] in our setting.

In the following, we will start with a sketch of the modified passive version of TurboPack, which is suitable in our setting.

2.1 Starting Point: TurboPack

Our starting point is the observation that the passive version of the online protocol from TurboPack [EGPS22], which is set in the *honest majority* setting, also works for a dishonest majority by setting the parameters correctly. We focus mostly on multiplication gates. So we ignore details regarding input and output gates.

Preprocessing. We consider an arithmetic circuit whose wires are indexed by certain identifiers, which we denote using lowercase Greek letters α, β, γ, etc. Our work is set in the client-server model where there are input and output gates associated to *clients*, who will be in charge of providing input/receiving output. Each multiplication layer of the circuit is split into *batches* of size k. Similarly, each input and output layer assigned to a given client are split into batches of size k. The invariant in TURBOPACK is the following. First, every wire α that is not the output of an addition gate has associated to it a uniformly random value λ_α. If a wire γ is the output of an addition gate with input wires α with wire β, then λ_γ is defined (recursively) as $\lambda_\alpha + \lambda_\beta$.

The parties are assumed to have the following (circuit-dependent) preprocessing material: For every group of k multiplication gates with input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$ and output wires $\boldsymbol{\gamma}$, the parties have $[\boldsymbol{\lambda_\alpha}]_{n-k}$, $[\boldsymbol{\lambda_\beta}]_{n-k}$, and $[\boldsymbol{\lambda_\gamma}]_{n-1}$ (The degree of the last sharing is chosen to be $n-1$ on purpose). In addition, all parties also hold a fresh packed Beaver triple $([\boldsymbol{a}]_{n-k}, [\boldsymbol{b}]_{n-k}, [\boldsymbol{c}]_{n-1})$ for this gate (Again, the degree of the last sharing is chosen to be degree-$(n-1)$ on purpose).

Main Invariant. The main invariant in TURBOPACK is that for every wire α, P_1 knows the value $\mu_\alpha = v_\alpha - \lambda_\alpha$, where v_α denotes the actual value in wire α for a given choice of inputs. Notice that this invariant preserves the privacy of all intermediate wires, since P_1 only learns a masked version of the wire values, and the masks, the λ_α's, are uniformly random and they are kept private with packed Shamir sharings of degree $n-k = t+(k-1)$. We now discuss how, in the original TURBOPACK work, this invariant is maintained throughout the circuit execution. We only focus on (groups of) multiplication gates. Addition gates can be processed locally. Groups of *input* gates with wires $\boldsymbol{\alpha}$ make use of a simple protocol in which the client who owns the gates learns the corresponding masks $\boldsymbol{\lambda_\alpha}$, and sends $\boldsymbol{\mu_\alpha} = \boldsymbol{v_\alpha} - \boldsymbol{\lambda_\alpha}$ to P_1. Groups of output gates are handled in a similar way.

Maintaining the Invariant for Multiplication Gates. Consider a group of multiplication gates in a given circuit level, having input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$, and output wires $\boldsymbol{\gamma}$. Assume that the invariant holds for the input wires, meaning that P_1 knows $\boldsymbol{\mu_\alpha} = \boldsymbol{v_\alpha} - \boldsymbol{\lambda_\alpha}$ and $\boldsymbol{\mu_\beta} = \boldsymbol{v_\beta} - \boldsymbol{\lambda_\beta}$. Recall that the parties have the preprocessed sharings $[\boldsymbol{\lambda_\alpha}]_{n-k}$, $[\boldsymbol{\lambda_\beta}]_{n-k}$, and $[\boldsymbol{\lambda_\gamma}]_{n-1}$. To maintain the invariant, P_1 must learn $\boldsymbol{\mu_\gamma} = \boldsymbol{v_\gamma} - \boldsymbol{\lambda_\gamma}$, where $\boldsymbol{v_\gamma} = \boldsymbol{v_\alpha} * \boldsymbol{v_\beta}$. This is achieved by using the techniques of packed Beaver triples introduced in [GPS22]. Recall that all parties also hold a fresh packed Beaver triple $([\boldsymbol{a}]_{n-k}, [\boldsymbol{b}]_{n-k}, [\boldsymbol{c}]_{n-1})$. All parties proceeds as follows:

1. All parties locally compute the packed Shamir sharing $[\boldsymbol{\lambda_\alpha} - \boldsymbol{a}]_{n-k} = [\boldsymbol{\lambda_\alpha}]_{n-k} - [\boldsymbol{a}]_{n-k}$ and let P_1 learn $\boldsymbol{\lambda_\alpha} - \boldsymbol{a}$. Similar step is done to let P_1 learn $\boldsymbol{\lambda_\beta} - \boldsymbol{b}$.
2. P_1 computes $\boldsymbol{v_\alpha} - \boldsymbol{a} = \boldsymbol{\mu_\alpha} + (\boldsymbol{\lambda_\alpha} - \boldsymbol{a})$ and computes $\boldsymbol{v_\beta} - \boldsymbol{b}$ similarly. Then, P_1 distributes shares $[\boldsymbol{v_\alpha} - \boldsymbol{a}]_{k-1}$ and $[\boldsymbol{v_\beta} - \boldsymbol{b}]_{k-1}$ to the parties.
3. Using the received shares and the shares obtained in the preprocessing phase, the parties compute locally

$$\begin{aligned}[\boldsymbol{v_\gamma}]_{n-1} = [\boldsymbol{v_\alpha} - \boldsymbol{a}]_{k-1} * [\boldsymbol{v_\beta} - \boldsymbol{b}]_{k-1} + [\boldsymbol{v_\alpha} - \boldsymbol{a}]_{k-1} * [\boldsymbol{b}]_{n-k} \\ + [\boldsymbol{v_\beta} - \boldsymbol{b}]_{k-1} * [\boldsymbol{a}]_{n-k} + [\boldsymbol{c}]_{n-1}.\end{aligned}$$

   and $[\boldsymbol{\mu_\gamma}]_{n-1} = [\boldsymbol{v_\gamma}]_{n-1} - [\boldsymbol{\lambda_\gamma}]_{n-1}$.
4. The parties send their shares $[\boldsymbol{\mu_\gamma}]_{n-1}$ to P_1, who reconstructs $\boldsymbol{\mu_\gamma}$. It is easy to see that $\boldsymbol{\mu_\gamma} = \boldsymbol{v_\alpha} * \boldsymbol{v_\beta} - \boldsymbol{\lambda_\gamma}$.

Note that the first step can be completely moved to the circuit-dependent preprocessing phase since both $[\boldsymbol{\lambda_\alpha}]_{n-k}$ and $[\boldsymbol{a}]_{n-k}$ are preprocessed data. With this optimization, the online protocol only requires all parties to communicate $3n$ elements for $k = \epsilon n/2$ multiplication gates, which is $6/\epsilon$ elements per gate among all parties.

2.2 Achieving Active Security

There are multiple places where an active adversary can cheat in the previous protocol, with the most obvious being distributing incorrect (or even invalid) $[\boldsymbol{v_\alpha} - \boldsymbol{a}]_{k-1}$ and $[\boldsymbol{v_\beta} - \boldsymbol{b}]_{k-1}$ at a group of multiplication gates, either by corrupting P_1, or by sending incorrect shares in previous gates to P_1. This is prevented in TURBOPACK by explicitly making use of the honest majority assumption: Using the degree-$(k-1)$ packed Shamir sharings distributed by P_1, the parties will be able to obtain a certain "individual" (*i.e.* non-packed) degree-t Shamir sharing for each wire value. As we discussed above, a degree-t Shamir sharing in the honest majority setting allows honest parties to fully determine the secret. This enables the use of distributed zero-knowledge techniques [BBC+19] to check the correctness of the computation.

In our case where $t \geq n/2$, these techniques cannot be used. Instead, we rely on a different type of redundancy widely used in the dishonest majority setting, namely, we make use of message authentication codes, or MACs, to ensure the parties cannot deviate from the protocol execution when performing actions like reconstructing secret-shared values. We observe that the use of MACs has the following two advantages:

- With MACs, corrupted parties cannot change the secrets of a degree-$(n-k)$ packed Shamir sharing without being detected except with a negligible probability.
- In addition to adding verifiability to packed Shamir sharings, we show how to allow all parties to directly compute MACs of the secret values that are shared by P_1 using degree-$(k-1)$ packed Shamir sharings. This allows us to directly verify whether $\boldsymbol{v_\alpha} - \boldsymbol{a}$ and $\boldsymbol{v_\beta} - \boldsymbol{b}$ are correct without doing distributed zero-knowledge like [EGPS22].

Before we describe our approach, let us introduce some notation. We use $[x|_i]_t$ to denote a Shamir secret sharing of degree t, where the secret is in position $-(i-1)$. I.e., the corresponding polynomial $f(\mathrm{x})$ satisfies that $f(-(i-1)) = x$. We also use $\langle x \rangle$ to denote an additive secret sharing of x. Observe that from a Shamir sharing of x (or a packed Shamir sharing that contains x), all parties can locally obtain an additive sharing of x by locally multiplying suitable Lagrange coefficients.

To achieve active security, we need the parties to hold preprocessing data of the following form:

- Shares of a global random key $\Delta \in \mathbb{F}$ in the form $([\Delta|_1]_t, \ldots, [\Delta|_k]_t)$.
- For every group of k multiplication gates with input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$ and output wires $\boldsymbol{\gamma}$, recall that all parties hold a fresh packed Beaver triple $([\boldsymbol{a}]_{n-k}, [\boldsymbol{b}]_{n-k}, [\boldsymbol{c}]_{n-1})$. They additionally hold $[\Delta \cdot \boldsymbol{a}]_{n-k}$, $[\Delta \cdot \boldsymbol{b}]_{n-k}$, and $\{\langle \Delta \cdot c_i \rangle\}_{i=1}^k$, and also $\{\langle \Delta \cdot \lambda_{\gamma_i} \rangle\}_{i=1}^k$.

With these at hand, the new invariant we maintain to ensure active security is that (1) as before, P_1 learns $\boldsymbol{\mu_\alpha}$ and $\boldsymbol{\lambda_\alpha} - \boldsymbol{a}$ for every group of input wires $\boldsymbol{\alpha}$ of multiplication gates, but in addition (2) the parties have shares $\langle \Delta \cdot \mu_{\alpha_i} \rangle$ and $\langle \Delta \cdot (\lambda_{\alpha_i} - a_i) \rangle$ for all $i \in \{1, \ldots, k\}$. In this way, the first part of the invariant enables the parties to compute the circuit, while the second ensures that P_1 distributed correct values.

Maintaining the New Invariant. Consider a group of multiplication gates with input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$, and output wires $\boldsymbol{\gamma}$. Assume that the invariant holds for the input wires, meaning that P_1 knows $\boldsymbol{\mu_\alpha} = \boldsymbol{v_\alpha} - \boldsymbol{\lambda_\alpha}$ and $\boldsymbol{\mu_\beta} = \boldsymbol{v_\beta} - \boldsymbol{\lambda_\beta}$ as well as $\boldsymbol{\lambda_\alpha} - \boldsymbol{a}$ and $\boldsymbol{\lambda_\beta} - \boldsymbol{b}$, and also the parties have $\{(\langle \Delta \cdot \mu_{\alpha_i} \rangle, \langle \Delta \cdot (\lambda_{\alpha_i} - a_i) \rangle)\}_{i=1}^k$ and $\{(\langle \Delta \cdot \mu_{\beta_i} \rangle, \langle \Delta \cdot (\lambda_{\beta_i} - b_i) \rangle)\}_{i=1}^k$.

The parties preserve the invariant as follows.

- For (1), we follow the passive protocol described above and reconstruct $\boldsymbol{\mu_\gamma}$ to P_1.
- For (2), to be able to compute $\langle \Delta \cdot \mu_{\alpha'_i} \rangle$ for some wire α'_i in the next layer, it is sufficient to let all parties hold $\langle \Delta \cdot \mu_{\gamma_i} \rangle$ for all $i \in \{1, \ldots, k\}$. To this end, we try to follow the procedure of computing $[\boldsymbol{\mu_\gamma}]_{n-1}$. Recall that

$$\begin{aligned}[\boldsymbol{\mu_\gamma}]_{n-1} = [\boldsymbol{v_\alpha} - \boldsymbol{a}]_{k-1} * [\boldsymbol{v_\beta} - \boldsymbol{b}]_{k-1} + [\boldsymbol{v_\alpha} - \boldsymbol{a}]_{k-1} * [\boldsymbol{b}]_{n-k} \\ + [\boldsymbol{v_\beta} - \boldsymbol{b}]_{k-1} * [\boldsymbol{a}]_{n-k} + [\boldsymbol{c}]_{n-1} - [\boldsymbol{\lambda_\gamma}]_{n-1}.\end{aligned}$$

  1. For $[\boldsymbol{v_\alpha} - \boldsymbol{a}]_{k-1} * [\boldsymbol{b}]_{n-k}$ and $[\boldsymbol{v_\beta} - \boldsymbol{b}]_{k-1} * [\boldsymbol{a}]_{n-k}$, since all parties also hold $[\Delta \cdot \boldsymbol{a}]_{n-k}$ and $[\Delta \cdot \boldsymbol{b}]_{n-k}$, they may locally compute $[\boldsymbol{v_\alpha} - \boldsymbol{a}]_{k-1} * [\Delta \cdot \boldsymbol{b}]_{n-k}$ and $[\boldsymbol{v_\beta} - \boldsymbol{b}]_{k-1} * [\Delta \cdot \boldsymbol{a}]_{n-k}$ and convert them locally to $\langle \Delta \cdot (v_{\alpha_i} - a_i) \cdot b_i \rangle$ and $\langle \Delta \cdot (v_{\beta_i} - b_i) \cdot a_i \rangle$.

2. For $[\boldsymbol{c}]_{n-1}$ and $[\boldsymbol{\lambda_\gamma}]_{n-1}$, all parties already hold $\langle \Delta \cdot c_i \rangle$ and $\langle \Delta \cdot \lambda_{\gamma_i} \rangle$.
3. The problematic part is to obtain $\langle \Delta \cdot (v_{\alpha_i} - a_i) \cdot (v_{\beta_i} - b_i) \rangle$. There we use the degree-t Shamir sharing $[\Delta|_i]_t$ as follows. We note that

$$[\Delta \cdot (v_{\alpha_i} - a_i) \cdot (v_{\beta_i} - b_i)|_i]_{n-1} = [\Delta|_i]_t * [\boldsymbol{v_\alpha} - \boldsymbol{a}]_{k-1} * [\boldsymbol{v_\beta} - \boldsymbol{b}]_{k-1}.$$

   This follows from the multiplication of the underlying polynomials and the fact that $n - 1 = t + 2(k-1)$. From $[\Delta \cdot (v_{\alpha_i} - a_i) \cdot (v_{\beta_i} - b_i)|_i]_{n-1}$, all parties can locally compute an additive sharing of $\Delta \cdot (v_{\alpha_i} - a_i) \cdot (v_{\beta_i} - b_i)$.

  Summing all terms up, all parties can *locally* obtain $\langle \Delta \cdot \mu_{\gamma_i} \rangle$.
- For (2), to be able to compute $\langle \Delta \cdot (\lambda_{\alpha'_i} - a'_i) \rangle$ for some wire α'_i in the next layer, it is sufficient to show how to obtain $\langle \Delta \cdot \lambda_{\alpha'_i} \rangle$ since all parties can obtain $\langle \Delta \cdot a'_i \rangle$ from $[\Delta \cdot \boldsymbol{a}']_{n-k}$ prepared in the preprocessing data. Note that all parties already hold $\langle \Delta \cdot \lambda_{\gamma_i} \rangle$ for the current layer. By following the circuit topology, they can locally compute $\langle \Delta \cdot \lambda_{\alpha'_i} \rangle$ for the next layer.

Checking the Correctness of the Computation. All parties together hold additive sharings $\langle \Delta \cdot \mu_{\alpha_i} \rangle$ and $\langle \Delta \cdot (\lambda_{\alpha_i} - a_i) \rangle$, they compute $\langle \Delta \cdot (v_{\alpha_i} - a_i) \rangle$. On the other hand, all parties hold a degree-$(k-1)$ packed Shamir sharing $[\boldsymbol{v_\alpha} - \boldsymbol{a}]_{k-1}$.

It is sufficient to check the following two points:

- The sharing $[\boldsymbol{v_\alpha} - \boldsymbol{a}]_{k-1}$ is a valid degree-$(k-1)$ packed Shamir sharing. I.e., the shares lie on a degree-$(k-1)$ polynomial. The check is done by opening a random linear combination of all degree-$(k-1)$ packed Shamir sharings distributed by P_1.
- The secrets of $[\boldsymbol{v_\alpha} - \boldsymbol{a}]_{k-1}$ are consistent with the MACs $\{\langle \Delta \cdot (v_{\alpha_i} - a_i) \rangle\}_{i=1}^{k}$. This is done by using $[\boldsymbol{v_\alpha} - \boldsymbol{a}]_{k-1}$ and $\{[\Delta|_i]_t\}_{i=1}^{k}$ to compute another version of MACs: $\{\overline{\langle \Delta \cdot (v_{\alpha_i} - a_i) \rangle}\}_{i=1}^{k}$, and then check whether these two versions have the same secrets inside.

Both of these two checks are natural extensions of the checks done in SPDZ [DPSZ12]. We thus omit the details and refer the readers to Section 4.4 for more details.

2.3 Instantiating the Circuit-Dependent Preprocessing

The preprocessing required by the parties is summarized as follows.

- A circuit-independent part, which are the global key $[\Delta|_1]_t, \ldots, [\Delta|_k]_t$ and a fresh packed Beaver triple with authentications per group of multiplication gates $([\boldsymbol{a}]_{n-k}, [\Delta \cdot \boldsymbol{a}]_{n-k}), ([\boldsymbol{b}]_{n-k}, [\Delta \cdot \boldsymbol{b}]_{n-k}), ([\boldsymbol{c}]_{n-1}, \{\langle \Delta \cdot c_i \rangle\}_{i=1}^{k})$.
- A circuit-dependent part that consists of $[\boldsymbol{\lambda_\alpha}]_{n-k}, [\boldsymbol{\lambda_\beta}]_{n-k}, ([\boldsymbol{\lambda_\gamma}]_{n-1}, \{\langle \Delta \cdot \lambda_{\gamma_i} \rangle\}_{i=1}^{k})$. Also P_1 needs to obtain $\boldsymbol{\lambda_\alpha} - \boldsymbol{a}$ and $\boldsymbol{\lambda_\beta} - \boldsymbol{b}$.

For the circuit-independent part, we will focus more on the preparation of the packed Beaver triples with authentications in the next section since the size of $[\Delta|_1]_t, \ldots, [\Delta|_k]_t$ is independent of the circuit size. As for the circuit-dependent part, we essentially follow the same idea in TurboPack [EGPS22] including the preprocessing data we need from a circuit-independent preprocessing, with the only exception that the preprocessing data should be authenticated. We refer the readers to [EGPS22] and Section 5 for more details.

On the Necessity of a Circuit-Dependent Preprocessing. At a first glance, it may appear that if the circuit only contain multiplication gates, then there is no need to have a circuit-dependent preprocessing phase since all λ values are uniform. We stress that this is not the case. This is because each wire α is served as an output wire in a previous layer and then served as an input layer in a next layer. We need all parties to hold two packed Shamir sharings that contain λ_α, one for a previous layer where α is an output wire, and the other one for a next layer where α is an input wire. In particular, the positions of λ_α depend on the circuit topology since we need the two input packed Shamir sharings of a group of multiplication gates to have their secrets correctly aligned.

2.4 Instantiating the Circuit-Independent Preprocessing

Next, we focus on the preparation of authenticated packed Beaver triples:

$$([\boldsymbol{a}]_{n-k}, [\Delta \cdot \boldsymbol{a}]_{n-k}), ([\boldsymbol{b}]_{n-k}, [\Delta \cdot \boldsymbol{b}]_{n-k}), ([\boldsymbol{c}]_{n-1}, \{\langle \Delta \cdot c_i \rangle\}_{i=1}^{k}),$$

where $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$.

To this end, we make use of two functionalities $\mathcal{F}_{\mathsf{nVOLE}}$ and $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ from [RS22]. In [RS22], these two functionalities are used to efficiently prepare Beaver triples using additive sharings. At a high level,

1. All parties first use $\mathcal{F}_{\mathsf{nVOLE}}$ to prepare authenticated random additive sharings. In particular,
   - All parties receive an additive sharing $\langle \Delta \rangle = (\Delta^1, \ldots, \Delta^n)$ from $\mathcal{F}_{\mathsf{nVOLE}}$, where Δ is served as the MAC key. (Here Δ^i is the i-th share of $\langle \Delta \rangle$.)
   - Each party P_i receives a vector $\boldsymbol{u}^i$, which is served as the additive shares held by P_i. We denote the additive sharings by $\langle u_1 \rangle, \ldots, \langle u_m \rangle$.
   - For every ordered pair (P_i, P_j), they together hold an additive sharing of $\boldsymbol{u}^i \cdot \Delta^j$. From these, all parties locally transform them to additive sharings $\langle \Delta \cdot u_1 \rangle, \ldots, \langle \Delta \cdot u_m \rangle$.
2. After using $\mathcal{F}_{\mathsf{nVOLE}}$ to prepare two vectors of additive sharings, say $(\langle a_1 \rangle, \langle b_1 \rangle), \ldots, (\langle a_m \rangle, \langle b_m \rangle)$ together with their MACs, every ordered pair of parties (P_i, P_j) invokes $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ to compute additive sharings of $a_\ell^i \cdot b_\ell^j$ for all $\ell \in \{1, \ldots, m\}$. (Here a_ℓ^i is the i-th share of $\langle a_\ell \rangle$ and b_ℓ^j is the j-th share of $\langle b_\ell \rangle$.) These allow all parties to obtain additive sharings of $\boldsymbol{c} = (a_1 \cdot b_1, \ldots, a_m \cdot b_m)$. Note that the MACs of $\langle c_1 \rangle, \ldots, \langle c_m \rangle$ are *not* computed in this step.
3. Finally, all parties authenticate $\langle c_1 \rangle, \ldots, \langle c_m \rangle$ by using random additive sharings $(\langle r_1 \rangle, \ldots, \langle r_m \rangle)$ with authentications which can be prepared using Step 1.

As we discussed above, one direct solution would be using the above approach in a black box way and then transforming additive sharings to packed Shamir sharings. However, the direct consequence is that we need to not only pay the same cost as that in [RS22], but pay the additional cost for the sharing transformation as well. In the following we discuss how to take the advantage of the constant fraction of honest parties when preparing packed Beaver triples.

Obtaining Authenticated Shares $([\boldsymbol{a}]_{n-k}, [\Delta \cdot \boldsymbol{a}]_{n-k})$. We first discuss how the parties can obtain $[\boldsymbol{a}]_{n-k}$ and $[\Delta \cdot \boldsymbol{a}]_{n-k}$ (and also $[\boldsymbol{b}]_{n-k}$ and $[\Delta \cdot \boldsymbol{b}]_{n-k}$).

Our main observation is that the shares of a random degree-$(n-1)$ packed Shamir sharing are uniformly distributed. This is because a random degree-$(n-1)$ packed Shamir sharing corresponds to a random degree-$(n-1)$ polynomial, which satisfies that any n evaluations are uniformly distributed. On the other hand, the shares of a random additive sharing are also uniformly distributed. Thus, we may naturally view the random additive sharings prepared in $\mathcal{F}_{\mathsf{nVOLE}}$ as degree-$(n-1)$ packed Shamir sharings. Concretely, for each random additive sharing $(u^1, \ldots, u^n)$, let $\boldsymbol{u}$ denote the secrets of the degree-$(n-1)$ packed Shamir sharing when the shares are $(u^1, \ldots, u^n)$. Then we may view that all parties hold the packed Shamir sharing $[\boldsymbol{u}]_{n-1}$. To obtain a degree-$(n-k)$ packed Shamir sharing of $\boldsymbol{u}$, we simply perform a sharing transformation via the standard "mask-open-unmask" approach following from the known techniques [DN07].

Now the problem is to prepare the MACs for $\boldsymbol{u}$. We observe that in $\mathcal{F}_{\mathsf{nVOLE}}$, for every ordered pair of parties (P_i, P_j), P_i, P_j together hold an additive sharing of $u^i \cdot \Delta^j$. Since each secret u_ℓ in $\boldsymbol{u}$ is a linear combination of $(u^1, \ldots, u^n)$, all parties can locally compute an additive sharing of $u_\ell \cdot \Delta^j$ for each $j \in \{1, \ldots, n\}$ and then compute an additive sharing of $\Delta \cdot u_\ell$. To obtain the MACs $[\Delta \cdot \boldsymbol{u}]_{n-k}$, we will perform a sharing transformation again via the standard "mask-open-unmask" approach following from the known techniques [DN07,GPS22].

In this way, to obtain a pair of authenticated sharings $([\boldsymbol{a}]_{n-k}, [\Delta \cdot \boldsymbol{a}]_{n-k})$, we only need to perform once the transformation from additive sharings to packed Shamir sharings. In addition, we essentially obtain such a pair of authenticated sharing from the same data that is only for one authenticated additive sharing in [RS22]. As a result, the amount of preprocessing data we need from $\mathcal{F}_{\mathsf{nVOLE}}$ is reduced by a factor of $k = \epsilon n/2$.

Authenticated Product $([\boldsymbol{c}]_{n-1}, \{\langle \Delta \cdot c_i \rangle\}_{i=1}^k)$. Once the parties have obtained $([\boldsymbol{a}]_{n-k}, [\Delta \cdot \boldsymbol{a}]_{n-k})$ and $([\boldsymbol{b}]_{n-k}, [\Delta \cdot \boldsymbol{b}]_{n-k})$, they need to obtain $([\boldsymbol{c}]_{n-1}, \{\langle \Delta \cdot c_i \rangle\}_{i=1}^k)$, where $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$.

To this end, we need to reuse the degree-$(n-1)$ packed Shamir sharings $[\boldsymbol{a}]_{n-1}$ and $[\boldsymbol{b}]_{n-1}$ output by $\mathcal{F}_{\mathsf{nVOLE}}$. As that in [RS22], every ordered pair of parties (P_i, P_j) invokes $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ to compute additive sharings of $a^i \cdot b^j$, where a^i is the i-th share of $[\boldsymbol{a}]_{n-1}$ and b^j is the j-th share of $[\boldsymbol{b}]_{n-1}$. From additive sharings of $\{a^i \cdot b^j\}_{i,j}$, all parties can locally compute an additive sharing of each $c_\ell = a_\ell \cdot b_\ell$ for all $\ell \in \{1, \ldots, k\}$. Finally, we obtain $[\boldsymbol{c}]_{n-1}$ with authentications by using random sharings $([\boldsymbol{r}]_{n-1}, \{\langle \Delta \cdot r_\ell \rangle\}_{\ell=1}^k)$ and follow the standard "mask-open-unmask" approach. Note that $([\boldsymbol{r}]_{n-1}, \{\langle \Delta \cdot r_\ell \rangle\}_{\ell=1}^k)$ can be directly obtained from $\mathcal{F}_{\mathsf{nVOLE}}$ by properly interpreting the output of $\mathcal{F}_{\mathsf{nVOLE}}$ as we discussed above.

Thus, to prepare the authenticated product $([\boldsymbol{c}]_{n-1}, \{\langle \Delta \cdot c_i \rangle\}_{i=1}^k)$, we only need to perform once the transformation from additive sharings to packed Shamir sharings. Again the amount of preprocessing data we need from $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ is also reduced by a factor of $k = \epsilon n/2$.

Remarks About Our Techniques. Note that we essentially follow the same steps as those in [RS22] but interpreting the output differently, and then perform sharing transformations to obtain sharings in the desired form. We would like to point out that *following the same steps as those in [RS22]* is crucial since in [RS22], $\mathcal{F}_{\mathsf{nVOLE}}$ only outputs random seeds to parties and the parties need to compute their shares by locally expanding the seeds using a proper PRG. And the same seeds are fed in $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ to compute the product sharings. Only in this way together with proper realizations of $\mathcal{F}_{\mathsf{nVOLE}}$ and $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$, [RS22] can achieve sub-linear communication complexity in preparing Beaver triples (without authenticating the product sharing $\langle c \rangle$). Thus, to be able to properly use the functionalities in [RS22], we should follow a similar pattern to that in [RS22].

Verification of Packed Beaver Triples. We note that the packed Beaver triples we obtained may be incorrect. This is because the invocations of $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ are between every pair of parties and the functionality $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ does not force the same party to use the same input across different invocations. Also when the product sharings are authenticated, corrupted parties may introduce additive errors. The same issues also appear in [RS22].

To obtain correct packed Beaver triples with authentications, our idea is to extend the technique of sacrificing [DKL⁺13] and use one possibly incorrect packed Beaver triple to check another possibly incorrect packed Beaver triple. To improve the concrete efficiency, we show that it is sufficient to have the sacrificed packed Beaver triple prepared in the form:

$$([\tilde{\boldsymbol{a}}]_{n-1}, \{\langle \Delta \cdot \tilde{a}_i \rangle\}_{i=1}^k), ([\tilde{\boldsymbol{b}}]_{n-1}, \{\langle \Delta \cdot \tilde{b}_i \rangle\}_{i=1}^k), ([\tilde{\boldsymbol{c}}]_{n-1}, \{\langle \Delta \cdot \tilde{c}_i \rangle\}_{i=1}^k).$$

I.e., we do not need to do any sharing transformation for the first two pairs of sharings and only need to authenticate the product sharing. We refer the readers to Section D.4 for more details.

2.5 Organization of the Document

In Section 3 we present some preliminaries, including basic notation, packed Shamir secret-sharing, a functionalities for coin tossing and commitments. Section 4 presents the online phase of our protocol, which makes use of circuit-dependent preprocessing. Section 5 then discuss how to instantiate this circuit-dependent offline phase, making use of circuit-independent preprocessing. This is followed by Section 6, which shows precisely how to instantiate the circuit-independent offline phase making use of OLE and Vector OLE functionalities, which completes the description of our end-to-end protocol. In Section 7 we discuss our implementation and experimental results.

Supplementary material. Section A includes some extra preliminaries. Section B presents the full online protocol and its security proof. Section C includes the security proof of the protocol for circuit-dependent preprocessing. Section D contains some missing proofs of the circuit-independent preprocessing phase. Section E presents the previous best protocol we use as our point of comparison; this is a variant of the Turbospeedz protocol from [BNO19], adapted to work in three phases like ours (circuit-independent, circuit-dependent and online), and also making use of OLE and VOLE. Section F contains an extra discussion on the communication complexity of our protocol. Section G studies what is the overhead of our actively secure protocol with respect to its passively secure version.

Finally, for easy reference, we include towards the end of the document, in Figure 1 in page 61, a diagram of the different procedures, protocols and functionalities we consider in the document, together with their dependencies.

3 Preliminaries

The Model. We consider the task of secure multiparty computation in the client-server model, where a set of clients $\mathcal{C} = \{C_1, \dots, C_m\}$ provide inputs to a set of computing parties $\mathcal{P} = \{P_1, P_2, \dots, P_n\}$, who carry out the computation and return output to the clients. Clients are connected to parties, and parties are connected to each other using a secure (private and authentic) synchronous channel. The communication complexity is measured by the total number of bits via private channels.

We focus on functions that can be represented as an arithmetic circuit C over a finite field $\mathbb{F}$ with input, addition, multiplication, and output gates.[11] The circuit C takes inputs $(\boldsymbol{x}_1, \dots, \boldsymbol{x}_m)$ and returns $(\boldsymbol{y}_1, \dots, \boldsymbol{y}_m)$, where $\boldsymbol{x}_i \in \mathbb{F}^{I_i}$ and $\boldsymbol{y}_i \in \mathbb{F}^{O_i}$, for $i \in \{1, \dots, m\}$. We use the convention of labeling wires by means of greek letters (*e.g.* α, β, γ), and we use v_α to denote the value stored in a wire labeled by α for a given execution. We use κ to denote the security parameter, and we assume that $|\mathbb{F}| \geq 2^\kappa$. We assume that the number of parties n and the circuit size $|C|$ are bounded by polynomials of the security parameter κ.

We study the dishonest majority setting where the adversary corrupts a majority of the parties, but we focus on the case where the number of corruptions may not be equal to $n-1$. Instead, the adversary corrupts $t < n(1-\epsilon)$ parties for some constant $0 < \epsilon < 1/2$. For security we use Canetti's UC framework [Can01], where security is argued by the indistinguishability of an ideal world, modeled by a *functionality* (denoted in this work by the letter $\mathcal{F}$ and some subscript), and the real world, instantiated by a *protocol* (denoted using the letter Π and some subscript). Protocols can also use *procedures*, denoted using the lowercase letter π and some subscripts, which are like protocols except they are not intended to instantiate a given functionality, and instead, they are used as "macros" inside other protocols that instantiate some functionality. We refer the readers to Section A.1 for the details on the security definition.

We denote by $\mathcal{F}_{\mathsf{MPC}}$ the functionality that receives inputs from the clients, evaluates the function f, and returns output to the clients. This is given in detail in Section A.2 in the Supplementary Material. Security with unanimous abort, where all honest parties may jointly abort in the computation, is the best that can be achieved in the dishonest majority setting. Here we achieve security with selective abort, where the adversary can choose which honest parties abort, which can be compiled to unanimous abort using a broadcast channel [GL05]. To accommodate for aborts, every functionality in this work implicitly allows the adversary to send an abort signal to a specific honest party. We do not write this explicitly.

Packed Shamir Secret Sharing. In our work, we make use of packed Shamir secret sharing, introduced by Franklin and Yung [FY92]. This is a generalization of the standard Shamir secret sharing scheme [Sha79]. Let n be the number of parties and k be the number of secrets to pack in one sharing. A *degree-d* ($d \geq k-1$) packed Shamir sharing of $\boldsymbol{x} = (x_1, \dots, x_k) \in \mathbb{F}^k$ is a vector $(w_1, \dots, w_n)$ for which there exists a polynomial $f(\cdot) \in \mathbb{F}[X]$ of degree at most d such that $f(-i+1) = x_i$ for all $i \in \{1, 2, \dots, k\}$, and $f(i) = w_i$ for all $i \in \{1, 2, \dots, n\}$. The i-th share w_i is held by party P_i. Reconstructing a degree-d packed Shamir sharing requires $d+1$ shares and can be done by Lagrange interpolation. For a random degree-d packed Shamir sharing of $\boldsymbol{x}$, any $d-k+1$ shares are independent of the secret $\boldsymbol{x}$. If $d-(k-1) \geq t$, then knowing t of the shares does not leak anything about the k secrets. In particular, a sharing of degree $t+(k-1)$ keeps hidden the underlying k secret.

[11] In this work, we only focus on deterministic functions. A randomized function can be transformed into a deterministic function by taking as input an additional random tape from each party. The XOR of the input random tapes of all parties is used as the randomness of the randomized function.

In our work, we use $[\boldsymbol{x}]_d$ to denote a degree-d packed Shamir sharing of $\boldsymbol{x} \in \mathbb{F}^k$. In the following, operations (addition and multiplication) between two packed Shamir sharings are coordinate-wise, and $*$ denotes element-wise product. We recall two properties of the packed Shamir sharing scheme:

- **Linear Homomorphism:** For all $d \geq k-1$ and $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}^k$, $[\boldsymbol{x}+\boldsymbol{y}]_d = [\boldsymbol{x}]_d + [\boldsymbol{y}]_d$.
- **Multiplicativity:** Let $*$ denote the coordinate-wise multiplication operation. For all $d_1, d_2 \geq k-1$ subject to $d_1 + d_2 < n$, and for all $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}^k$, $[\boldsymbol{x} * \boldsymbol{y}]_{d_1+d_2} = [\boldsymbol{x}]_{d_1} * [\boldsymbol{y}]_{d_2}$.

Note that the second property implies that, for all $\boldsymbol{x}, \boldsymbol{c} \in \mathbb{F}^k$, all parties can locally compute $[\boldsymbol{c} * \boldsymbol{x}]_{d+k-1}$ from $[\boldsymbol{x}]_d$ and the public vector $\boldsymbol{c}$. To see this, all parties can locally transform $\boldsymbol{c}$ to a degree-$(k-1)$ packed Shamir sharing $[\boldsymbol{c}]_{k-1}$. Then, they can use the property of the packed Shamir sharing scheme to compute $[\boldsymbol{c} * \boldsymbol{x}]_{d+k-1} = [\boldsymbol{c}]_{k-1} * [\boldsymbol{x}]_d$. We simply write $[\boldsymbol{c} * \boldsymbol{x}]_{d+k-1} = \boldsymbol{c} * [\boldsymbol{x}]_d$ to denote this procedure.

When the packing parameter $k = 1$, a packed Shamir sharing degrades to a Shamir sharing. Generically, a Shamir sharing uses the default evaluation point 0 to store the secret. In our work, we are interested in using different evaluation points in different Shamir secret sharings. Concretely, for all $i \in \{1, \dots, k\}$, we use $[x|_i]_d$ to represent a degree-d Shamir sharing of x such that the secret is stored at the evaluation point $-i+1$. If we use f to denote the degree-d polynomial corresponding to $[x|_i]_d$, then $f(-i+1) = x$.

In this work, we choose the packing parameter to be $k = (n-t+1)/2$ (assume for simplicity that this division is exact), or equivalently $n = t + 2k - 1 = t + 2(k-1) + 1$. This implies not only that a sharing of degree $t + (k-1)$ (which keeps the privacy of k secrets) is well defined as there are more parties than the degree plus one, but also if a sharing of such degree is multiplied by a degree-$(k-1)$ sharing, the resulting degree-$(t + 2(k-1))$ sharing is also well defined. Also, we observe that with these parameters, a sharing of degree at most $2(k-1)$ is fully determined by the honest parties' shares since $n - t = 2(k-1) + 1$, which in particular means that such sharings can be reconstructed to obtain the correct underlying secrets (*i.e.* the secrets determined by the honest parties' shares). Finally, recall that $t < n(1-\epsilon)$. We assume that $t + 1 = (1-\epsilon)n$ for simplicity, and in this case, it can be checked that $k = \frac{\epsilon}{2} \cdot n + 1 = \Theta(n)$.

Some Functionalities. For our protocols we assume the existence of two widely used functionalities. One is $\mathcal{F}_{\mathsf{Coin}}$, which upon being called provides the parties with a uniformly random value $r \in \mathbb{F}$. This can be easily implemented by having the parties open some random shared value $\langle r \rangle$, and if more coins are needed these can be expanded with the help of a PRG. The second functionality is $\mathcal{F}_{\mathsf{Commit}}$, which enables a party to commit to some values of their choice towards a given set of receivers, without revealing this value. At a later point, the party can open their committed values with the guarantee that these opened terms are exactly the same as the ones committed to initially. This can be instantiated with the help of a hash function modeled as a random oracle H (*cf.* [DKL+13]): a party commits to m towards a set of receivers by sampling $r \in \{0,1\}^\sigma$ and sending $c = \mathsf{H}(m, r) \in \{0,1\}^\sigma$, and later on this party opens m by sending the pair (m, r) to these receivers, who check that $c = \mathsf{H}(m, r)$. The communication complexity of committing a message m towards one receiver is σ bits, and the communication complexity of opening the commitment of a message m is $|m| + \sigma$ bits. Here, σ is a computational security parameter, which can be taken in practice to be 128.

4 Online Protocol

We begin by describing the online phase of SuperPack.

4.1 Circuit-Dependent Preprocessing Functionality

In order to securely compute the given function, our online phase must make use of certain *circuit-dependent* preprocessing, which is modeled in Functionality $\mathcal{F}_{\mathsf{PrepMal}}$ below.

Functionality 1: $\mathcal{F}_{\mathsf{PrepMal}}$

1. **Assign Random Values to Wires in C**: $\mathcal{F}_{\mathsf{PrepMal}}$ receives the circuit C from all parties. Then $\mathcal{F}_{\mathsf{PrepMal}}$ assigns random values to wires in C as follows.
   (a) For each output wire α of an input gate or a multiplication gate, $\mathcal{F}_{\mathsf{PrepMal}}$ samples a uniform value λ_α and associates it with the wire α.
   (b) Starting from the first layer of C to the last layer, for each addition gate with input wires α, β and output wire γ, $\mathcal{F}_{\mathsf{PrepMal}}$ sets $\lambda_\gamma = \lambda_\alpha + \lambda_\beta$.
2. **Settling Authentication Keys**: $\mathcal{F}_{\mathsf{PrepMal}}$ samples a random value Δ. Then $\mathcal{F}_{\mathsf{PrepMal}}$ samples k random degree-t Shamir sharings $([\Delta|_1]_t, \ldots, [\Delta|_k]_t)$ and distributes the shares to all parties.
3. **Preparing Packed Beaver Triples with Authentications**: For each group of k multiplication gates, $\mathcal{F}_{\mathsf{PrepMal}}$ samples a random packed Beaver triple with authentications as follows:
   (a) $\mathcal{F}_{\mathsf{PrepMal}}$ samples two random vectors $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{F}_p^k$ and computes $\Delta \cdot \boldsymbol{a}, \Delta \cdot \boldsymbol{b}$. Then $\mathcal{F}_{\mathsf{PrepMal}}$ samples two pairs of random degree-$(n-k)$ packed Shamir sharings $[\![\boldsymbol{a}]\!]_{n-k} = ([\boldsymbol{a}]_{n-k}, [\Delta \cdot \boldsymbol{a}]_{n-k}), [\![\boldsymbol{b}]\!]_{n-k} = ([\boldsymbol{b}]_{n-k}, [\Delta \cdot \boldsymbol{b}]_{n-k})$.
   (b) $\mathcal{F}_{\mathsf{PrepMal}}$ computes $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$ and $\Delta \cdot \boldsymbol{c}$. Then $\mathcal{F}_{\mathsf{PrepMal}}$ samples a random degree-$(n-1)$ packed Shamir sharing $[\boldsymbol{c}]_{n-1}$. For all $i \in \{1, \ldots, k\}$, $\mathcal{F}_{\mathsf{PrepMal}}$ samples a random additive sharing $\langle \Delta \cdot c_i \rangle$.

   $\mathcal{F}_{\mathsf{PrepMal}}$ distributes the shares of $([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, ([\boldsymbol{c}]_{n-1}, \{\langle \Delta \cdot c_i \rangle\}_{i=1}^k))$ to all parties.
4. **Distributing $\boldsymbol{\lambda_\alpha} - \boldsymbol{a}$ and $\boldsymbol{\lambda_\beta} - \boldsymbol{b}$ to P_1**: For each group of multiplication gates, let $\boldsymbol{\alpha}, \boldsymbol{\beta}$ denote the batch of first input wires and that of the second input wires respectively. Let $([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, ([\boldsymbol{c}]_{n-1}, \{\langle \Delta \cdot c_i \rangle\}_{i=1}^k)$ be the packed Beaver triple with authentications associated with these gates.
   $\mathcal{F}_{\mathsf{PrepMal}}$ receives two vectors of additive errors $\boldsymbol{\delta_\alpha}, \boldsymbol{\delta_\beta}$ from the adversary, computes $\boldsymbol{\lambda_\alpha} - \boldsymbol{a} + \boldsymbol{\delta_\alpha}$ and $\boldsymbol{\lambda_\beta} - \boldsymbol{b} + \boldsymbol{\delta_\beta}$, and sends them to P_1. Here $\boldsymbol{\lambda_\alpha}$ and $\boldsymbol{\lambda_\beta}$ are the random values associated with the wires $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$.
   $\mathcal{F}_{\mathsf{PrepMal}}$ also samples random additive sharings $\{\langle \Delta \cdot (\lambda_{\alpha_i} - a_i) \rangle, \langle \Delta \cdot (\lambda_{\beta_i} - b_i) \rangle\}_{i=1}^k$ and distributes the shares to all parties.
5. **Preparing Authenticated Packed Sharings for Multiplication Gates**: For each group of multiplication gates with output wires $\boldsymbol{\gamma}$, $\mathcal{F}_{\mathsf{PrepMal}}$ samples
   – A random degree-$(n-1)$ packed Shamir sharing $[\boldsymbol{\lambda_\gamma}]_{n-1}$,
   – k additive sharings $\{\langle \Delta \cdot \lambda_{\gamma_i} \rangle\}_{i=1}^k$,

   and distributes the shares to honest parties.
6. **Preparing Random Sharings for Input and Output Gates**: For each group of k input gates or output gates, $\mathcal{F}_{\mathsf{PrepMal}}$ prepares the following random sharings.
   (a) Let $\boldsymbol{\alpha}$ be the output wires of these k input gates or the input wires of these k output gates. $\mathcal{F}_{\mathsf{PrepMal}}$ samples
      – A random degree-$(n-1)$ packed Shamir sharing $[\boldsymbol{\lambda_\alpha}]_{n-1}$,
      – k additive sharings $\{\langle \Delta \cdot \lambda_{\alpha_i} \rangle\}_{i=1}^k$,

      and distributes the shares to honest parties.
   (b) $\mathcal{F}_{\mathsf{PrepMal}}$ also prepares a random packed Beaver triple with authentications $([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, ([\boldsymbol{c}]_{n-1}, \{\langle \Delta \cdot c_i \rangle\}_{i=1}^k)$ in the same way as Step 3. Later, we will view $\boldsymbol{b}$ as an authentication key and $\boldsymbol{c}$ as the MAC of $\boldsymbol{a}$. This allows the input holder to verify the correctness of $\boldsymbol{a}$.

Corrupted Parties: When $\mathcal{F}_{\mathsf{PrepMal}}$ prepares random sharings, corrupted parties can choose their shares. $\mathcal{F}_{\mathsf{PrepMal}}$ then samples the random sharings based on the secret it generated and the shares chosen by the corrupted parties.

4.2 Input Gates

In this section, we give the description of the procedure π_{Input}. This procedure enables P_1 to learn $\mu_\alpha = v_\alpha - \lambda_\alpha$ for every input wire α, where v_α is the input provided by the client owning the input gate. In addition, the parties output shares of the MAC of this value, namely $\{\langle \Delta \cdot \mu_{\alpha_i} \rangle\}_{i=1}^k$. In a bit more detail, recall that in $\mathcal{F}_{\mathsf{PrepMal}}$, all parties have prepared the following random sharings for each group of input gates:

– A random degree-$(n-1)$ packed Shamir sharing $[\boldsymbol{\lambda_\alpha}]_{n-1}$ with MACs $\{\langle \Delta \cdot \lambda_{\alpha_i} \rangle\}_{i=1}^k$.
– A packed Beaver triple with authentications

$$([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, ([\boldsymbol{c}]_{n-1}, \{\langle \Delta \cdot c_i \rangle\}_{i=1}^k).$$

The client receives the shares of $[\boldsymbol{\lambda}_{\boldsymbol{\alpha}}]_{n-1}$ and $[\boldsymbol{a}]_{n-k}$ from all parties and reconstructs the secrets $\boldsymbol{\lambda}_{\boldsymbol{\alpha}}, \boldsymbol{a}$. To verify the correctness of $\boldsymbol{a}$, all parties also send their shares of $[\boldsymbol{b}]_{n-k}, [\boldsymbol{c}]_{n-1}$ to the client so that $\boldsymbol{b}$ serves as the MAC key and $\boldsymbol{c}$ serves as the MAC of $\boldsymbol{a}$. Then the client sends $\boldsymbol{x}-\boldsymbol{a}$ to all parties, which allows all parties to obtain a packed Shamir sharing of $\boldsymbol{x}$ together with the MAC. The client also sends $\boldsymbol{\mu}_{\boldsymbol{\alpha}} = \boldsymbol{x} - \boldsymbol{\lambda}_{\boldsymbol{\alpha}}$ to P_1. Note that P_1 may receive incorrect $\boldsymbol{\mu}_{\boldsymbol{\alpha}}$ (either because an honest client reconstructs an incorrect $\boldsymbol{\lambda}_{\boldsymbol{\alpha}}$ or because a corrupted client sends incorrect $\boldsymbol{\mu}_{\boldsymbol{\alpha}}$ to $P-1$). However, all parties can locally compute the MAC of $\boldsymbol{\mu}_{\boldsymbol{\alpha}}$ which allows us to verify the computation at the end of the protocol. The description of π_{Input} appears below.

Procedure 1: π_{Input}

1. For each group of input gates that belongs to `Client`, let $\boldsymbol{\alpha}$ denote the batch of output wires of these input gates. All parties receive from $\mathcal{F}_{\mathsf{PrepMal}}$
   – A random degree-$(n-1)$ packed Shamir sharing $[\boldsymbol{\lambda}_{\boldsymbol{\alpha}}]_{n-1}$ with MACs $\{\langle \Delta \cdot \lambda_{\alpha_i}\rangle\}_{i=1}^k$.
   – A packed Beaver triple with authentications
   $$([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, ([\boldsymbol{c}]_{n-1}, \{\langle \Delta \cdot c_i\rangle\}_{i=1}^k).$$
   Let $\boldsymbol{v}_{\boldsymbol{\alpha}}$ denote the inputs held by `Client`.
2. All parties send to `Client` their shares of $[\boldsymbol{\lambda}_{\boldsymbol{\alpha}}]_{n-1}, [\boldsymbol{a}]_{n-k}, [\boldsymbol{b}]_{n-k}, [\boldsymbol{c}]_{n-1}$.
3. `Client` reconstructs the secrets $\boldsymbol{\lambda}_{\boldsymbol{\alpha}}, \boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}$ and checks whether $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$. If not, `Client` aborts. Otherwise, `Client` computes $\boldsymbol{\mu}_{\boldsymbol{\alpha}} = \boldsymbol{v}_{\boldsymbol{\alpha}} - \boldsymbol{\lambda}_{\boldsymbol{\alpha}}$ and $[\boldsymbol{v}_{\boldsymbol{\alpha}} - \boldsymbol{a}]_{2k-2}$.
4. `Client` sends $\boldsymbol{\mu}_{\boldsymbol{\alpha}}$ to P_1 and distributes the shares of $[\boldsymbol{v}_{\boldsymbol{\alpha}} - \boldsymbol{a}]_{2k-2}$ to all parties.
5. For all $i \in \{1, \dots, k\}$, all parties locally compute $\langle \Delta \cdot \mu_{\alpha_i}\rangle$ as follows:
   (a) Recall that all parties hold $[\Delta|_i]_t$ generated in $\mathcal{F}_{\mathsf{PrepMal}}$. All parties locally compute $[\Delta \cdot (v_{\alpha_i} - a_i)|_i]_{n-1} = [\Delta|_i]_t * [\boldsymbol{v}_{\boldsymbol{\alpha}} - \boldsymbol{a}]_{2k-2}$. Then all parties locally transform it to an additive sharing $\langle \Delta \cdot (v_{\alpha_i} - a_i)\rangle$.
   (b) Recall that all parties hold $[\Delta \cdot \boldsymbol{a}]_{n-k}$. All parties locally transform it to an additive sharing $\langle \Delta \cdot a_i\rangle$.
   (c) Recall that all parties hold $\langle \Delta \cdot \lambda_{\alpha_i}\rangle$. All parties locally compute
   $$\langle \Delta \cdot \mu_{\alpha_i}\rangle = \langle \Delta \cdot (v_{\alpha_i} - a_i)\rangle + \langle \Delta \cdot a_i\rangle - \langle \Delta \cdot \lambda_{\alpha_i}\rangle.$$

Communication complexity of π_{Input}. The communication complexity per group of k inputs gates consists of:

– (Step 2) $2n + 2(n-k+1) = (4-\epsilon)\cdot n$ shares from the parties to the client
– (Step 4) $k = \frac{\epsilon}{2}\cdot n + 1$ elements from the client to P_1
– (Step 4) $n - (k-1) - 1 = (1 - \frac{\epsilon}{2})n - 1$ elements from the client to the parties.

This leads to a total of $(5-\epsilon)\cdot n$ elements per group of k input gates. Therefore, per input gate, this becomes $\frac{(5-\epsilon)n}{k} = \frac{2(5-\epsilon)n}{\epsilon\cdot n+2} \le \frac{10}{\epsilon}$.

4.3 Computing Addition and Multiplication Gates

After receiving the inputs from all clients, all parties start to evaluate the circuit gate by gate. We will maintain the invariant that for each output wire α of an input gate or a multiplication gate, P_1 learns μ_α in clear. In the procedure, P_1 distributes shares of certain values, which may be incorrect. To prevent cheating, the parties get additive shares of the MAC of these values, which are used in a verification step in the output phase to check for correctness.

The evaluation proceeds as follows:

– For each addition gate with input wires α, β and output wire γ, P_1 locally computes $\mu_\gamma = \mu_\alpha + \mu_\beta$, and the parties locally compute $\langle \Delta \cdot \mu_\gamma\rangle = \langle \Delta \cdot \mu_\alpha\rangle + \langle \Delta \cdot \mu_\beta\rangle$.
– For each group of k multiplication gates, all parties run the procedure π_{Mult} to allow P_1 to learn $\boldsymbol{\mu}_{\boldsymbol{\gamma}}$. Note that, all parties can compute the MACs of $\boldsymbol{v}_{\boldsymbol{\alpha}} - \boldsymbol{a}$ and $\boldsymbol{v}_{\boldsymbol{\beta}} - \boldsymbol{b}$, which allows us to check whether P_1 shares correct values to all parties. Therefore in π_{Mult}, all parties further compute $\{\langle \theta_{\alpha_i}\rangle, \langle \theta_{\beta_i}\rangle\}_{i=1}^k$, which should be additive sharings of 0 if P_1 shares correct values. We will check these additive sharings at the end of the computation.

Procedure 2: π_{Mult}

The procedure is executed for a group of k multiplication gates with input wires $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$, and output wires $\boldsymbol{\gamma}$.

1. All parties hold
   - A packed Beaver triple with authentications
   $$([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, ([\boldsymbol{c}]_{n-1}, \{\langle \Delta \cdot c_i \rangle\}_{i=1}^{k}).$$
   - A random degree-$(n-1)$ packed Shamir sharing $[\boldsymbol{\lambda_\gamma}]_{n-1}$ with MACs $\{\langle \Delta \cdot \lambda_{\gamma_i} \rangle\}_{i=1}^{k}$.
   - Additive sharings $\{\langle \Delta \cdot (\lambda_{\alpha_i} - a_i) \rangle, \langle \Delta \cdot (\lambda_{\beta_i} - b_i) \rangle\}_{i=1}^{k}$.

   And P_1 learns
   - $\boldsymbol{\mu_\alpha} = \boldsymbol{v_\alpha} - \boldsymbol{\lambda_\alpha}$, $\boldsymbol{\mu_\beta} = \boldsymbol{v_\beta} - \boldsymbol{\lambda_\beta}$ from the previous layers;
   - $\boldsymbol{\lambda_\alpha} - \boldsymbol{a}$, $\boldsymbol{\lambda_\beta} - \boldsymbol{b}$ received from $\mathcal{F}_{\mathsf{PrepMal}}$.
2. P_1 locally computes $\boldsymbol{v_\alpha} - \boldsymbol{a} = \boldsymbol{\mu_\alpha} + \boldsymbol{\lambda_\alpha} - \boldsymbol{a}$. Similarly, P_1 locally computes $\boldsymbol{v_\beta} - \boldsymbol{b}$. Then P_1 distributes shares of $[\boldsymbol{v_\alpha} - \boldsymbol{a}]_{k-1}$ and $[\boldsymbol{v_\beta} - \boldsymbol{b}]_{k-1}$ to all parties.
3. For all $i \in \{1, \ldots, k\}$, all parties locally compute $\langle \theta_{\alpha_i} \rangle$ and $\langle \theta_{\beta_i} \rangle$ as follows.
   (a) Recall that all parties have computed additive sharings of the MACs of the μ values for output wires of multiplication gates and input gates in previous layers. By using these additive sharings, all parties locally compute $\langle \Delta \cdot \mu_{\alpha_i} \rangle$, $\langle \Delta \cdot \mu_{\beta_i} \rangle$.
   (b) Recall that all parties hold $\langle \Delta \cdot (\lambda_{\alpha_i} - a_i) \rangle, \langle \Delta \cdot (\lambda_{\beta_i} - b_i) \rangle$. They locally compute
   $$\langle \Delta \cdot (v_{\alpha_i} - a_i) \rangle = \langle \Delta \cdot \mu_{\alpha_i} \rangle + \langle \Delta \cdot (\lambda_{\alpha_i} - a_i) \rangle,$$
   $$\langle \Delta \cdot (v_{\beta_i} - b_i) \rangle = \langle \Delta \cdot \mu_{\beta_i} \rangle + \langle \Delta \cdot (\lambda_{\beta_i} - b_i) \rangle.$$
   (c) Also recall that all parties hold $[\Delta|_i]_t$. All parties locally compute $[\Delta|_i]_t * [\boldsymbol{v_\alpha} - \boldsymbol{a}]_{k-1}$ and transform it to an additive sharing $\langle \overline{\Delta \cdot (v_{\alpha_i} - a_i)} \rangle$. Similarly, all parties locally compute $[\Delta|_i]_t * [\boldsymbol{v_\beta} - \boldsymbol{b}]_{k-1}$ and transform it to an additive sharing $\langle \overline{\Delta \cdot (v_{\beta_i} - b_i)} \rangle$.
   (d) All parties locally compute $\langle \theta_{\alpha_i} \rangle = \langle \Delta \cdot (v_{\alpha_i} - a_i) \rangle - \langle \overline{\Delta \cdot (v_{\alpha_i} - a_i)} \rangle$ and $\langle \theta_{\beta_i} \rangle = \langle \Delta \cdot (v_{\beta_i} - b_i) \rangle - \langle \overline{\Delta \cdot (v_{\beta_i} - b_i)} \rangle$.
4. All parties locally compute
$$\begin{aligned}[\boldsymbol{\mu_\gamma}]_{n-1} = {} & [\boldsymbol{v_\alpha} - \boldsymbol{a}]_{k-1} * [\boldsymbol{v_\beta} - \boldsymbol{b}]_{k-1} + [\boldsymbol{v_\alpha} - \boldsymbol{a}]_{k-1} * [\boldsymbol{b}]_{n-k} \\ & + [\boldsymbol{v_\beta} - \boldsymbol{b}]_{k-1} * [\boldsymbol{a}]_{n-k} + [\boldsymbol{c}]_{n-1} - [\boldsymbol{\lambda_\gamma}]_{n-1}.\end{aligned}$$
5. For all $i \in \{1, \ldots, k\}$, all parties locally compute an additive sharing $\langle \Delta \cdot \mu_{\gamma_i} \rangle$ as follows.
   (a) Recall that all parties hold $[\Delta|_i]_t$ from $\mathcal{F}_{\mathsf{PrepMal}}$. All parties locally compute $[\Delta|_i]_t * [\boldsymbol{v_\alpha} - \boldsymbol{a}]_{k-1} * [\boldsymbol{v_\beta} - \boldsymbol{b}]_{k-1}$ and transform it to an additive sharing $\langle \Delta \cdot (v_{\alpha_i} - a_i) \cdot (v_{\beta_i} - b_i) \rangle$.
   (b) Recall that all parties hold $[\Delta \cdot \boldsymbol{a}]_{n-k}$ and $[\Delta \cdot \boldsymbol{b}]_{n-k}$. All parties locally compute $[\boldsymbol{v_\alpha} - \boldsymbol{a}]_{k-1} * [\Delta \cdot \boldsymbol{b}]_{n-k} + [\boldsymbol{v_\beta} - \boldsymbol{b}]_{k-1} * [\Delta \cdot \boldsymbol{a}]_{n-k}$ and transform it to an additive sharing $\langle \Delta \cdot ((v_{\alpha_i} - a_i) \cdot b_i + (v_{\beta_i} - b_i) \cdot a_i) \rangle$.
   (c) Recall that all parties hold $\langle \Delta \cdot c_i \rangle$ and $\langle \Delta \cdot \lambda_{\gamma_i} \rangle$. All parties locally compute
   $$\begin{aligned}\langle \Delta \cdot \mu_{\gamma_i} \rangle = {} & \langle \Delta \cdot (v_{\alpha_i} - a_i) \cdot (v_{\beta_i} - b_i) \rangle \\ & + \langle \Delta \cdot ((v_{\alpha_i} - a_i) \cdot b_i + (v_{\beta_i} - b_i) \cdot a_i) \rangle \\ & + \langle \Delta \cdot c_i \rangle - \langle \Delta \cdot \lambda_{\gamma_i} \rangle.\end{aligned}$$
6. P_1 collects the whole sharing $[\boldsymbol{\mu_\gamma}]_{n-1}$ from all parties and reconstructs $\boldsymbol{\mu_\gamma}$.

Communication complexity of π_{Mult}. The communication complexity per group of k multiplication gates consists of:

- (Step 2) $2(n-1)$ shares from P_1 to the parties
- (Step 6) $n-1$ shares from the parties to P_1.

This leads to a total of $3n-3$ elements per group of $k = \frac{\epsilon}{2} \cdot n + 1$ input gates. Therefore, per multiplication gate, this becomes $\frac{3n-3}{k} = \frac{2(3n-3)}{\epsilon \cdot n + 2} \leq \frac{6}{\epsilon}$.

4.4 Output Gates and Verification

At the end of the protocol, all parties together check the correctness of the computation. We first transform the output sharings to sharings that can be conveniently checked by clients. However, before reconstructing these outputs to the clients, the parties jointly verify the correctness of the computation by checking that (1) the sharings distributed by P_1 in π_{Mult} have the correct degree $\leq k-1$, and (2) the underlying secrets are correct, for which the MACs computed in the online phase are used.

In a bit more detail, first recall that for each group of k output gates, all parties have prepared the following random sharings:

– A random degree-$(n-1)$ packed Shamir sharing $[\boldsymbol{\lambda_\alpha}]_{n-1}$ with MACs $\{\langle \Delta \cdot \lambda_{\alpha_i} \rangle\}_{i=1}^{k}$.
– A packed Beaver triple with authentications

$$(\llbracket \boldsymbol{a} \rrbracket_{n-k}, \llbracket \boldsymbol{b} \rrbracket_{n-k}, ([\boldsymbol{c}]_{n-1}, \{\langle \Delta \cdot c_i \rangle\}_{i=1}^{k})).$$

In addition, P_1 learns $\boldsymbol{\mu_\alpha}$ in clear. Similarly to input gates, we shall use $\boldsymbol{b}$ as the MAC key and $\boldsymbol{c}$ as the MAC of $\boldsymbol{a}$. This allows the client to check the correctness of $\boldsymbol{a}$. Thus, our first step is to allow all parties obtain a degree-$(2k-2)$ packed Shamir sharing of $\boldsymbol{v_\alpha} - \boldsymbol{a}$. Note that the secrets of a degree-$(2k-2)$ packed Shamir sharing are fully determined by the shares of honest parties: corrupted parties cannot change the secrets without being detected.

To this end, all parties locally compute $[\boldsymbol{\lambda_\alpha} - \boldsymbol{a}]_{n-1} = [\boldsymbol{\lambda_\alpha}]_{n-1} - [\boldsymbol{a}]_{n-k}$ and send their shares to P_1. P_1 reconstructs $\boldsymbol{\lambda_\alpha} - \boldsymbol{a}$ and shares $\boldsymbol{v_\alpha} - \boldsymbol{a} = \boldsymbol{\mu_\alpha} + \boldsymbol{\lambda_\alpha} - \boldsymbol{a}$ using a degree-$(2k-2)$ packed Shamir sharing. Following the same idea in π_{Mult}, all parties compute $\{\langle \theta_{\alpha_i} \rangle\}_{i=1}^{k}$ and check whether they are additive sharings of 0 later.

Before reconstructing the outputs to clients, we need to check the correctness of the computation. It is sufficient to ensure that

– For each group of multiplication gates, P_1 distributes correct degree-$(k-1)$ packed Shamir sharings $[\boldsymbol{v_\alpha} + \boldsymbol{a}]_{k-1}$ and $[\boldsymbol{v_\beta} + \boldsymbol{b}]_{k-1}$ to all parties. This includes checking the degree of each sharing is correct and checking the secrets are correct (with respect to the MACs).
  We will achieve the verification by checking that the degree-$(k-1)$ packed Shamir sharings distributed by P_1 in π_{Mult} are valid and checking that all additive sharings in the form of $\langle \theta_\alpha \rangle$ computed in π_{Mult} are additive sharings of 0.
– For each group of output gates, P_1 distributes correct degree-$(2k-2)$ packed Shamir sharings $[\boldsymbol{v_\alpha} + \boldsymbol{a}]_{2k-2}$ to all parties. This time, we only need to check the secrets are correct (with respect to the MACs) as the shares of honest parties always form a valid degree-$(2k-2)$ packed Shamir sharing.
  We will achieve the verification by checking that all additive sharings in the form of $\langle \theta_\alpha \rangle$ computed when transforming the output sharings are additive sharings of 0.

Due to space constraints, we describe the procedure π_{Output} in detail in Section B.1 in the Supplementary Material, including the computation of the output gates, the verification of the computation (degree and MAC check), and the reconstruction of the outputs.

4.5 Full Online Protocol

Our final online protocol makes use of the procedures π_{Input} (Proc. 1, Section 4.2) to let the clients distribute their inputs, π_{Mult} (Proc. 2, Section 4.3) to process each group of k multiplication gates, and π_{Output} (Proc. 4, Section B.1) to verify the correctness of the computation and reconstruct output to the clients. The online protocol Π_{Online} is presented in full detail in Section B.2 in the Supplementary Material. We prove the following:

Theorem 2. *Let c denote the number of servers and n denote the number of parties (servers). For all $0 < \epsilon \leq 1/2$, protocol Π_{Online} instantiates Functionality $\mathcal{F}_{\mathsf{MPC}}$ in the $\mathcal{F}_{\mathsf{PrepMal}}$-hybrid model, with statistical security against a fully malicious adversary who can control up to c clients and $t = (1-\epsilon)n$ parties (servers).*

Communication complexity of Π_{Online}. Let I and O be the number of input wires and output wires, and assume that each client owns a number of input and output gates that is a multiple of k. We assume for simplicity that n divides each of these terms, and also that n divides the number of multiplication gates in each layer. Let us also denote by $|C|$ the number of multiplication gates in the circuit C. The total communication complexity is given by $\frac{10}{\epsilon} \cdot I + \frac{24}{\epsilon} \cdot O + \frac{6}{\epsilon} \cdot |C|$, ignoring small terms that are independent of I, O and $|C|$.

5 Circuit-Dependent Preprocessing Phase

In this section, we discuss how to realize the ideal functionality for the circuit-dependent preprocessing phase, $\mathcal{F}_{\text{PrepMal}}$, presented as Functionality 1. Recall that $k = (n - t + 1)/2$. For simplicity, we only focus on the scenario where $t \geq n/2$.

We realize $\mathcal{F}_{\text{PrepMal}}$ by using a circuit-independent functionality, $\mathcal{F}_{\text{PrepIndMal}}$, which is described below.

Functionality 2: $\mathcal{F}_{\text{PrepIndMal}}$

1. **Setting Authentication Keys**: $\mathcal{F}_{\text{PrepIndMal}}$ samples a random value Δ. Then $\mathcal{F}_{\text{PrepIndMal}}$ samples k random degree-t Shamir sharings $([\Delta|_1]_t, \dots, [\Delta|_k]_t)$ and distributes the shares to all parties.
2. **Preparing Random Packed Sharings**: For each output wire α of an input gate or a multiplication gate in the circuit C, $\mathcal{F}_{\text{PrepIndMal}}$ samples a random value as λ_α and computes $\lambda_\alpha \cdot \mathbf{1}$, where $\mathbf{1} = (1, \dots, 1) \in \mathbb{F}^k$. Then $\mathcal{F}_{\text{PrepIndMal}}$ samples
   - a random degree-$(n-k)$ packed Shamir sharing $[\lambda_\alpha \cdot \mathbf{1}]_{n-k}$,
   - and a random additive sharing $\langle \Delta \cdot \lambda_\alpha \rangle$,

   and distributes the shares to all parties.
3. **Preparing Packed Beaver Triples with Authentications**: For each group of k multiplication gates, $\mathcal{F}_{\text{PrepIndMal}}$ samples a random packed Beaver triple with authentications as follows:
   (a) $\mathcal{F}_{\text{PrepIndMal}}$ samples two random vectors $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{F}_p^k$ and computes $\Delta \cdot \boldsymbol{a}, \Delta \cdot \boldsymbol{b}$. Then $\mathcal{F}_{\text{PrepIndMal}}$ samples two pairs of random degree-$(n-k)$ packed Shamir sharings $[\![\boldsymbol{a}]\!]_{n-k} = ([\boldsymbol{a}]_{n-k}, [\Delta \cdot \boldsymbol{a}]_{n-k})$, $[\![\boldsymbol{b}]\!]_{n-k} = ([\boldsymbol{b}]_{n-k}, [\Delta \cdot \boldsymbol{b}]_{n-k})$.
   (b) $\mathcal{F}_{\text{PrepIndMal}}$ computes $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$ and $\Delta \cdot \boldsymbol{c}$. Then $\mathcal{F}_{\text{PrepIndMal}}$ samples a random degree-$(n-1)$ packed Shamir sharing $[\boldsymbol{c}]_{n-1}$. For all $i \in \{1, \dots, k\}$, $\mathcal{F}_{\text{PrepIndMal}}$ samples a random additive sharing $\langle \Delta \cdot c_i \rangle$.

   $\mathcal{F}_{\text{PrepIndMal}}$ distributes the shares of $([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, ([\boldsymbol{c}]_{n-1}, \{\langle \Delta \cdot c_i \rangle\}_{i=1}^k)$ to all parties.
4. **Preparing Random Masked Sharings for Multiplication Gates**: For each group of k multiplication gates, $\mathcal{F}_{\text{PrepIndMal}}$ sets $\boldsymbol{o}^{(1)} = \boldsymbol{o}^{(2)} = \boldsymbol{o}^{(3)} = \mathbf{0} \in \mathbb{F}^k$. Then $\mathcal{F}_{\text{PrepIndMal}}$ samples three random degree-$(n-1)$ packed Shamir sharings $[\boldsymbol{o}^{(1)}]_{n-1}, [\boldsymbol{o}^{(2)}]_{n-1}, [\boldsymbol{o}^{(3)}]_{n-1}$ and distributes the shares to all parties.
5. **Preparing Random Sharings for Input and Output Gates**: For each group of k input gates or output gates, $\mathcal{F}_{\text{PrepIndMal}}$ prepares the following random sharings.
   (a) $\mathcal{F}_{\text{PrepIndMal}}$ prepares a random degree-$(n-1)$ packed Shamir sharing of $\mathbf{0} \in \mathbb{F}^k$, denoted by $[\boldsymbol{o}]_{n-1}$, in the same way as Step 4.
   (b) $\mathcal{F}_{\text{PrepIndMal}}$ also prepares a random packed Beaver triple with authentications $([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, ([\boldsymbol{c}]_{n-1}, \{\langle \Delta \cdot c_i \rangle\}_{i=1}^k)$ in the same way as Step 3. Later, we will view $\boldsymbol{b}$ as an authentication key and $\boldsymbol{c}$ as the MAC of $\boldsymbol{a}$. This allows the input holder to verify the correctness of $\boldsymbol{a}$.

Corrupted Parties: When $\mathcal{F}_{\text{PrepIndMal}}$ prepares random sharings, corrupted parties can choose their shares. $\mathcal{F}_{\text{PrepIndMal}}$ then samples the random sharings based on the secret it generated and the shares chosen by the corrupted parties.

To instantiate the circuit-dependent preprocessing functionality $\mathcal{F}_{\text{PrepMal}}$ using the circuit-independent preprocessing $\mathcal{F}_{\text{PrepIndMal}}$, we follow the idea in [EGPS22]. Concretely,

- Step 2, Step 3, and Step 6(b) in $\mathcal{F}_{\text{PrepMal}}$ are prepared directly in Step 1, Step 2, and Step 5(b) in $\mathcal{F}_{\text{PrepIndMal}}$.
- As for Step 1 in $\mathcal{F}_{\text{PrepMal}}$, recall that the functionality samples a random value for each output wire α of an input gate or a multiplication gate. In $\mathcal{F}_{\text{PrepIndMal}}$, for each output wire α of an input gate or a multiplication gate, the functionality prepares a random degree-$(n-k)$ packed

Shamir sharing $[\lambda_\alpha \cdot \mathbf{1}]_{n-k}$ together with the MAC $\langle \Delta \cdot \lambda_\alpha \rangle$. We will use the secret value λ_α as the random value assigned to α. Then all parties follow Step 1 in $\mathcal{F}_{\mathsf{PrepMal}}$ and compute $([\lambda_\alpha \cdot \mathbf{1}]_{n-k}, \langle \Delta \cdot \lambda_\alpha \rangle)$ for every wire α in the circuit.

– To achieve Step 4, we first follow [EGPS22] and compute a packed Shamir sharing of $\boldsymbol{\lambda_\alpha}$ as follows: Recall that for each wire α_i, all parties hold $([\lambda_{\alpha_i} \cdot \mathbf{1}]_{n-k}, \langle \Delta \cdot \lambda_{\alpha_i} \rangle)$. Let $\boldsymbol{e}_i \in \mathbb{F}_p^k$ be the vector where the i-th entry is 1 and all other entries are 0. Then

$$[\boldsymbol{\lambda_\alpha}]_{n-1} = \sum_{i=1}^{k} \boldsymbol{e}_i * [\lambda_{\alpha_i} \cdot \mathbf{1}]_{n-k}.$$

To reconstruct $\boldsymbol{\lambda_\alpha} - \boldsymbol{a}$ to P_1, all parties locally compute $[\boldsymbol{\lambda_\alpha} - \boldsymbol{a}]_{n-1} = [\boldsymbol{\lambda_\alpha}]_{n-1} - [\boldsymbol{a}]_{n-k} + [\boldsymbol{o}^{(1)}]_{n-1}$ and send their shares to P_1. Here both $[\boldsymbol{a}]_{n-k}$ and $[\boldsymbol{o}^{(1)}]_{n-1}$ are prepared in $\mathcal{F}_{\mathsf{PrepIndMal}}$ and $\boldsymbol{o}^{(1)} = \mathbf{0}$.

To compute $\langle \Delta \cdot (\lambda_{\alpha_i} - a_i) \rangle$, all parties transform $[\Delta \cdot \boldsymbol{a}]_{n-k}$ to an additive sharing $\langle \Delta \cdot a_i \rangle$, compute $\langle \Delta \cdot (\lambda_{\alpha_i} - a_i) \rangle = \langle \Delta \cdot \lambda_{\alpha_i} \rangle - \langle \Delta \cdot a_i \rangle$, and refresh the obtained additive sharing.

– As for Step 5 and Step 6(b), all parties similarly compute $[\boldsymbol{\lambda_\alpha}]_{n-1}$ from $\{[\lambda_{\alpha_i} \cdot \mathbf{1}]_{n-k}\}_{i=1}^{k}$. Note that all parties have already obtained $\{\langle \Delta \cdot \lambda_{\alpha_i} \rangle\}_{i=1}^{k}$.

We describe the protocol Π_{PrepMal} below.

Protocol 1: Π_{PrepMal}

1. All parties invoke $\mathcal{F}_{\mathsf{PrepIndMal}}$.
2. **Setting Authentication Keys**: All parties use $\{[\Delta|_i]_t\}_{i=1}^{k}$ generated in $\mathcal{F}_{\mathsf{PrepIndMal}}$.
3. **Preparing Packed Beaver Triples with Authentications**: All parties use the packed Beaver triples with authentications prepared in $\mathcal{F}_{\mathsf{PrepIndMal}}$.
4. **Distributing $\boldsymbol{\lambda_\alpha} - \boldsymbol{a}$ and $\boldsymbol{\lambda_\beta} - \boldsymbol{b}$ to P_1**: In $\mathcal{F}_{\mathsf{PrepIndMal}}$, for each output wire α of an input gate or a multiplication gate, all parties obtain a random degree-$(n-k)$ packed Shamir sharing with authentication in the form of $([\lambda_\alpha \cdot \mathbf{1}]_{n-k}, \langle \Delta \cdot \lambda_\alpha \rangle)$. All parties locally compute $([\lambda_\alpha \cdot \mathbf{1}]_{n-k}, \langle \Delta \cdot \lambda_\alpha \rangle)$ for every wire α in the circuit.
   For each group of multiplication gates, let $\boldsymbol{\alpha}, \boldsymbol{\beta}$ denote the batch of the first input wires and that of the second input wires respectively. Let $([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, ([\boldsymbol{c}]_{n-1}, \{\langle \Delta \cdot c_i \rangle\}_{i=1}^{k})$ be the packed Beaver triple with authentications associated with these gates. Let $([\boldsymbol{o}^{(1)}]_{n-1}, [\boldsymbol{o}^{(2)}]_{n-1}, [\boldsymbol{o}^{(3)}]_{n-1})$ be the random degree-$(n-1)$ packed Shamir sharings of $\mathbf{0}$ prepared in $\mathcal{F}_{\mathsf{PrepIndMal}}$. All parties run the following steps:
   (a) All parties locally compute
   $$[\boldsymbol{\lambda_\alpha} - \boldsymbol{a}]_{n-1} = \left(\sum_{i=1}^{k} \boldsymbol{e}_i * [\lambda_{\alpha_i} \cdot \mathbf{1}]_{n-k} \right) - [\boldsymbol{a}]_{n-k} + [\boldsymbol{o}^{(1)}]_{n-1}$$
   and send their shares to P_1.
   (b) P_1 reconstructs $\boldsymbol{\lambda_\alpha} - \boldsymbol{a}$.
   (c) For all $i \in \{1, \ldots, k\}$, all parties locally transform $[\Delta \cdot \boldsymbol{a}]_{n-k}$ to an additive sharing $\langle \Delta \cdot a_i \rangle$. Then all parties locally compute $\langle \Delta \cdot (\lambda_{\alpha_i} - a_i) \rangle = \langle \Delta \cdot \lambda_{\alpha_i} \rangle - \langle \Delta \cdot a_i \rangle$. All parties locally refresh the obtained additive sharing.[a] As a result, all parties hold a random additive sharing of $\Delta \cdot (\lambda_{\alpha_i} - a_i)$.
   (d) Repeat the above steps for $\boldsymbol{\lambda_\beta} - \boldsymbol{b}$ using $[\boldsymbol{o}^{(2)}]_{n-1}$.
5. **Preparing Authenticated Packed Sharings for Multiplication Gates**: For each group of multiplication gates with output wires $\boldsymbol{\gamma}$, let $([\boldsymbol{o}^{(1)}]_{n-1}, [\boldsymbol{o}^{(2)}]_{n-1}, [\boldsymbol{o}^{(3)}]_{n-1})$ be the random degree-$(n-1)$ packed Shamir sharings of $\mathbf{0}$ prepared in $\mathcal{F}_{\mathsf{PrepIndMal}}$. Recall that all parties hold $\{([\lambda_{\gamma_i} \cdot \mathbf{1}]_{n-k}, \langle \Delta \cdot \lambda_{\gamma_i} \rangle)\}_{i=1}^{k}$.
   All parties locally compute
   $$[\boldsymbol{\lambda_\gamma}]_{n-1} = \left(\sum_{i=1}^{k} \boldsymbol{e}_i * [\lambda_{\gamma_i} \cdot \mathbf{1}]_{n-k} \right) + [\boldsymbol{o}^{(3)}]_{n-1}.$$
6. **Preparing Random Sharings for Input and Output Gates**: For each group of k input gates or output gates, let $\boldsymbol{\alpha}$ denote the output wires of these k input gates or the input wires of these k output gates. Recall that all parties obtain $[\boldsymbol{o}]_{n-1}$ and $([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, ([\boldsymbol{c}]_{n-1}, \{\langle \Delta \cdot c_i \rangle\}_{i=1}^{k})$ in $\mathcal{F}_{\mathsf{PrepIndMal}}$.

Also recall that all parties hold $\{[\lambda_{\alpha_i} \cdot \mathbf{1}]_{n-k}, \langle \Delta \cdot \lambda_{\alpha_i} \rangle\}_{i=1}^{k}$. All parties locally compute

$$[\boldsymbol{\lambda_\alpha}]_{n-1} = \left(\sum_{i=1}^{k} \boldsymbol{e}_i * [\lambda_{\alpha_i} \cdot \mathbf{1}]_{n-k} \right) + [\boldsymbol{o}]_{n-1}.$$

[a] We discuss how parties locally refresh an additive sharing in Section D

Lemma 1. *Protocol Π_{PrepMal} securely computes $\mathcal{F}_{\mathsf{PrepMal}}$ in the $\mathcal{F}_{\mathsf{PrepIndMal}}$-hybrid model against a malicious adversary who controls t out of n parties.*

Lemma 1 is proven in Section C in the Supplementary Material.

Communication complexity of Π_{PrepMal}. The only communication in Protocol Π_{PrepMal} (ignoring calls to $\Pi_{\mathsf{PrepIndMal}}$) happens in Step 4a. This amounts to $2(n-1)$ shares sent to P_1, per group of k multiplication gates, so $\frac{2n-2}{k} = \frac{4n-4}{\epsilon \cdot n+2} \leq \frac{4}{\epsilon}$ per multiplication gate.

6 Circuit-Independent Preprocessing Phase

In this section, we discuss how to realize the ideal functionality $\mathcal{F}_{\mathsf{PrepIndMal}}$ for the circuit-independent preprocessing phase. Recall that $k = (n-t+1)/2$. For simplicity, we only focus on the scenario where $t \geq n/2$. Due to space constrains, part of the procedures we use to instantiate $\mathcal{F}_{\mathsf{PrepIndMal}}$ appear in Section D in the Supplementary Material. Here, we focus on the fundamental aspects of the instantiation. Recall that $\mathcal{F}_{\mathsf{PrepIndMal}}$ is in charge of generating the following correlations:

1. The global random key $[\Delta|_1]_t, \ldots, [\Delta|_k]_t$;
2. Shamir sharings $[\lambda_\alpha \cdot \mathbf{1}]_{n-k}$ and additive sharings $\langle \Delta \cdot \lambda_\alpha \rangle$ for every wire α;
3. A tuple $(⟦\boldsymbol{a}⟧_{n-k}, ⟦\boldsymbol{b}⟧_{n-k}, ([\boldsymbol{c}]_{n-1}, \{\langle \Delta \cdot c_i \rangle\}_{i=1}^{k}))$ and shares of zero $[\boldsymbol{o}^{(1)}]_{n-1}, [\boldsymbol{o}^{(2)}]_{n-1}, [\boldsymbol{o}^{(3)}]_{n-1}$ for every group of k multiplication gates;
4. A tuple $(⟦\boldsymbol{a}⟧_{n-k}, ⟦\boldsymbol{b}⟧_{n-k}, ([\boldsymbol{c}]_{n-1}, \{\langle \Delta \cdot c_i \rangle\}_{i=1}^{k}))$ and a share of zero $[\boldsymbol{o}]_{n-1}$ for every group of k input or output gates.

In this section we will focus our attention on how to generate the packed Beaver triples $(⟦\boldsymbol{a}⟧_{n-k}, ⟦\boldsymbol{b}⟧_{n-k}, ([\boldsymbol{c}]_{n-1}, \{\langle \Delta \cdot c_i \rangle\}_{i=1}^{k}))$. All of the remaining correlations are discussed in full detail in Section D in the Supplementary Material.

Building blocks: OLE and VOLE. It is known that protocols in the dishonest majority setting require computational assumptions. In our work, these appear in the use of oblivious linear evaluation. Here, we make use of two functionalities, $\mathcal{F}_{\mathsf{nVOLE}}$ and $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$, which sample OLE correlations as follows. We consider an expansion function $\mathtt{Expand} : S \to \mathbb{F}_p^m$ with seed space S and output length m, ultimately corresponding to the amount of correlations we aim at generating.

- $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ is a two-party functionality such that, on input seeds s_a from party P_A and s_b from party P_B, samples $\boldsymbol{v} \leftarrow \mathbb{F}_p^m$, and outputs $\boldsymbol{w} = \boldsymbol{u} * \boldsymbol{x} - \boldsymbol{v}$ to P_A and $\boldsymbol{v}$ to P_B. Here, $\boldsymbol{u} = \mathtt{Expand}(s_a)$ and $\boldsymbol{v} = \mathtt{Expand}(s_b)$. Notice that in this functionality the parties can choose their inputs (at least, choose their seeds).
- $\mathcal{F}_{\mathsf{nVOLE}}$ is an n-party functionality that first distributes $\Delta^i \leftarrow \mathbb{F}$ to each party P_i in an initialize phase, and then, to sample m correlations, the functionality sends $s^i, (\boldsymbol{w}_j^i, \boldsymbol{v}_j^i)_{j \neq i}$ to each party P_i, where s^i is a uniformly random seed, $\boldsymbol{v}_j^i \leftarrow \mathbb{F}_p^m$, and $\boldsymbol{w}_j^i = \boldsymbol{u}^i \cdot \Delta^j - \boldsymbol{v}_i^j$, and $\boldsymbol{u}^i = \mathtt{Expand}(s^i)$. Notice that in this functionality, the parties do not choose their inputs (seeds), but rather, the functionality samples the seeds and sends them to the parties.

The functionalities above are presented in full detail in Section D in the Supplementary Material. At a high level, $\mathcal{F}_{\mathsf{nVOLE}}$ is used to generate authenticated sharings of a uniformly random value, and $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$, which allows the parties to set their inputs, is used to secure multiply two already-shared secret values. $\mathcal{F}_{\mathsf{nVOLE}}$ can be instantiated using pseudo-random correlator generators, as suggested in [RS22]. On the other hand, for $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ we can use the implementation from [RS22]. As we are using *exactly* the same functionalities as in [RS22], we refer the reader to that work for instantiations and complexity measures.

Omitted procedures. For our triple generation protocol, we will make use of a series of procedures that are described in full detail in Section D in the Supplementary Material. These procedures are the following:

- π_{RandSh} (Procedure 5): this procedure generates sharings (under some secret-sharing scheme, which will be clear from context) of uniformly random values. For this, the trick of using a Vandermonde matrix for randomness extraction from [DN07] is used.
- $\pi_{\mathsf{DegReduce}}$ (Procedure 6): this procedure takes as input a sharing $[\boldsymbol{u}]_{n-1}$ and outputs $[\boldsymbol{u}]_{n-k}$. This is achieved by using the trick of masking with a random value $[\boldsymbol{r}]_{n-1}$, opening, and unmasking with $[\boldsymbol{r}]_{n-k}$. This random pair is generated using π_{RandSh}.
- π_{AddTran} (Procedure 7): this procedure takes as input sharings $(\langle \Delta \cdot u_1 \rangle, \ldots, \langle \Delta \cdot u_k \rangle)$ and converts them to $[\Delta \cdot \boldsymbol{u}]_{n-k}$. Once again, the trick of masking with a random sharing $\langle r_1 \rangle, \ldots, \langle r_k \rangle$, opening, and unmasking with $[\boldsymbol{r}]$, is used. The sharing $[\boldsymbol{r}]$ is obtained using π_{RandSh}, and each $\langle r_i \rangle$ can be derived from it non-interactively.
- π_{MACKey} (Procedure 8): this procedure enables the parties to obtain individual Shamir sharings of the global MAC key $[\Delta|_1]_t, \ldots, [\Delta|_k]_t$, starting from additive shares of it $\langle \Delta \rangle$ which are obtained using $\mathcal{F}_{\mathsf{nVOLE}}$. This is done by using the standard trick of masking with a random value $\langle r \rangle$, opening, and unmasking with each $[r|_i]_t$. These random sharings are obtained using π_{RandSh}.
- π_{Auth} (Procedure 9): this procedure takes as input sharings $(\langle u_1 \rangle, \ldots, \langle u_k \rangle)$ to $([\boldsymbol{u}]_{n-1}, \{\langle \Delta \cdot u_i \rangle\}_{i=1}^k)$. The trick here is to mask with $\langle r_1 \rangle, \ldots, \langle r_k \rangle$, open, and adding $[\boldsymbol{r}]_{n-1}$ to obtain $[\boldsymbol{u}]_{n-1}$. The authenticated part can be obtained by first multiplying locally by each $[\Delta|_i]_t$ and then adding each $\langle \Delta \cdot r_i \rangle$. The pair $([\boldsymbol{r}]_{n-1}, \{\langle \Delta \cdot r_i \rangle\}_{i=1}^k)$ is produced using π_{RandSh}.

Preparing packed beaver triples with authentications. The procedure to generate packed Beaver triples with authentications, π_{Triple}, is described below. This protocol calls $\pi_{\mathsf{DegReduce}}$ twice, π_{AddTran} twice, and π_{Auth} once per triple.

Procedure 3: π_{Triple}

Initialization: All parties run the following initialization step only once.

1. Each P_i calls $\mathcal{F}_{\mathsf{nVOLE}}$ with input `Init` and receives Δ^i.
2. All parties invoke π_{RandSh} to prepare random sharings $\{[r|_i]_t\}_{i=1}^k$ and then invoke π_{MACKey} and obtain $\{[\Delta|_i]_t\}_{i=1}^k$.

Generation:

1. Each P_i calls $\mathcal{F}_{\mathsf{nVOLE}}$ twice with input `Extend` and receives the seeds s_a^i, s_b^i. Use the outputs to define degree-$(n-1)$ packed Shamir sharings $\{[\boldsymbol{a}_\ell]_{n-1}\}_{\ell=1}^m, \{[\boldsymbol{b}_\ell]_{n-1}\}_{\ell=1}^m$, where m is the output length of the expansion function defined in $\mathcal{F}_{\mathsf{nVOLE}}$, such that the i-th shares of $\{[\boldsymbol{a}_\ell]_{n-1}\}_{\ell=1}^m$ are $\mathtt{Expand}(s_a^i)$, and the i-th shares of $\{[\boldsymbol{b}_\ell]_{n-1}\}_{\ell=1}^m$ are $\mathtt{Expand}(s_b^i)$. All parties locally compute and refresh $\{(\langle \Delta \cdot a_{\ell,1} \rangle, \ldots, \langle \Delta \cdot a_{\ell,k} \rangle)\}_{\ell=1}^m$ and $\{(\langle \Delta \cdot b_{\ell,1} \rangle, \ldots, \langle \Delta \cdot b_{\ell,k} \rangle)\}_{\ell=1}^m$.
2. Every ordered pair (P_i, P_j) calls $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ with P_i sending s_a^i and P_j sending s_b^j. $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ sends back $\boldsymbol{u}^{i,j}$ to P_i and $\boldsymbol{v}^{j,i}$ to P_j such that $\boldsymbol{u}^{i,j} + \boldsymbol{v}^{j,i} = \mathtt{Expand}(s_a^i) * \mathtt{Expand}(s_b^j)$. All parties locally compute $\{(\langle c_{\ell,1} \rangle, \ldots, \langle c_{\ell,k} \rangle)\}_{\ell=1}^m$ where $\boldsymbol{c}_\ell = \boldsymbol{a}_\ell * \boldsymbol{b}_\ell$.
3. All parties invoke π_{RandSh} to prepare m random sharings in the form of $([\boldsymbol{r}]_{n-k}, [\boldsymbol{r}]_{n-1})$. For all $\ell \in \{1, \ldots, m\}$, consume a pair of random sharings $([\boldsymbol{r}]_{n-k}, [\boldsymbol{r}]_{n-1})$ and invoke $\pi_{\mathsf{DegReduce}}$ to transform $[\boldsymbol{a}_\ell]_{n-1}$ to $[\boldsymbol{a}_\ell]_{n-k}$.
   Repeat this step for $\{[\boldsymbol{b}_\ell]_{n-1}\}_{\ell=1}^m$.
4. All parties invoke π_{RandSh} to prepare m random sharings in the form of $[\boldsymbol{r}]_{n-k}$. For all $\ell \in \{1, \ldots, m\}$, consume a random sharing $[\boldsymbol{r}]_{n-k}$ and invoke π_{AddTran} to transform $(\langle \Delta \cdot a_{\ell,1} \rangle, \ldots, \langle \Delta \cdot a_{\ell,k} \rangle)$ to $[\Delta \cdot \boldsymbol{a}_\ell]_{n-k}$.
   Repeat this step for $\{(\langle \Delta \cdot b_{\ell,1} \rangle, \ldots, \langle \Delta \cdot b_{\ell,k} \rangle)\}_{\ell=1}^m$.
5. All parties follow Step 1 to prepare m random sharings with authentications in the form of $([\boldsymbol{r}]_{n-1}, \{\langle \Delta \cdot r_i \rangle\}_{i=1}^k)$. For all $\ell \in \{1, \ldots, m\}$, consume a random sharing $([\boldsymbol{r}]_{n-1}, \{\langle \Delta \cdot r_i \rangle\}_{i=1}^k)$ and invoke π_{Auth} to transform $(\langle c_{\ell,1} \rangle, \ldots, \langle c_{\ell,k} \rangle)$ to $([\boldsymbol{c}_\ell]_{n-1}, \{\langle \Delta \cdot c_{\ell,i} \rangle\}_{i=1}^k)$.

We remark that the triples produced by π_{Triple} may not be correct, but this can be checked by running a verification step in which the parties generate an extra triple and “sacrifice” it in order to check for correctness. This is described in Section D.4 in the Supplementary Material, where three procedures $\pi_{\mathsf{Sacrifice}}$, $\pi_{\mathsf{CheckZero}}$ and $\pi_{\mathsf{VerifyDeg}}$ to perform this check are introduced.

Communication complexity of π_{Triple}. This is derived as follows

- (Step 3) Two calls to π_{RandSh} to generate two pairs $([\boldsymbol{r}]_{n-k}, [\boldsymbol{r}]_{n-1})$, which costs $2n$, and two calls to $\pi_{\mathsf{DegReduce}}$, which costs $2(2n-k)$. These sum up to $6n-2k$
- (Step 4) Two calls to π_{RandSh} to generate $[\boldsymbol{r}]_{n-k}$ and two calls to π_{AddTran}. These add up to $2(n/2) + 2(k \cdot (n-2) + n + 1)$.
- (Step 5) One call to π_{Auth}, which is $k \cdot (n-2) + n + 1$.

The above totals $k \cdot (3n-8) + 10n + 3$.

Remark 1 (On the output size of $\mathcal{F}_{\mathsf{OLE}}^{prog}$ and $\mathcal{F}_{\mathsf{nVOLE}}$). We make the crucial observation that, in order to obtain m packed multiplication triples, we require the `Expand` function used in Functionalities $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ and $\mathcal{F}_{\mathsf{nVOLE}}$ to output m field elements. However, since each such packed triple is used for a group of k multiplication gates, this effectively means that, if there are $|C|$ multiplication gates in total, we only require `Expand` to output $O(|C|/k) = O(|C|/(\epsilon n))$ correlations. In contrast, as we see in Section E in the Supplementary Material, the best prior work Turbospeedz [BNO19], when instantiated with the preprocessing from Le Mans [RS22], would require $O(|C|)$ correlations from the $\mathcal{F}_{\mathsf{nVOLE}}$ and $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$. As a result, we manage to reduce by a factor of k the expansion requirements on VOLE/OLE techniques, which has a direct effect on the resulting efficiency since this allows us to choose better parameters for the realizations of $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ and $\mathcal{F}_{\mathsf{nVOLE}}$. On the other hand, we require correlations among n parties whereas the Turbospeedz variant we consider here requires these correlations among $(1-\epsilon)n$ parties. We do not explore the concrete effects in efficiency of these observations as it goes beyond the scope of our work, but we refer the reader to [RS22] where an instantiation of $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ and a discussion on PCG-based $\mathcal{F}_{\mathsf{nVOLE}}$ is presented.

Final circuit-independent preprocessing protocol. In Section D.5 in the Supplementary Material we present the final protocol, $\Pi_{\mathsf{PrepIndMal}}$, that puts together the pieces we have discussed so far, together with the techniques presented in Section D in the Supplementary Material to generate the remaining correlations, in order to instantiate Functionality $\mathcal{F}_{\mathsf{PrepIndMal}}$. In Section D.6 in the Supplementary Material, we prove the lemma below. We also analyze the communication complexity of $\Pi_{\mathsf{PrepIndMal}}$ and conclude that, per multiplication gate (ignoring terms that are independent of the circuit size), $6n + \frac{35}{\epsilon}$ elements are required.

Lemma 2. *Protocol $\Pi_{\mathsf{PrepIndMal}}$ securely computes $\mathcal{F}_{\mathsf{PrepIndMal}}$ in the $\{\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}, \mathcal{F}_{\mathsf{nVOLE}}, \mathcal{F}_{\mathsf{Commit}}, \mathcal{F}_{\mathsf{Coin}}\}$-hybrid model against a malicious adversary who controls t out of n parties.*

7 Implementation and Experimental Results

We have fully implemented the three phases of SuperPack, Π_{Online}, Π_{PrepMal} and $\Pi_{\mathsf{PrepIndMal}}$, only ignoring the calls to the $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ and $\mathcal{F}_{\mathsf{nVOLE}}$ functionalities for the implementation of $\Pi_{\mathsf{PrepIndMal}}$. In this section we discuss our experimental results.

Implementation setup. We implement SuperPack[12] by using as a baseline the code of TurboPack [EGPS22].[13] As TurboPack, our program is written in `C++` with no dependencies beyond the standard library. Our implementation includes fully functional networking code. However, for the experiments, we deploy the protocol as multiple processes in a single machine, and emulate real network conditions using the package `netem`[14], which allows us to set bandwidth and latency constraints. We use the same machine as in [EGPS22] for the experiments, namely an AWS `c5.metal` instance with 96 vCPUs and 192 GiB of memory. For our protocol, we use a finite field $\mathbb{F} = \mathbb{F}_p$ where $p = 2^{61} - 1$. We explore how the performance of our protocol is affected by the parameters including the number of parties n, the width and depth of the circuit, the network bandwidth and the values of ϵ such that $t = n(1-\epsilon)$ is the threshold for corrupted parties.

[12] Available at https://github.com/ckweng/SuperPack
[13] Available at https://github.com/deescuderoo/turbopack
[14] https://wiki.linuxfoundation.org/networking/netem

Width	**# Parties**	**Percentage of corrupt parties**			
		90%	80%	70%	60%
100	**16**	0.63, 0.07, 1.22	0.56, 0.06, 1.26	0.33, 0.06, 1.25	0.34, 0.06, 1.26
	32	0.47, 0.11, 2.86	0.47, 0.11, 2.90	0.75, 0.11, 2.86	0.62, 0.09, 2.87
	48	0.35, 0.18, 5.77	0.63, 0.16, 6.41	0.68, 0.15, 6.33	0.68, 0.13, 6.01
1k	**16**	0.44, 0.21, 2.10	0.45, 0.16, 2.14	0.45, 0.20, 2.1	0.66, 0.16, 2.15
	32	0.50, 0.69, 8.38	0.61, 0.63, 9.08	0.58, 0.54, 9.05	0.64, 0.62, 8.78
	48	0.59, 1.46, 21.31	0.97, 1.15, 25.93	0.90, 1.05, 24.70	0.69, 1.01, 24.20
10k	**16**	1.74, 2.03, 14.10	1.49, 1.64, 13.36	1.45, 1.64, 13.39	1.28, 1.43, 12.44
	32	2.36, 6.25, 70.71	2.03, 5.60, 73.98	2.26, 4.80, 70.47	2.32, 4.45, 67.16
	48	3.24, 12.48, 196.97	3.19, 10.39, 238.32	3.49, 9.49, 227.80	4.14, 7.87, 201.17
100k	**16**	11.84, 15.39, 147.03	9.60, 12.46, 140.01	9.63, 12.56, 140.18	8.74, 10.68, 129.89
	32	19.84, 64.61, 714.02	17.46, 46.56, 749.22	18.39, 38.70, 716.26	19.18, 35.03, 682.54
	48	27.62, 124.22, 1978.42	27.56, 103.55, 2374.39	31.55, 92.74, 2256.70	36.98, 78.55, 1998.26

Table 2. Running times in seconds of SuperPack across its three different phases, for different circuit widths, number of parties, and values of ϵ. Each cell is a triple corresponding to the runtimes of the online phase, circuit-dependent offline phase, and circuit-independent offline phase (ignoring OLE calls), respectively. All the circuits have depth 10.

End-to-end runtimes. We first report the running times of our SuperPack protocol for each of the three phases: circuit-independent preprocessing, circuit-dependent preprocessing, and online phase. The results are given in Table 2. In our experiments, we show the running time of our protocol for different parameters. We throttle the bandwidth to 1Gbps and network latency to 1ms to simulate a LAN setting. We generate four generic 10-layer circuits of widths 100, 1k, 10k and 100k. For each circuit, we benchmark the SuperPack protocol of which the number of parties are chosen from $\{16, 32, 48\}$. After fixing the circuit and parties, the percentage of corrupt parties varies from $60\%, 70\%, 80\%$ and 90%. Generally the running time increases as the width and number of parties increase. As demonstrated in Table 2, the majority of running time is incurred by the circuit-independent preprocessing. For $n = 48$ and width larger than 1k, the online phase only occupies less than 5% of the total running time. Furthermore, it is important to observe that the runtimes of the online and circuit-dependent offline phases do not grow at the same rate as the runtimes for the circuit-independent offline phase. This is consistent with what we expect: as can be seen from Table 1, the communication in the first two phases is independent of the number of parties for a given ϵ, which is reflected in the low increase rate in runtimes for these phases (there is still a small but noticeable growth, but this is not surprising since even though communication is constant, *computation* is not). In contrast, the communication in the circuit-independent offline phase depends linearly on the number of parties, which impacts runtimes accordingly.

Experimental comparison to Turbospeedz. Now we compare the online phase of our protocol and compare it against that of Turbospeedz [BNO19],[15] for a varying number of parties n and parameter ϵ. We fix the circuit to have width 100k and depth 10, but we vary the bandwidth in $\{500, 100, 50, 10\}$mbps. The results are given in Table 3. Notice that we do not report concrete runtimes but rather the improvement factor of our online phase with respect to that of Turbospeedz. We also report the communication factors between our protocol and Turbospeedz, for reference. The exact running time of our online protocol for the same settings will be provided in Table 6 in the Supplementary Material.

Table 3 shows interesting patterns. First, as expected (and as analyzed theoretically in Table 5 in Section F), our improvement factor with respect to Turbospeedz improves (*i.e.* increases) as the number of parties grows—since in this case communication in Turbospeedz grows but in our case remains constant—or as the percentage of corruptions decreases—since in this case we can pack more secrets per sharing. Now, notice the following interesting behavior. The last rows next to the "comm. factor" rows represent the improvement factor of our online phase with respect to Turbopeedz, in terms of *communication*. In principle, this is the improvement factor we would expect to see in in terms of runtimes. However, we observe that the expected factor is only reasonably close to the experimental ones for low bandwidths such as 10, 50 and 100 mbps. For the larger

[15] We implemented the online phase of Turbospeedz in our framework for a fair comparison.

bandwidth of 500 mbps, we see that the experimental improvement factors are much lower than the ones we would expect, and in fact, there are several cases where we expect our protocol to be even slightly better, and instead it performs *worse*.

The behavior above can be explained in different ways. First, we notice that it is not surprising that our improvement factor increases as the bandwidth decreases, since in this case the execution of the protocol becomes communication bounded, and computation overhead becomes negligible. In contrast, when the bandwidth is high, communication no longer becomes a bottleneck, and computation plays a major role. Here is where our protocol is in a slight disadvantage: in SUPERPACK, the parties (in particular P_1) must perform polynomial interpolation in a regular basis, while in Turbospeedz these operations correspond to simple field element multiplications, which are less expensive. We remark that our polynomial interpolation is very rudimentary, and a more optimized implementation (*e.g.* using FFTs) may be the key to bridging the gap between our protocol and Turbospeedz, even for the case when bandwidth is large. Finally, we remark that SUPERPACK remains the best option even with high bandwidth when the fraction of honest parties is large enough.

Bandwidth	**# Parties**	**Percentage of corrupt parties**			
		90%	80%	70%	60%
500 mbps	**16**	0.51	0.44	0.42	0.50
	32	0.55	0.68	0.68	0.72
	48	0.58	0.87	1.00	1.14
	64	0.75	0.92	1.30	1.22
	80	0.95	1.27	1.57	1.40
100 mbps	**16**	0.97	1.08	1.05	1.20
	32	1.43	1.67	1.88	1.95
	48	1.51	2.38	2.78	3.07
	64	2.08	2.95	3.37	3.47
	80	2.51	3.88	4.57	4.56
50 mbps	**16**	1.08	1.31	1.31	1.45
	32	1.57	1.99	2.43	2.44
	48	1.73	2.88	3.43	3.76
	64	2.24	3.60	4.55	4.34
	80	2.76	4.51	5.30	5.59
10 mbps	**16**	1.10	1.40	1.39	1.53
	32	1.58	2.00	2.53	2.68
	48	1.81	3.04	3.61	3.94
	64	2.31	3.60	4.73	5.28
	80	2.91	4.56	5.73	6.22
Comm. factor	**16**	0.48	0.85	1.12	1.28
	32	0.96	1.71	2.24	2.56
	48	1.44	2.56	3.36	3.84
	64	1.92	3.41	4.48	5.12
	80	2.4	4.27	5.6	6.4

Table 3. Improvement factors of our online protocol with respect to the online phase in Turbospeedz, for a varying number of parties, ϵ and network bandwidth. The network delay is 1ms for the simulation of LAN network. The number represents how much better (or worse) our online phase is with respect to that of Turbospeedz. The circuits have depth 10 and width 10k. In the final five rows we show the corresponding factors but measuring communication complexity, instead of runtimes.

Acknowledgments

V. Goyal–Supported by the NSF award 1916939, DARPA SIEVE program under Agreement No. HR00112020025, a gift from Ripple, a DoE NETL award, a JP Morgan Faculty Fellowship, a PNC center for financial services innovation award, and a Cylab seed funding award.

This paper was prepared in part for information purposes by the Artificial Intelligence Research Group and the AlgoCRYPT CoE of JPMorgan Chase & Co and its affiliates ("JP Morgan") and is not a product of the Research Department of JP Morgan. JP Morgan makes no representation and warranty whatsoever and disclaims all liability, for the completeness, accuracy, or reliability of the information contained herein. This document is not intended as investment research or investment advice, or a recommendation, offer, or solicitation for the purchase or sale of any security, financial instrument, financial product, or service, or to be used in any way for evaluating the merits of participating in any transaction, and shall not constitute a solicitation under any jurisdiction or to any person, if such solicitation under such jurisdiction or to such person would be unlawful. 2023 JP Morgan Chase & Co. All rights reserved.

References

BBC$^+$19. Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Zero-knowledge proofs on secret-shared data via fully linear PCPs. pages 67–97, 2019.

BDOZ11. Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. pages 169–188, 2011.

Bea92. Donald Beaver. Efficient multiparty protocols using circuit randomization. pages 420–432, 1992.

BGIN20. Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. Efficient fully secure computation via distributed zero-knowledge proofs. pages 244–276, 2020.

BGJK21. Gabrielle Beck, Aarushi Goel, Abhishek Jain, and Gabriel Kaptchuk. Order-C secure multiparty computation for highly repetitive circuits. pages 663–693, 2021.

BGW88. Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). pages 1–10, 1988.

BNO19. Aner Ben-Efraim, Michael Nielsen, and Eran Omri. Turbospeedz: Double your online SPDZ! Improving SPDZ using function dependent preprocessing. pages 530–549, 2019.

Can00. Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, 2000.

Can01. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. pages 136–145, 2001.

CGH$^+$18. Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority MPC for malicious adversaries. pages 34–64, 2018.

Cou19. Geoffroy Couteau. A note on the communication complexity of multiparty computation in the correlated randomness model. pages 473–503, 2019.

DIK10. Ivan Damgård, Yuval Ishai, and Mikkel Krøigaard. Perfectly secure multiparty computation and the computational overhead of cryptography. pages 445–465, 2010.

DKL$^+$13. Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. pages 1–18, 2013.

DN07. Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. pages 572–590, 2007.

DPSZ12. Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. pages 643–662, 2012.

EGPS22. Daniel Escudero, Vipul Goyal, Antigoni Polychroniadou, and Yifan Song. TurboPack: Honest majority MPC with constant online communication. pages 951–964, 2022.

FY92. Matthew K. Franklin and Moti Yung. Communication complexity of secure computation (extended abstract). pages 699–710, 1992.

GIP$^+$14. Daniel Genkin, Yuval Ishai, Manoj M. Prabhakaran, Amit Sahai, and Eran Tromer. Circuits resilient to additive attacks with applications to secure computation. In *Proceedings of the Forty-sixth Annual ACM Symposium on Theory of Computing*, STOC '14, pages 495–504, New York, NY, USA, 2014. ACM.

GIP15. Daniel Genkin, Yuval Ishai, and Antigoni Polychroniadou. Efficient multi-party computation: From passive to active security via secure SIMD circuits. pages 721–741, 2015.

GL05. Shafi Goldwasser and Yehuda Lindell. Secure multi-party computation without agreement. *J. Cryptol.*, 18(3):247–287, jul 2005.

GLO$^+$21. Vipul Goyal, Hanjun Li, Rafail Ostrovsky, Antigoni Polychroniadou, and Yifan Song. ATLAS: Efficient and scalable MPC in the honest majority setting. pages 244–274, 2021.

GPS21. Vipul Goyal, Antigoni Polychroniadou, and Yifan Song. Unconditional communication-efficient MPC via hall's marriage theorem. pages 275–304, 2021.

GPS22. Vipul Goyal, Antigoni Polychroniadou, and Yifan Song. Sharing transformation and dishonest majority MPC with packed secret sharing. pages 3–32, 2022.

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

GSY21. S. Dov Gordon, Daniel Starin, and Arkady Yerukhimovich. The more the merrier: Reducing the cost of large scale MPC. pages 694–723, 2021.

HOSS18a. Carmit Hazay, Emmanuela Orsini, Peter Scholl, and Eduardo Soria-Vazquez. Concretely efficient large-scale MPC with active security (or, TinyKeys for TinyOT). pages 86–117, 2018.

HOSS18b. Carmit Hazay, Emmanuela Orsini, Peter Scholl, and Eduardo Soria-Vazquez. TinyKeys: A new approach to efficient multi-party computation. pages 3–33, 2018.

LN17. Yehuda Lindell and Ariel Nof. A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. pages 259–276, 2017.

NV18. Peter Sebastian Nordholt and Meilof Veeningen. Minimising communication in honest-majority MPC by batchwise multiplication verification. pages 321–339, 2018.

RS22. Rahul Rachuri and Peter Scholl. Le mans: Dynamic and fluid MPC for dishonest majority. pages 719–749, 2022.

Sha79. Adi Shamir. How to Share a Secret. *Commun. ACM*, 22(11):612–613, November 1979.

Supplementary Material

A Appendix to the Preliminaries

A.1 Security Definition

Security Definition In this work, we focus on the honest majority setting. Let $t = (n-1)/2$ be an integer. Let $\mathcal{F}$ be a secure function evaluation functionality. An adversary $\mathcal{A}$ can corrupt at most t parties, provide inputs to corrupted parties, and receive all messages sent to corrupted parties. In this work, we consider both semi-honest adversaries and malicious adversaries.

- If $\mathcal{A}$ is semi-honest, then corrupted parties honestly follow the protocol.
- If $\mathcal{A}$ is fully malicious, then corrupted parties can deviate from the protocol arbitrarily.

Real-World Execution. In the real world, the adversary $\mathcal{A}$ controlling corrupted parties interacts with honest parties. At the end of the protocol, the output of the real-world execution includes the inputs and outputs of honest parties and the view of the adversary.

Ideal-World Execution. In the ideal world, a simulator $\mathcal{S}$ simulates honest parties and interacts with the adversary $\mathcal{A}$. Furthermore, $\mathcal{S}$ has one-time access to $\mathcal{F}$, which includes providing inputs of corrupted parties to $\mathcal{F}$, receiving the outputs of corrupted parties, and sending instructions specified in $\mathcal{F}$ (e.g., asking $\mathcal{F}$ to abort). The output of the ideal-world execution includes the inputs and outputs of honest parties and the view of the adversary.

Semi-honest Security. We say that a protocol π computes $\mathcal{F}$ with perfect security if for all semi-honest adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that the distribution of the output of the real-world execution is *identical* to the distribution in the ideal-world execution.

Security-with-abort. We say that a protocol π securely computes $\mathcal{F}$ with abort if for all adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$, which is allowed to abort the protocol, such that the distribution of the output of the real-world execution is *statistically close* to the distribution in the ideal-world execution.

Hybrid Model. We follow [Can00] and use the hybrid model to prove security. In the hybrid model, all parties are given access to a trusted party (or alternatively, an ideal functionality) which computes a particular function for them. The modular sequential composition theorem from [Can00] shows that it is possible to replace the ideal functionality used in the construction by a secure protocol computing this function. When the ideal functionality is denoted by g, we say the construction works in the g-hybrid model.

Client-server Model. To simplify the security proofs, we consider consider the client-server model. In the client-server model, clients provide inputs to the functionality and receive outputs, and servers can participate in the computation but do not have inputs or get outputs. Each party may have different roles in the computation. Note that, if every party plays a single client and a single server, this corresponds to a protocol in the standard MPC model. Let c denote the number of clients and n denote the number of servers. For all clients and servers, we assume that every two of them are connected via a secure (private and authentic) synchronous channel so that they can directly send messages to each other. The communication complexity is measured in the same way as that in the standard MPC model.

Security in the Client-server Model. In the client-server model, an adversary $\mathcal{A}$ can corrupt at most c clients and t servers, provide inputs to corrupted clients, and receive all messages sent to corrupted clients and servers. The security is defined similarly to the standard MPC model.

Benefits of the Client-server Model. In our construction, the clients only participate in the input phase and the output phase. The main computation is conducted by the servers. For simplicity, we use $\{P_1, \ldots, P_n\}$ to denote the n servers, and refer to the servers as parties. Let $\mathcal{C}orr$ denote the set of all corrupted parties and $\mathcal{H}$ denote the set of all honest parties. One benefit of the client-server model is that it is sufficient to only consider maximum adversaries, i.e., adversaries which corrupt exactly t parties. At a high level, for an adversary $\mathcal{A}$ which controls $t' < t$ parties, we may construct another adversary $\mathcal{A}'$ which controls additional $t - t'$ parties and behaves as follows:

– For a party corrupted by $\mathcal{A}$, $\mathcal{A}'$ follows the instructions of $\mathcal{A}$. This is achieved by passing messages between this party and other $n - t'$ honest parties.
– For a party which is not corrupted by $\mathcal{A}$, but controlled by $\mathcal{A}'$, $\mathcal{A}'$ honestly follows the protocol.

Note that, if a protocol is secure against $\mathcal{A}'$, then this protocol is also secure against $\mathcal{A}$ since the additional $t - t'$ parties controlled by $\mathcal{A}'$ honestly follow the protocol in both cases. Thus, we only need to focus on $\mathcal{A}'$ instead of $\mathcal{A}$. Note that in the regular model, each honest party may have input. The same argument does not hold since the input of honest parties controlled by $\mathcal{A}'$ may be compromised.

A.2 Extra Functionalities

Functionality for Secure Computation. We first describe the functionality $\mathcal{F}_{\mathsf{MPC}}$ below, which models the task of secure computation, and constitutes the main functionality we wish to implement. The functionality interacts with the set of clients $\mathcal{C} = \{C_1, \ldots, C_m\}$, who provide input and receive output, and the set of parties $\mathcal{P} = \{P_1, \ldots, P_n\}$, who carry out the computation. Let $(\mathtt{Y}_1, \ldots, \mathtt{Y}_m) = f(\mathtt{X}_1, \ldots, \mathtt{X}_m)$ be an arithmetic circuit, which for modeling purposes we regard as a set of multivariate polynomials over $\mathbb{F}$, where each $\mathtt{X}_i$ has dimension I_i and y_i has dimension O_i, for $i \in \{1, \ldots, m\}$. Each client C_i provides the inputs $\boldsymbol{x}_i$, and obtains the outputs $\boldsymbol{y}_i$. The functionality $\mathcal{F}_{\mathsf{MPC}}$ below models secure computation of the function f.

We remark that this corresponds to a case of *non-reactive* computation, in which the clients evaluate the function f, which can only depend on their inputs. In contrast, *reactive* computation enables the clients to receive partial outputs, and provide new inputs based on these values, in order to compute the final result. This is better modeled using the *arithmetic black-box model*, which we do not use in our work. We notice however that this is done for the purpose of simplicity, and our protocol could easily be made to work in the reactive setting as well.

Functionality 3: $\mathcal{F}_{\mathsf{MPC}}$

Let $(\mathtt{Y}_1, \ldots, \mathtt{Y}_m) = f(\mathtt{X}_1, \ldots, \mathtt{X}_m)$ be an arithmetic circuit.

– **Clients provide inputs.** For each $i \in \{1, \ldots, m\}$, client C_i sends $\boldsymbol{x}_i \in \mathbb{F}^{I_i}$ to $\mathcal{F}_{\mathsf{MPC}}$, which stores these values
– **Functionality computes.** Once all the inputs are received, $\mathcal{F}_{\mathsf{MPC}}$ computes $(\boldsymbol{y}_1, \ldots, \boldsymbol{y}_m) = f(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m)$, and stores these values
– **Clients receive outputs.** For each $i \in \{1, \ldots, m\}$, client C_i receives the stored values $\boldsymbol{y}_i \in \mathbb{F}^{O_i}$ from $\mathcal{F}_{\mathsf{MPC}}$.

B Appendix to Section 4

B.1 Procedure for Output Gates π_{Output}

Procedure 4: π_{Output}

Computation of Output Gates

1. For each group of output gates that belongs to `Client`, let $\boldsymbol{\alpha}$ denote the batch of input wires of these output gates. All parties receive from $\mathcal{F}_{\mathsf{PrepMal}}$
   – A random degree-$(n-1)$ packed Shamir sharing $[\boldsymbol{\lambda}_{\boldsymbol{\alpha}}]_{n-1}$ with MACs $\{\langle \varDelta \cdot \lambda_{\alpha_i} \rangle\}_{i=1}^{k}$.

- A packed Beaver triple with authentications

$$([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, ([\boldsymbol{c}]_{n-1}, \{\langle \Delta \cdot c_i \rangle\}_{i=1}^{k}).$$

And P_1 learns $\boldsymbol{\mu_\alpha}$ in clear.

2. All parties invoke $\mathcal{F}_{\mathsf{Commit}}$ to commit their shares of $[\boldsymbol{a}]_{n-k}, [\boldsymbol{b}]_{n-k}, [\boldsymbol{c}]_{n-1}$ towards `Client`.
3. All parties locally compute $[\boldsymbol{\lambda_\alpha} - \boldsymbol{a}]_{n-1} = [\boldsymbol{\lambda_\alpha}]_{n-1} - [\boldsymbol{a}]_{n-k}$ and send their shares to P_1.
4. P_1 reconstructs the secrets $\boldsymbol{\lambda_\alpha} - \boldsymbol{a}$ and computes $\boldsymbol{v_\alpha} - \boldsymbol{a} = \boldsymbol{\mu_\alpha} + \boldsymbol{\lambda_\alpha} - \boldsymbol{a}$. Then P_1 distributes the shares of $[\boldsymbol{v_\alpha} - \boldsymbol{a}]_{2k-2}$ to all parties.
5. For all $i \in \{1, \ldots, k\}$, all parties locally compute $\langle \theta_{\alpha_i} \rangle$ as follows.
   (a) Recall that all parties have computed additive sharings of the MACs of the μ values for output wires of multiplication gates and input gates in previous layers. By using these additive sharings, all parties locally compute $\langle \Delta \cdot \mu_{\alpha_i} \rangle$.
   (b) Recall that all parties hold $\langle \Delta \cdot \lambda_{\alpha_i} \rangle$ and $[\Delta \cdot \boldsymbol{a}]_{n-k}$. They locally transform $[\Delta \cdot \boldsymbol{a}]_{n-k}$ to an additive sharing $\langle \Delta \cdot a_i \rangle$ and compute $\langle \Delta \cdot (v_{\alpha_i} - a_i) \rangle$.
   (c) Also recall that all parties hold $[\Delta|_i]_t$. All parties locally compute $[\Delta|_i]_t * [\boldsymbol{v_\alpha} - \boldsymbol{a}]_{2k-2}$ and transform it to an additive sharing $\langle \overline{\Delta \cdot (v_{\alpha_i} - a_i)} \rangle$.
   (d) All parties locally compute $\langle \theta_{\alpha_i} \rangle = \langle \Delta \cdot (v_{\alpha_i} - a_i) \rangle - \langle \overline{\Delta \cdot (v_{\alpha_i} - a_i)} \rangle$.

Verification of the Computation

6. All parties verify that all degree-$(k-1)$ packed Shamir sharings distributed by P_1 are valid. Suppose all degree-$(k-1)$ packed Shamir sharings distributed by P_1 are denoted by

$$[\boldsymbol{x}_1]_{k-1}, \ldots, [\boldsymbol{x}_m]_{k-1}.$$

   (a) All parties call $\mathcal{F}_{\mathsf{Coin}}$ to obtain random values $\chi_1, \ldots, \chi_m \in \mathbb{F}$.
   (b) All parties locally compute $[\boldsymbol{x}]_{k-1} = \sum_{\ell=1}^{m} \chi_\ell \cdot [\boldsymbol{x}_\ell]_{k-1}$.
   (c) All parties exchange their shares of $[\boldsymbol{x}]_{k-1}$ and check that the shares of $[\boldsymbol{x}]_{k-1}$ lie on a degree-$(k-1)$ polynomial. If false, all parties abort.
7. Recall that for each input wire α of a multiplication gate or an output gate, all parties have computed $\langle \theta_\alpha \rangle$. All parties verify that $\theta_\alpha = 0$ as follows. Suppose all parties hold $\langle \theta_1 \rangle, \ldots, \langle \theta_m \rangle$.
   (a) All parties call $\mathcal{F}_{\mathsf{Coin}}$ to obtain random values $\chi'_1, \ldots, \chi'_m \in \mathbb{F}$.
   (b) All parties locally compute $\langle \theta \rangle = \sum_{j=1}^{m} \chi'_j \cdot \langle \theta_j \rangle$ and locally refresh the obtained additive sharing.[a]
   (c) All parties call $\mathcal{F}_{\mathsf{Commit}}$ to commit their shares of $\langle \theta \rangle$ towards each other.
   (d) All parties open the commitments of the shares of $\langle \theta \rangle$ and check whether it is an additive sharing of 0. If not, all parties abort.

Reconstruction of Outputs

8. All parties send to `Client` their shares of $[\boldsymbol{v_\alpha} - \boldsymbol{a}]_{2k-2}$, and open the commitments of their shares of $[\boldsymbol{a}]_{n-k}, [\boldsymbol{b}]_{n-k}, [\boldsymbol{c}]_{n-1}$ to `Client`.
9. `Client` reconstructs the secrets $\boldsymbol{v_\alpha} - \boldsymbol{a}, \boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}$ and checks whether $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$. If not, `Client` aborts. Otherwise, `Client` computes and outputs $\boldsymbol{v_\alpha}$.

[a] We will discuss how parties locally refresh an additive sharing in Section D

Communication complexity of π_{Output}. The communication complexity per group of k output gates, ignoring the verification, consists of:

- (Step 3) $n - 1$ shares from the parties to P_1
- (Step 4) $n - (k-1) - 1 = (1 - \frac{\epsilon}{2})n - 1$ shares from P_1 to the parties.
- (Step 2 and Step 8) n shares from parties to `Client`, and n commitments with openings, each being a commitment to 3 field elements. In total this is $n + 3n$ field elements, plus $2n\sigma$ bits from the commitments ($n\sigma$ from committing, and $n\sigma$ from opening). The commitments are independent of the total number of groups of k gates. For simplifying the analysis let us take $\sigma = 128$ and $|\mathbb{F}| \geq 2^\kappa \approx 2^{40}$, so that these $2n\sigma$ bits correspond to $6n$ field elements.

Suppose that `Client` owns $k \cdot m$ output gates, or m groups of k gates each. The total communication is then $m \cdot \left((6 - \frac{\epsilon}{2})n - 2\right)$ field elements, plus $6n$ field elements. Per output gate, dividing by

$m \cdot k$ with $k = \frac{\epsilon}{2} \cdot n + 1$, we obtain that the cost is $\frac{(12-\epsilon)n-4+12n/m}{2k} = \frac{(12-\epsilon)n-4+12n/m}{\epsilon \cdot n+2} \leq \frac{12+12/m}{\epsilon}$. As m grows the cost of the commitments is amortized away and this equals $12/\epsilon$, but for small m such as $m = 1$, the cost is $24/\epsilon$.

We ignore the cost of the verification step, since it is independent of the amount of gates of any type.

B.2 Full Online Protocol Π_{Online}

Protocol 2: Π_{Online}

The parties compute the circuit in a layer-by-layer manner. The parties call $\mathcal{F}_{\mathsf{PrepMal}}$ to obtain the following:

- Sharings $([\Delta|_1]_t, \ldots, [\Delta|_k]_t)$;
- For every group of k multiplication gates with inputs $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$, and outputs $\boldsymbol{\gamma}$:
  - A tuple $([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, ([\boldsymbol{c}]_{n-1}, \{\langle \Delta \cdot c_i \rangle\}_{i=1}^k))$,
  - A pair $\{\langle \Delta \cdot (\lambda_{\alpha_i} - a_i) \rangle, \langle \Delta \cdot (\lambda_{\beta_i} - b_i) \rangle\}_{i=1}^k$,
  - A packed Shamir sharing $[\boldsymbol{\lambda_\gamma}]_{n-1}$ and additive sharings $\{\langle \Delta \cdot \lambda_{\gamma_i} \rangle\}_{i=1}^k$,
  - P_1 obtains $\boldsymbol{\lambda_\alpha} - \boldsymbol{a} + \boldsymbol{\delta_\alpha}$ and $\boldsymbol{\lambda_\beta} - \boldsymbol{b} + \boldsymbol{\delta_\beta}$.
- For every group of k input gates or output gates, let $\boldsymbol{\alpha}$ be the output wires of these k input gates or the input wires of these k output gates. All parties prepare:
  - A tuple $([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, ([\boldsymbol{c}]_{n-1}, \{\langle \Delta \cdot c_i \rangle\}_{i=1}^k))$,
  - A packed Shamir sharing $[\boldsymbol{\lambda_\alpha}]_{n-1}$ and additive sharings $\{\langle \Delta \cdot \lambda_{\alpha_i} \rangle\}_{i=1}^k$.

The parties proceed as follows:

1. The parties begin by executing Procedure π_{Input}, which results in P_1 learning $\boldsymbol{\mu_\alpha}$ and the parties obtaining additive shares $\{\langle \Delta \cdot \mu_{\alpha_i} \rangle\}_{i=1}^k$, for every input batch with input wires $\boldsymbol{\alpha}$.
2. For every addition gate with input wires α and β, and output wire γ, P_1 locally computes $\mu_\gamma = \mu_\alpha + \mu_\beta$, and the parties locally compute $\langle \Delta \cdot \mu_\gamma \rangle = \langle \Delta \cdot \mu_\alpha \rangle + \langle \Delta \cdot \mu_\beta \rangle$
3. For every group of k multiplication gates with input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$ and output wires $\boldsymbol{\gamma}$, the parties run procedure π_{Mult}, which results in P_1 learning $\boldsymbol{\mu_\gamma}$ and the parties obtaining additive shares $\{\langle \Delta \cdot \lambda_{\gamma_i} \rangle\}_{i=1}^k$. The parties also obtain sharings $\langle \theta_{\alpha_i} \rangle$ and $\langle \theta_{\beta_i} \rangle$ for $i \in \{1, \ldots, k\}$, as explained in Procedure π_{Mult} (Proc. 2).
4. The parties call Procedure π_{Output} to (1) maintain the invariant for the output gates, (2) verify the correctness of the computation (using the sharings $\langle \theta_{\alpha_i} \rangle$ and $\langle \theta_{\beta_i} \rangle$ from the multiplication gates above), and (3) reconstruct the output to the clients.

Theorem 2. *Let c denote the number of servers and n denote the number of parties (servers). For all $0 < \epsilon \leq 1/2$, protocol Π_{Online} instantiates Functionality $\mathcal{F}_{\mathsf{MPC}}$ in the $\mathcal{F}_{\mathsf{PrepMal}}$-hybrid model, with statistical security against a fully malicious adversary who can control up to c clients and $t = (1-\epsilon)n$ parties (servers).*

Proof. We will construct a simulator $\mathcal{S}$ to simulate the behaviors of honest parties. Let $\mathcal{C}orr$ denote the set of corrupted parties and $\mathcal{H}$ denote the set of honest parties. Recall that in the client-server model, it is sufficient to only consider the scenario where the number of corrupted parties is exactly $t = (1-\epsilon) \cdot n$. Also recall that $k = (n - t + 1)/2$. In the following, for a value v, we use $\eta(v)$ to denote additive error of v due to the malicious behaviors of corrupted parties. For a vector of values $\boldsymbol{v} = (v_1, \ldots, v_k)$, we simply write $\eta(\boldsymbol{v}) = (\eta(v_1), \ldots, \eta(v_k))$.

The simulator $\mathcal{S}$ works as follows. In the beginning, $\mathcal{S}$ emulates the ideal functionality $\mathcal{F}_{\mathsf{PrepMal}}$ as follows.

- For sharings $([\Delta|_1]_t, \ldots, [\Delta|_k]_t)$, $\mathcal{S}$ receives the shares of corrupted parties.
- For every group of k multiplication gates with inputs $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$, and outputs $\boldsymbol{\gamma}$:
  - $\mathcal{S}$ receives the shares of $([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, ([\boldsymbol{c}]_{n-1}, \{\langle \Delta \cdot c_i \rangle\}_{i=1}^k))$ of corrupted parties.
  - $\mathcal{S}$ receives the shares of $\{\langle \Delta \cdot (\lambda_{\alpha_i} - a_i) \rangle, \langle \Delta \cdot (\lambda_{\beta_i} - b_i) \rangle\}_{i=1}^k$ of corrupted parties.
  - $\mathcal{S}$ receives the shares of $[\boldsymbol{\lambda_\gamma}]_{n-1}$ and additive sharings $\{\langle \Delta \cdot \lambda_{\gamma_i} \rangle\}_{i=1}^k$ of corrupted parties.
  - $\mathcal{S}$ receives two additive errors $\boldsymbol{\delta_\alpha}, \boldsymbol{\delta_\beta}$. $\mathcal{S}$ samples two random vectors as $\boldsymbol{\lambda_\alpha} - \boldsymbol{a}$ and $\boldsymbol{\lambda_\beta} - \boldsymbol{b}$, and computes $\boldsymbol{\lambda_\alpha} - \boldsymbol{a} + \boldsymbol{\delta_\alpha}$ and $\boldsymbol{\lambda_\beta} - \boldsymbol{b} + \boldsymbol{\delta_\beta}$. If P_1 is corrupted, $\mathcal{S}$ sends these two vectors to P_1.

– For every group of k input gates or output gates, let $\boldsymbol{\alpha}$ be the output wires of these k input gates or the input wires of these k output gates.
  - $\mathcal{S}$ receives the shares of $([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, ([\boldsymbol{c}]_{n-1}, \{\langle \Delta \cdot c_i \rangle\}_{i=1}^{k}))$ of corrupted parties.
  - $\mathcal{S}$ receives the shares of $[\boldsymbol{\lambda}_{\boldsymbol{\alpha}}]_{n-1}$ and additive sharings $\{\langle \Delta \cdot \lambda_{\alpha_i} \rangle\}_{i=1}^{k}$ of corrupted parties.

Then $\mathcal{S}$ proceeds to simulate the whole protocol:

1. In Step 1, $\mathcal{S}$ simulates π_{Input} as follows. For each group of k input gates that belong to some `Client`, let $\boldsymbol{\alpha}$ denote the output wires of these input gates. When `Client` is corrupted,
   (a) In Step 2 of π_{Input}, $\mathcal{S}$ randomly samples $\boldsymbol{\lambda}_{\boldsymbol{\alpha}}, \boldsymbol{a}, \boldsymbol{b}$ and computes $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$. Then $\mathcal{S}$ randomly samples the shares of $[\boldsymbol{\lambda}_{\boldsymbol{\alpha}}]_{n-1}$, $[\boldsymbol{a}]_{n-k}$, $[\boldsymbol{b}]_{n-k}$, $[\boldsymbol{c}]_{n-1}$ of honest parties based on the secrets $\mathcal{S}$ just generated and the shares of corrupted parties. Next, $\mathcal{S}$ sends these shares to `Client` on behalf of honest parties.
   (b) In Step 4 of π_{Input}, $\mathcal{S}$ receives the shares of $[\boldsymbol{v}_{\boldsymbol{\alpha}} - \boldsymbol{a}]_{2k-2}$ of honest parties. $\mathcal{S}$ reconstructs the whole sharing $[\boldsymbol{v}_{\boldsymbol{\alpha}} - \boldsymbol{a}]_{2k-2}$ using the shares of honest parties and recovers the secrets $\boldsymbol{v}_{\boldsymbol{\alpha}} - \boldsymbol{a}$. Then $\mathcal{S}$ computes the inputs $\boldsymbol{v}_{\boldsymbol{\alpha}}$ of `Client` and learns the shares of $[\boldsymbol{v}_{\boldsymbol{\alpha}} - \boldsymbol{a}]_{2k-2}$ of corrupted parties.
   If P_1 is honest, $\mathcal{S}$ also receives $\overline{\boldsymbol{\mu}_{\boldsymbol{\alpha}}}$ (which is to distinguish from the correct values $\boldsymbol{\mu}_{\boldsymbol{\alpha}} = \boldsymbol{v}_{\boldsymbol{\alpha}} - \boldsymbol{\lambda}_{\boldsymbol{\alpha}}$). Note that both the corrupted `Client` and $\mathcal{S}$ learns $\boldsymbol{\mu}_{\boldsymbol{\alpha}}$. $\mathcal{S}$ computes $\eta(\boldsymbol{\mu}_{\boldsymbol{\alpha}}) = \overline{\boldsymbol{\mu}_{\boldsymbol{\alpha}}} - \boldsymbol{\mu}_{\boldsymbol{\alpha}}$.
   (c) In Step 5 of π_{Input}, $\mathcal{S}$ follows the protocol and computes the shares of $\langle \Delta \cdot \mu_{\alpha_i} \rangle$ of corrupted parties for all $i \in \{1, \ldots, k\}$.

   When `Client` is honest,
   (a) In Step 2 of π_{Input}, $\mathcal{S}$ receives the shares of $[\boldsymbol{\lambda}_{\boldsymbol{\alpha}}]_{n-1}$, $[\boldsymbol{a}]_{n-k}, [\boldsymbol{b}]_{n-k}, [\boldsymbol{c}]_{n-1}$ of corrupted parties.
      – For each of $[\boldsymbol{\lambda}_{\boldsymbol{\alpha}}]_{n-1}, [\boldsymbol{c}]_{n-1}$, $\mathcal{S}$ defines a degree-$(n-1)$ packed Shamir sharing such that the shares of corrupted parties are the differences between the shares received from corrupted parties and those they should hold, and the shares of honest parties are 0. Then $\mathcal{S}$ reconstructs the secrets, denoted by $\eta(\boldsymbol{\lambda}_{\boldsymbol{\alpha}}), \eta(\boldsymbol{c})$.
      – For each of $[\boldsymbol{a}]_{n-k}, [\boldsymbol{b}]_{n-k}$, $\mathcal{S}$ defines a degree-$(n-k)$ packed Shamir sharing such that the shares of corrupted parties are the differences between the shares received from corrupted parties and those they should hold, and the shares of honest parties are 0. Then $\mathcal{S}$ checks whether all shares lie on a degree-$(n-k)$ polynomial. If not, $\mathcal{S}$ aborts on behalf of the honest `Client`. Otherwise, $\mathcal{S}$ reconstructs the secrets, denoted by $\eta(\boldsymbol{a}), \eta(\boldsymbol{b})$.

      If any of $\eta(\boldsymbol{a}), \eta(\boldsymbol{b}), \eta(\boldsymbol{c})$ is not an all-0 vector, $\mathcal{S}$ aborts on behalf of the honest `Client`.
   (b) In Step 4 of π_{Input}, $\mathcal{S}$ samples two random vectors as $\boldsymbol{v}_{\boldsymbol{\alpha}} - \boldsymbol{a}$ and $\boldsymbol{\mu}_{\boldsymbol{\alpha}}$. Then $\mathcal{S}$ computes and distributes the shares of $[\boldsymbol{v}_{\boldsymbol{\alpha}} - \boldsymbol{a}]_{2k-2}$ to corrupted parties. $\mathcal{S}$ also sets $\eta(\boldsymbol{\mu}_{\boldsymbol{\alpha}}) = -\eta(\boldsymbol{\lambda}_{\boldsymbol{\alpha}})$ and computes $\overline{\boldsymbol{\mu}_{\boldsymbol{\alpha}}} = \boldsymbol{\mu}_{\boldsymbol{\alpha}} + \eta(\boldsymbol{\mu}_{\boldsymbol{\alpha}})$. If P_1 is corrupted, $\mathcal{S}$ sends $\overline{\boldsymbol{\mu}_{\boldsymbol{\alpha}}}$ to P_1.
   (c) In Step 5 of π_{Input}, $\mathcal{S}$ follows the protocol and computes the shares of $\langle \Delta \cdot \mu_{\alpha_i} \rangle$ of corrupted parties for all $i \in \{1, \ldots, k\}$.

   At the end of this step, $\mathcal{S}$ sends to $\mathcal{F}_{\mathsf{MPC}}$ the inputs of corrupted clients and receives their outputs.
2. In Step 2, for each addition gate, $\mathcal{S}$ computes $\mu_\gamma = \mu_\alpha + \mu_\beta$ and the shares of $\langle \Delta \cdot \mu_\gamma \rangle$ of corrupted parties. If P_1 is honest, $\mathcal{S}$ also computes $\overline{\mu_\gamma} = \overline{\mu_\alpha} + \overline{\mu_\beta}$.
3. In Step 3, for every group of k multiplication gates with input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$ and output wires $\boldsymbol{\gamma}$, $\mathcal{S}$ simulates π_{Mult} as follows. When P_1 is honest,
   (a) In Step 2 of π_{Mult}, $\mathcal{S}$ computes $\overline{\boldsymbol{v}_{\boldsymbol{\alpha}} - \boldsymbol{a}} = \overline{\boldsymbol{\mu}_{\boldsymbol{\alpha}}} + (\boldsymbol{\lambda}_{\boldsymbol{\alpha}} - \boldsymbol{a} + \boldsymbol{\delta}_{\boldsymbol{\alpha}})$. $\mathcal{S}$ also computes $\boldsymbol{v}_{\boldsymbol{\alpha}} - \boldsymbol{a} = \boldsymbol{\mu}_{\boldsymbol{\alpha}} + \boldsymbol{\lambda}_{\boldsymbol{\alpha}} - \boldsymbol{a}$. Similarly, $\mathcal{S}$ computes $\overline{\boldsymbol{v}_{\boldsymbol{\beta}} - \boldsymbol{b}}$ and $\boldsymbol{v}_{\boldsymbol{\beta}} - \boldsymbol{b}$. $\mathcal{S}$ sets $\eta(\boldsymbol{v}_{\boldsymbol{\alpha}} - \boldsymbol{a}) = \overline{\boldsymbol{v}_{\boldsymbol{\alpha}} - \boldsymbol{a}} - (\boldsymbol{v}_{\boldsymbol{\alpha}} - \boldsymbol{a})$ and $\eta(\boldsymbol{v}_{\boldsymbol{\beta}} - \boldsymbol{b}) = \overline{\boldsymbol{v}_{\boldsymbol{\beta}} - \boldsymbol{b}} - (\boldsymbol{v}_{\boldsymbol{\beta}} - \boldsymbol{b})$. Then $\mathcal{S}$ distributes the shares of $[\overline{\boldsymbol{v}_{\boldsymbol{\alpha}} - \boldsymbol{a}}]_{k-1}$ and $[\overline{\boldsymbol{v}_{\boldsymbol{\beta}} - \boldsymbol{b}}]_{k-1}$ to corrupted parties on behalf of P_1.
   (b) In Step 3 of π_{Mult}, $\mathcal{S}$ follows the protocol and computes the shares of $\langle \theta_{\alpha_i} \rangle$ and $\langle \theta_{\beta_i} \rangle$ of corrupted parties. Note that $\theta_{\alpha_i} = \Delta \cdot \eta(v_{\alpha_i} - a_i)$ and $\theta_{\beta_i} = \Delta \cdot \eta(v_{\beta_i} - b_i)$.
   (c) In Step 4 of π_{Mult}, $\mathcal{S}$ computes the shares of $[\boldsymbol{\mu}_{\boldsymbol{\gamma}}]_{n-1}$ of corrupted parties. In Step 5 of π_{Mult}, $\mathcal{S}$ follows the protocol and computes the shares of $\langle \Delta \cdot \mu_{\gamma_i} \rangle$ of corrupted parties for all $i \in \{1, \ldots, k\}$.

(d) In Step 6, $\mathcal{S}$ samples random values as the shares of $[\boldsymbol{\mu_\gamma}]_{n-1}$ of honest parties. Then $\mathcal{S}$ reconstructs the secrets $\boldsymbol{\mu_\gamma}$. $\mathcal{S}$ also receives the shares of corrupted parties on behalf of P_1. $\mathcal{S}$ uses the shares of corrupted parties he received and the shares of honest parties to compute the secrets $\overline{\boldsymbol{\mu_\gamma}}$ and sets $\eta(\boldsymbol{\mu_\gamma}) = \overline{\boldsymbol{\mu_\gamma}} - \boldsymbol{\mu_\gamma}$.

When P_1 is corrupted,

(a) In Step 2 of π_{Mult}, $\mathcal{S}$ computes $\boldsymbol{v_\alpha} - \boldsymbol{a} = \boldsymbol{\mu_\alpha} + \boldsymbol{\lambda_\alpha} - \boldsymbol{a}$ and $\boldsymbol{v_\beta} - \boldsymbol{b} = \boldsymbol{\mu_\beta} + \boldsymbol{\lambda_\beta} - \boldsymbol{b}$. $\mathcal{S}$ receives the shares of $[\boldsymbol{v_\alpha} - \boldsymbol{a}]_{k-1}$ and $[\boldsymbol{v_\beta} - \boldsymbol{b}]_{k-1}$ of honest parties. Then $\mathcal{S}$ uses the shares of the first k honest parties to reconstruct the whole sharings and computes the secrets $\overline{\boldsymbol{v_\alpha} - \boldsymbol{a}}$ and $\overline{\boldsymbol{v_\beta} - \boldsymbol{b}}$. If the shares of $[\boldsymbol{v_\alpha} - \boldsymbol{a}]_{k-1}$ of honest parties do not lie on a degree-$(k-1)$ packed Shamir sharing, or the shares of $[\boldsymbol{v_\beta} - \boldsymbol{b}]_{k-1}$ of honest parties do not lie on a degree-$(k-1)$ packed Shamir sharing, $\mathcal{S}$ marks the execution as `abort`. Later, $\mathcal{S}$ will abort in the verification step.
$\mathcal{S}$ sets $\eta(\boldsymbol{v_\alpha} - \boldsymbol{a}) = \overline{\boldsymbol{v_\alpha} - \boldsymbol{a}} - (\boldsymbol{v_\alpha} - \boldsymbol{a})$ and $\eta(\boldsymbol{v_\beta} - \boldsymbol{b}) = \overline{\boldsymbol{v_\beta} - \boldsymbol{b}} - (\boldsymbol{v_\beta} - \boldsymbol{b})$.

(b) In Step 3 of π_{Mult}, $\mathcal{S}$ follows the protocol and computes the shares of $\langle \theta_{\alpha_i} \rangle$ and $\langle \theta_{\beta_i} \rangle$ of corrupted parties. Note that, when the execution is not marked as `abort` (i.e., all degree-$(k-1)$ packed Shamir sharings distributed by P_1 are valid), $\theta_{\alpha_i} = \Delta \cdot \eta(v_{\alpha_i} - a_i)$ and $\theta_{\beta_i} = \Delta \cdot \eta(v_{\beta_i} - b_i)$.

(c) In Step 4 of π_{Mult}, $\mathcal{S}$ computes the shares of $[\boldsymbol{\mu_\gamma}]_{n-1}$ of corrupted parties. In Step 5 of π_{Mult}, $\mathcal{S}$ follows the protocol and computes the shares of $\langle \Delta \cdot \mu_{\gamma_i} \rangle$ of corrupted parties for all $i \in \{1, \ldots, k\}$.

(d) In Step 6, $\mathcal{S}$ samples random values as the shares of $[\boldsymbol{\mu_\gamma}]_{n-1}$ of honest parties. Then $\mathcal{S}$ reconstructs the secrets $\boldsymbol{\mu_\gamma}$. $\mathcal{S}$ also sends the shares of $[\boldsymbol{\mu_\gamma}]_{n-1}$ of honest parties to P_1.

4. In Step 4, $\mathcal{S}$ simulates π_{Output} as follows. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{Commit}}$ and receives the shares of $([\boldsymbol{a}]_{n-k}, [\boldsymbol{b}]_{n-k}, [\boldsymbol{c}]_{n-1})$ of corrupted parties. $\mathcal{S}$ first simulates the first part of π_{Output}: for each group of output gates with input wires $\boldsymbol{\alpha}$,

– Case 1: P_1 is an honest party.

(a) In Step 3 of π_{Output}, $\mathcal{S}$ computes the shares of $[\boldsymbol{\lambda_\alpha} - \boldsymbol{a}]_{n-1}$ of corrupted parties. $\mathcal{S}$ samples random values as the shares of honest parties and reconstructs the secrets $\boldsymbol{\lambda_\alpha} - \boldsymbol{a}$.
$\mathcal{S}$ receives the shares of $[\boldsymbol{\lambda_\alpha} - \boldsymbol{a}]_{n-1}$ of corrupted parties on behalf P_1. $\mathcal{S}$ uses the shares of corrupted parties he received and the shares of honest parties he generated to compute the secrets $\overline{\boldsymbol{\lambda_\alpha} - \boldsymbol{a}}$.

(b) In Step 3 of π_{Output}, $\mathcal{S}$ computes $\overline{\boldsymbol{v_\alpha} - \boldsymbol{a}} = \overline{\boldsymbol{\mu_\alpha}} + \overline{\boldsymbol{\lambda_\alpha} - \boldsymbol{a}}$. $\mathcal{S}$ also computes $\boldsymbol{v_\alpha} - \boldsymbol{a} = \boldsymbol{\mu_\alpha} + \boldsymbol{\lambda_\alpha} - \boldsymbol{a}$. $\mathcal{S}$ sets $\eta(\boldsymbol{v_\alpha} - \boldsymbol{a}) = \overline{\boldsymbol{v_\alpha} - \boldsymbol{a}} - (\boldsymbol{v_\alpha} - \boldsymbol{a})$. Then $\mathcal{S}$ randomly samples and distributes the shares of $[\overline{\boldsymbol{v_\alpha} - \boldsymbol{a}}]_{2k-2}$ to corrupted parties on behalf of P_1.

(c) In Step 4 of π_{Output}, $\mathcal{S}$ follows the protocol and computes the shares of $\langle \theta_{\alpha_i} \rangle$ and $\langle \theta_{\beta_i} \rangle$ of corrupted parties. Note that $\theta_{\alpha_i} = \Delta \cdot \eta(v_{\alpha_i} - a_i)$.

– Case 2: P_1 is a corrupted party.

(a) In Step 2 of π_{Output}, $\mathcal{S}$ computes the shares of $[\boldsymbol{\lambda_\alpha} - \boldsymbol{a}]_{n-1}$ of corrupted parties. $\mathcal{S}$ samples random values as the shares of honest parties and reconstructs the secrets $\boldsymbol{\lambda_\alpha} - \boldsymbol{a}$.
$\mathcal{S}$ sends the shares of $[\boldsymbol{\lambda_\alpha} - \boldsymbol{a}]_{n-1}$ of honest parties to P_1.

(b) In Step 3 of π_{Output}, $\mathcal{S}$ computes $\boldsymbol{v_\alpha} - \boldsymbol{a} = \boldsymbol{\mu_\alpha} + \boldsymbol{\lambda_\alpha} - \boldsymbol{a}$. $\mathcal{S}$ receives the shares of $[\boldsymbol{v_\alpha} - \boldsymbol{a}]_{2k-2}$. Then $\mathcal{S}$ uses the shares of honest parties to reconstruct the whole sharing and computes the secrets $\overline{\boldsymbol{v_\alpha} - \boldsymbol{a}}$. $\mathcal{S}$ sets $\eta(\boldsymbol{v_\alpha} - \boldsymbol{a}) = \overline{\boldsymbol{v_\alpha} - \boldsymbol{a}} - (\boldsymbol{v_\alpha} - \boldsymbol{a})$

(c) In Step 4 of π_{Mult}, $\mathcal{S}$ follows the protocol and computes the shares of $\langle \theta_{\alpha_i} \rangle$ and $\langle \theta_{\beta_i} \rangle$ of corrupted parties. Note that, when the execution is not marked as `abort` (i.e., all degree-$(k-1)$ packed Shamir sharings distributed by P_1 are valid), $\theta_{\alpha_i} = \Delta \cdot \eta(v_{\alpha_i} - a_i)$.

$\mathcal{S}$ then simulates the second part of π_{Output}. For Step 5, note that $\mathcal{S}$ has generated or learnt the shares of honest parties of all degree-$(k-1)$ packed Shamir sharings distributed by P_1. $\mathcal{S}$ honestly emulates $\mathcal{F}_{\mathsf{Coin}}$ and $\mathcal{F}_{\mathsf{Commit}}$, and honestly follows Step 5. If this execution has been marked as `abort` but all parties accept the check. $\mathcal{S}$ aborts.

For Step 6, $\mathcal{S}$ honestly emulates $\mathcal{F}_{\mathsf{Coin}}$ and $\mathcal{F}_{\mathsf{Commit}}$. $\mathcal{S}$ follows this step and computes the shares of $\langle \theta \rangle$ of corrupted parties. Recall that for all $j \in \{1, \ldots, m\}$, $\mathcal{S}$ has computed some η_j such that θ_j should be equal to $\Delta \cdot \eta_j$. $\mathcal{S}$ computes $\eta = \sum_{j=1}^{m} \chi'_j \cdot \eta_j$ and randomly samples Δ. Then $\mathcal{S}$ computes $\theta = \Delta \cdot \eta$ and randomly samples the shares of $\langle \theta \rangle$ of honest parties based on the secret θ and the shares of corrupted parties. $\mathcal{S}$ follows the rest of steps. If there exists j such that $\eta_j \neq 0$ but all parties accept the check, $\mathcal{S}$ aborts.

$\mathcal{S}$ finally simulates the last part of π_{Output}. For each group of k output gates that belong to some `Client`, let $\boldsymbol{\alpha}$ denote the input wires of these output gates. When `Client` is corrupted,

- $\mathcal{S}$ computes $\boldsymbol{a} = \boldsymbol{v_\alpha} - (\boldsymbol{v_\alpha} - \boldsymbol{a})$. Then $\mathcal{S}$ randomly samples $\boldsymbol{b}$ and computes $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$. Based on the secrets and the shares of corrupted parties, $\mathcal{S}$ randomly samples the shares of $[\boldsymbol{a}]_{n-k}, [\boldsymbol{b}]_{n-k}, [\boldsymbol{c}]_{n-1}$ of honest parties. Finally, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{Commit}}$ and opens the shares of honest parties to `Client`.

When `Client` is honest,

- $\mathcal{S}$ receives the shares of $[\boldsymbol{v_\alpha} - \boldsymbol{a}]_{2k-2}$, $[\boldsymbol{a}]_{n-k}, [\boldsymbol{b}]_{n-k}, [\boldsymbol{c}]_{n-1}$ of corrupted parties. Then $\mathcal{S}$ checks the following:
  - The shares of $[\boldsymbol{v_\alpha} - \boldsymbol{a}]_{2k-2}$ received from corrupted parties are the same as those computed by $\mathcal{S}$ when simulating Step 3 of π_{Output}.
  - For each of $[\boldsymbol{a}]_{n-k}, [\boldsymbol{b}]_{n-k}$, $\mathcal{S}$ defines a degree-$(n-k)$ packed Shamir sharing such that the shares of corrupted parties are the differences between the shares received from corrupted parties and those they should hold, and the shares of honest parties are 0. Then $\mathcal{S}$ checks whether all shares lie on a degree-$(n-k)$ polynomial. If not, $\mathcal{S}$ aborts on behalf of the honest `Client`. Otherwise, $\mathcal{S}$ reconstructs the secrets, denoted by $\eta(\boldsymbol{a}), \eta(\boldsymbol{b})$.
  - For $[\boldsymbol{c}]_{n-1}$, $\mathcal{S}$ defines a degree-$(n-1)$ packed Shamir sharing such that the shares of corrupted parties are the differences between the shares received from corrupted parties and those they should hold, and the shares of honest parties are 0. Then $\mathcal{S}$ reconstructs the secrets, denoted by $\eta(\boldsymbol{c})$.
  - If any of $\eta(\boldsymbol{a}), \eta(\boldsymbol{b}), \eta(\boldsymbol{c})$ is not an all-0 vector, $\mathcal{S}$ aborts on behalf of `Client`.

This completes the description of the simulator $\mathcal{S}$.

Now we use hybrid arguments to prove the security of Π_{Online}.

Hybrid$_0$: In this hybrid, $\mathcal{S}$ honestly follows the protocol. This corresponds to the real world.

Hybrid$_1$: In this hybrid, $\mathcal{S}$ checks the degree-$(k-1)$ packed Shamir sharings distributed by P_1 as described above. If there exists some degree-$(k-1)$ packed Shamir sharing such that the shares of honest parties do not lie on a degree-$(k-1)$ packed Shamir sharing, $\mathcal{S}$ marks the computation as `abort`. $\mathcal{S}$ simulates Step 5 ofπ_{Output} as described above. If $\mathcal{S}$ has marked the computation as `abort` but no honest party aborts in Step 5 of π_{Output}, $\mathcal{S}$ aborts.

Recall that $\chi_1, \ldots, \chi_m$ are random values in $\mathbb{F}$ and $p \geq 2^\kappa$. Thus, if there exists some degree-$(k-1)$ packed Shamir sharing such that the shares of honest parties do not lie on a degree-$(k-1)$ packed Shamir sharing, with overwhelming probability, $[\boldsymbol{x}]_{k-1}$ in Step 5 is not valid. Thus, **Hybrid**$_1$ is statistically close to **Hybrid**$_0$.

Hybrid$_2$: In this hybrid, $\mathcal{F}_{\mathsf{PrepMal}}$ is emulated by $\mathcal{S}$ as described above. Note that the emulation does not generate the shares of honest parties. $\mathcal{S}$ pushes the generation of the shares of honest parties to whenever they are needed. When needed, $\mathcal{S}$ prepares the shares of honest parties as follows:

- $\mathcal{S}$ follows Step 1 of $\mathcal{F}_{\mathsf{PrepMal}}$ to compute a value λ_α for each wire α.
- $\mathcal{S}$ samples a random value as Δ.
- For each group of multiplication gates with input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$ and output wires $\boldsymbol{\gamma}$, $\mathcal{S}$ has generated $\boldsymbol{\lambda_\alpha} - \boldsymbol{a}, \boldsymbol{\lambda_\beta} - \boldsymbol{b}$ and $\boldsymbol{\lambda_\alpha}, \boldsymbol{\lambda_\beta}$. $\mathcal{S}$ computes $\boldsymbol{a}$ and $\boldsymbol{b}$, and then computes $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$.
- $\mathcal{S}$ generates the shares of honest parties following from $\mathcal{F}_{\mathsf{PrepMal}}$.

Observe that the only difference is the generation of $\boldsymbol{a}, \boldsymbol{b}$ for each group of multiplication gates. In **Hybrid**$_1$, $\boldsymbol{a}, \boldsymbol{b}$ are randomly sampled and then $\boldsymbol{\lambda_\alpha} - \boldsymbol{a}, \boldsymbol{\lambda_\beta} - \boldsymbol{b}$ are computed accordingly. In **Hybrid**$_2$, $\mathcal{S}$ first randomly samples $\boldsymbol{\lambda_\alpha} - \boldsymbol{a}, \boldsymbol{\lambda_\beta} - \boldsymbol{b}$ and then computes $\boldsymbol{a}, \boldsymbol{b}$. Note that the distribution of these values remains the same in both hybrids.

Thus, the distribution of **Hybrid**$_2$ is identical to that of **Hybrid**$_1$.

Hybrid$_3$: In this hybrid, when `Client` is corrupted, π_{Input} is simulated by $\mathcal{S}$ as described above. Note that $\mathcal{S}$ essentially does not change the behaviors of honest parties. From the messages $\mathcal{S}$ received from `Client`, $\mathcal{S}$ extracts the inputs of `Client`. The distribution of **Hybrid**$_3$ is identical to that of **Hybrid**$_2$.

Hybrid$_4$: In this hybrid, when `Client` is honest, π_{Input} is simulated by $\mathcal{S}$ as described above. Note that the simulation does not generate the shares of honest parties. $\mathcal{S}$ pushes the generation of the shares of honest parties to whenever they are needed. When needed, $\mathcal{S}$ computes the shares of honest parties by using the inputs of `Client`.

We first show that in Step 3 of π_{Input}, the probability that $\mathcal{S}$ aborts in **Hybrid**$_4$ but not in **Hybrid**$_3$ is negligible.

– When checking the degree of $[\boldsymbol{a}]_{n-k}, [\boldsymbol{b}]_{n-k}$, observe that if corrupted parties use their correct shares, then the whole sharings are valid degree-$(n-k)$ packed Shamir sharings. In **Hybrid**$_3$, $\mathcal{S}$ checks these two sharings directly while in **Hybrid**$_4$, $\mathcal{S}$ checks the difference between the sharing received from all parties and the sharing all parties should hold (in particular, the shares of honest parties in two cases are identical). Thus, $\mathcal{S}$ aborts in **Hybrid**$_4$ if and only if $\mathcal{S}$ aborts in **Hybrid**$_3$.
– When checking whether $\bar{\boldsymbol{c}} = \bar{a} * \bar{b}$, in **Hybrid**$_3$, $\mathcal{S}$ directly checks the relation of these three vectors. In **Hybrid**$_4$, $\mathcal{S}$ first computes the additive errors due to the malicious behaviors of corrupted parties, which are denoted by $\eta(\boldsymbol{a}), \eta(\boldsymbol{b}), \eta(\boldsymbol{c})$. Since $\boldsymbol{a}, \boldsymbol{b}$ are random vectors and $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$, therefore

$$\bar{\boldsymbol{c}} - \bar{a} * \bar{b} = \eta(\boldsymbol{c}) - \eta(\boldsymbol{a}) * \eta(\boldsymbol{b}) - \eta(\boldsymbol{a}) * \boldsymbol{b} - \eta(\boldsymbol{b}) * \boldsymbol{a}.$$

It is not hard to see that, when at least one of $\eta(\boldsymbol{a}), \eta(\boldsymbol{b}), \eta(\boldsymbol{c})$ is not an all-0 vector, the probability that $\bar{\boldsymbol{c}} - \bar{a} * \bar{b} = \mathbf{0}$ is negligible. Thus, the probability that $\bar{\boldsymbol{c}} = \bar{a} * \bar{b}$ but at least one of $\eta(\boldsymbol{a}), \eta(\boldsymbol{b}), \eta(\boldsymbol{c})$ is not an all-0 vector is negligible.

Therefore, the probability that $\mathcal{S}$ aborts in **Hybrid**$_4$ but not in **Hybrid**$_3$ is negligible.

We then show that the messages sent from `Client` to all parties have the same distribution in both hybrids. Since $\boldsymbol{\lambda_\alpha}$ and $\boldsymbol{a}$ are random vectors, $\boldsymbol{v_\alpha} - \boldsymbol{\lambda_\alpha}$ and $\boldsymbol{v_\alpha} - \boldsymbol{a}$ are also random vectors. In **Hybrid**$_3$, $\mathcal{S}$ sends $\boldsymbol{v_\alpha} - \overline{\boldsymbol{\lambda_\alpha}}$ to P_1 and distributes $[\boldsymbol{v_\alpha} - \boldsymbol{a}]_{2k-2}$ to all parties. In **Hybrid**$_4$, $\mathcal{S}$ randomly samples $\boldsymbol{v_\alpha} - \boldsymbol{\lambda_\alpha}$ and $\boldsymbol{v_\alpha} - \boldsymbol{a}$ and then does the same as that in **Hybrid**$_3$.

In summary, **Hybrid**$_4$ is statistically close to **Hybrid**$_3$.

Hybrid$_5$: In this hybrid, $\mathcal{S}$ simulates the behaviors of honest parties when evaluating addition gates. Note that $\mathcal{S}$ essentially follows the protocol. Therefore, the distribution of **Hybrid**$_5$ is identical to that of **Hybrid**$_4$.

Hybrid$_6$: In this hybrid, when P_1 is honest, π_{Mult} is simulated by $\mathcal{S}$ as described above. Note that in Step 2, $\mathcal{S}$ honestly follows the protocol to simulate the behaviors of P_1. In Step 6, $\mathcal{S}$ randomly samples the shares of $[\boldsymbol{\mu_\gamma}]_{n-1}$ of honest parties. In **Hybrid**$_5$, since $[\boldsymbol{\lambda_\gamma}]_{n-1}$ is a random degree-$(n-1)$ packed Shamir sharing and

$$\begin{aligned}[\boldsymbol{\mu_\gamma}]_{n-1} = {} & [\boldsymbol{v_\alpha} - \boldsymbol{a}]_{k-1} * [\boldsymbol{v_\beta} - \boldsymbol{b}]_{k-1} + [\boldsymbol{v_\alpha} - \boldsymbol{a}]_{k-1} * [\boldsymbol{b}]_{n-k} \\ & + [\boldsymbol{v_\beta} - \boldsymbol{b}]_{k-1} * [\boldsymbol{a}]_{n-k} + [\boldsymbol{c}]_{n-1} - [\boldsymbol{\lambda_\gamma}]_{n-1},\end{aligned}$$

$[\boldsymbol{\lambda_\gamma}]_{n-1}$ is a random degree-$(n-1)$ packed Shamir sharing.

Thus, the distribution of **Hybrid**$_6$ is identical to that of **Hybrid**$_5$.

Hybrid$_7$: In this hybrid, when P_1 is corrupted, π_{Mult} is simulated by $\mathcal{S}$ as described above. In Step 6, $\mathcal{S}$ randomly samples the shares of $[\boldsymbol{\mu_\gamma}]_{n-1}$ of honest parties. With the same argument as above, the shares of honest parties have the same distribution as that in **Hybrid**$_6$.

Thus, the distribution of **Hybrid**$_7$ is identical to that of **Hybrid**$_6$.

Hybrid$_8$: In this hybrid, when P_1 is honest, the first part of π_{Output} is simulated by $\mathcal{S}$ as described above. In Step 2, $\mathcal{S}$ samples random values as the shares of $[\boldsymbol{\lambda_\alpha} - \boldsymbol{a}]_{n-1}$ of honest parties. In **Hybrid**$_7$, since $[\boldsymbol{\lambda_\alpha}]_{n-1}$ is a random degree-$(n-1)$ packed Shamir sharing of $\boldsymbol{\lambda_\alpha}$ and $\boldsymbol{a}$ is a random vector, the sharing $[\boldsymbol{\lambda_\alpha} - \boldsymbol{a}]_{n-1} = [\boldsymbol{\lambda_\alpha}]_{n-1} - [\boldsymbol{a}]_{n-k}$ is a random degree-$(n-1)$ packed Shamir sharing. Thus, the shares of $[\boldsymbol{\lambda_\alpha} - \boldsymbol{a}]_{n-1}$ of honest parties have the same distribution in both hybrids.

In Step 3, $\mathcal{S}$ honestly follows the protocol to simulate P_1. Thus, the distribution of **Hybrid**$_8$ is identical to that of **Hybrid**$_7$.

Hybrid$_9$: In this hybrid, when P_1 is corrupted, the first part of π_{Output} is simulated by $\mathcal{S}$ as described above. In Step 2, $\mathcal{S}$ samples random values as the shares of $[\boldsymbol{\lambda_\alpha} - \boldsymbol{a}]_{n-1}$ of honest parties. With the same argument as above, the shares of honest parties have the same distribution as that in **Hybrid**$_8$. Thus, the distribution of **Hybrid**$_9$ is identical to that of **Hybrid**$_8$.

Hybrid$_{10}$: In this hybrid, Step 6 of π_{Output} is simulated by $\mathcal{S}$ as described above. We first argue that the shares of $\langle\theta\rangle$ prepared by $\mathcal{S}$ are computationally indistinguishable from those in **Hybrid**$_9$.

Note that after step 5 of π_{Output}, all degree-$(k-1)$ packed Shamir sharings distributed by P_1 are valid. Therefore, the only attack an adversary can do is to change the secrets of degree-$(k-1)$ packed Shamir sharings (by either instructing corrupted parties to send incorrect shares to P_1 or instructing the corrupted P_1 to distribute incorrect degree-$(k-1)$ packed Shamir sharings). For each $j \in \{1,\dots,m\}$, $\mathcal{S}$ has computed η_j such that $\theta_j = \varDelta \cdot \eta_j$. $\mathcal{S}$ follows the protocol and computes $\eta = \sum_{j=1}^{m} \chi'_j \cdot \eta_j$. Then, $\mathcal{S}$ randomly samples $\varDelta$ and randomly samples the shares of $\langle\theta\rangle$ of honest parties based on the shares of corrupted parties and the secret $\theta = \varDelta \cdot \eta$.

In **Hybrid**$_9$, $\langle\theta\rangle$ has been refreshed by all parties. The refresh step ensures that, given the shares of corrupted parties and the secret, the distribution of the shares of $\langle\theta\rangle$ of honest parties is computationally indistinguishable from that of a random additive sharing of $\theta = \varDelta \cdot \eta$. Thus, the shares of $\langle\theta\rangle$ prepared by $\mathcal{S}$ are computationally indistinguishable from those in **Hybrid**$_9$.

We then show that the probability that $\mathcal{S}$ aborts in **Hybrid**$_{10}$ but not in **Hybrid**$_9$ is negligible. Since $\chi'_1,\dots,\chi'_m$ are random elements in $\mathbb{F}$ and $p \geq 2^\kappa$, if there exists $\eta_j \neq 0$, with overwhelming probability, $\eta \neq 0$. Then $\theta \neq 0$. In Step 6(c), corrupted parties can only change their shares without learning $\varDelta$, which translates to an additive error to the secret θ. Thus, the probability that the secret $\bar{(\theta)} = 0$ is negligible.

Thus, the distribution of **Hybrid**$_{10}$ is computationally indistinguishable from that of **Hybrid**$_9$.

Hybrid$_{11}$: In this hybrid, $\mathcal{S}$ simulates the last part of π_{Output} as described above. We consider the following two cases:

- When `Client` is honest, $\mathcal{S}$ only accepts when corrupted parties use their correct shares of $[\boldsymbol{v_\alpha} - \boldsymbol{a}]_{2k-2}$ and the additive errors to $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}$ are all 0s. We show that, with overwhelming probability, this is also the case in **Hybrid**$_{10}$.
  Regarding $[\boldsymbol{v_\alpha} - \boldsymbol{a}]_{2k-2}$, note that the whole sharing is determined by the shares of honest parties. If corrupted parties do not use their correct shares, then the degree-$(2k-2)$ packed Shamir sharing `Client` received is not valid, and $\mathcal{S}$ aborts in **Hybrid**$_{10}$.
  Regarding the additive errors to $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}$, in **Hybrid**$_{10}$, `Client` checks whether $\overline{\boldsymbol{c}} = \overline{\boldsymbol{a}} * \overline{\boldsymbol{b}}$. It is equivalent to $\eta(\boldsymbol{c}) - \eta(\boldsymbol{a}) * (\eta(\boldsymbol{b}) + \boldsymbol{b}) - \eta(\boldsymbol{b}) * \boldsymbol{a} = \boldsymbol{0}$. Note that corrupted parties should commit their shares before all parties sending $[\boldsymbol{\lambda_\alpha} - \boldsymbol{a}]_{n-1}$ to P_1. At that moment $\boldsymbol{a}, \boldsymbol{b}$ are uniform vectors and unknown to the adversary. Therefore, the probability that at least one of $\eta(\boldsymbol{a}), \eta(\boldsymbol{b}), \eta(\boldsymbol{c})$ is a non-zero vector and $\overline{\boldsymbol{c}} = \overline{\boldsymbol{a}} * \overline{\boldsymbol{b}}$ is negligible.
  In summary, with overwhelming probability, $\mathcal{S}$ also aborts in **Hybrid**$_{10}$.
- When `Client` is corrupted, $\mathcal{S}$ learns the outputs $\boldsymbol{v_\alpha}$ of `Client` from $\mathcal{F}_{\mathsf{MPC}}$. $\mathcal{S}$ computes $\boldsymbol{a}$ from $\boldsymbol{v_\alpha}, \boldsymbol{v_\alpha} - \boldsymbol{a}$, randomly samples $\boldsymbol{b}$, and computes $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$. Then $\mathcal{S}$ randomly samples the shares of honest parties based on the secrets and the shares of corrupted parties. Note that the shares of honest parties are identically distributed to those in **Hybrid**$_{10}$.

Thus, the distribution of **Hybrid**$_{11}$ is statistically close to **Hybrid**$_{10}$. Note that **Hybrid**$_{11}$ is the execution in the ideal world. We have that the distribution of **Hybrid**$_{11}$ is computationally indistinguishable from the distribution of **Hybrid**$_0$, the execution in the real world. Therefore, protocol $\varPi_{\mathsf{Online}}$ securely computes the ideal functionality $\mathcal{F}_{\mathsf{MPC}}$ in the $\{\mathcal{F}_{\mathsf{PrepMal}}, \mathcal{F}_{\mathsf{Coin}}, \mathcal{F}_{\mathsf{Commit}}\}$-hybrid model against a fully malicious adversary who controls t corrupted parties and up to c clients.

C Proof of Lemma 1

Lemma 1. *Protocol $\varPi_{\mathsf{PrepMal}}$ securely computes $\mathcal{F}_{\mathsf{PrepMal}}$ in the $\mathcal{F}_{\mathsf{PrepIndMal}}$-hybrid model against a malicious adversary who controls t out of n parties.*

Proof. We will construct a simulator $\mathcal{S}$ to simulate the behaviors of honest parties. Let $\mathcal{C}orr$ denote the set of corrupted parties and $\mathcal{H}$ denote the set of honest parties. The simulator $\mathcal{S}$ works as follows.

$\mathcal{S}$ emulates the ideal functionality $\mathcal{F}_{\mathsf{PrepIndMal}}$ and receives the shares of corrupted parties. For Step 2, Step 3, and Step 6(b) in $\mathcal{F}_{\mathsf{PrepMal}}$, $\mathcal{S}$ provides the shares of corrupted parties to $\mathcal{F}_{\mathsf{PrepMal}}$.

Recall that from Step 2 in $\mathcal{F}_{\mathsf{PrepIndMal}}$, for each output wire α of an input gate or a multiplication gate, $\mathcal{S}$ receives the shares of $([\lambda_\alpha \cdot \mathbf{1}]_{n-k}, \langle \Delta \cdot \lambda_\alpha \rangle)$ of corrupted parties. For each wire α in the circuit, $\mathcal{S}$ computes the shares of $([\lambda_\alpha \cdot \mathbf{1}]_{n-k}, \langle \Delta \cdot \lambda_\alpha \rangle)$ of corrupted parties by following Step 1 in $\mathcal{F}_{\mathsf{PrepMal}}$.

Then to simulate Step 5 and Step 6(a) in $\mathcal{F}_{\mathsf{PrepMal}}$, $\mathcal{S}$ follows the protocol to compute corrupted parties' shares of

$$[\boldsymbol{\lambda_\alpha}]_{n-1} = \left(\sum_{i=1}^{k} \boldsymbol{e}_i * [\lambda_{\alpha_i} \cdot \mathbf{1}]_{n-k} \right) + [\boldsymbol{o}]_{n-1}.$$

$\mathcal{S}$ provides the shares of corrupted parties to $\mathcal{F}_{\mathsf{PrepMal}}$.

As for Step 4 in $\mathcal{F}_{\mathsf{PrepMal}}$, $\mathcal{S}$ first follows the protocol to compute corrupted parties' shares of

$$[\boldsymbol{\lambda_\alpha} - \boldsymbol{a}]_{n-1} = \left(\sum_{i=1}^{k} \boldsymbol{e}_i * [\lambda_{\alpha_i} \cdot \mathbf{1}]_{n-k} \right) - [\boldsymbol{a}]_{n-k} + [\boldsymbol{o}^{(1)}]_{n-1}.$$

Depending on whether P_1 is honest, there are two cases.

- If P_1 is honest, $\mathcal{S}$ receives the shares of $[\boldsymbol{\lambda_\alpha} - \boldsymbol{a}]_{n-1}$ from corrupted parties. Then $\mathcal{S}$ sets the shares of $[\boldsymbol{\delta_\alpha}]_{n-1}$ as follows:
  - For each corrupted party P_i, P_i's share is set to be the difference between the share of $[\boldsymbol{\lambda_\alpha} - \boldsymbol{a}]_{n-1}$ received from P_i and the share computed by $\mathcal{S}$.
  - For each honest party P_i, P_i's share is set to be $\mathbf{0}$.

  $\mathcal{S}$ reconstructs the secrets $\boldsymbol{\delta_\alpha}$ and sends the errors to $\mathcal{F}_{\mathsf{PrepMal}}$.
- If P_1 is corrupted, $\mathcal{S}$ sends $\boldsymbol{\delta_\alpha} = \mathbf{0}$ to $\mathcal{F}_{\mathsf{PrepMal}}$ and receives $\boldsymbol{\lambda_\alpha} - \boldsymbol{a}$. Then $\mathcal{S}$ randomly samples the shares of $[\boldsymbol{\lambda_\alpha} - \boldsymbol{a}]_{n-1}$ of honest parties based on the shares of corrupted parties and the secrets $\boldsymbol{\lambda_\alpha} - \boldsymbol{a}$. Finally, $\mathcal{S}$ sends the shares of $[\boldsymbol{\lambda_\alpha} - \boldsymbol{a}]_{n-1}$ of honest parties to P_1.

Note that $\mathcal{S}$ has computed the shares of $\{\langle \Delta \cdot \lambda_{\alpha_i} \rangle\}_{i=1}^{k}$ of corrupted parties. And $\mathcal{S}$ also receives the shares of $\{\langle \Delta \cdot a_i \rangle\}_{i=1}^{k}$ of corrupted parties when emulating $\mathcal{F}_{\mathsf{PrepIndMal}}$. $\mathcal{S}$ follows the protocol to compute the shares of $\{\langle \Delta \cdot (\lambda_{\alpha_i} - a_i) \rangle\}_{i=1}^{k}$ of corrupted parties and then provides them to $\mathcal{F}_{\mathsf{PrepMal}}$.

This completes the description of the simulator $\mathcal{S}$.

Now we use hybrid arguments to prove the security of Π_{PrepMal}.

Hybrid$_0$: In this hybrid, $\mathcal{S}$ honestly follows the protocol. This corresponds to the real world.

Hybrid$_1$: In this hybrid, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{PrepIndMal}}$ and receives the shares of corrupted parties. Regarding the shares of honest parties, when needed, $\mathcal{S}$ follows $\mathcal{F}_{\mathsf{PrepIndMal}}$ to generate the shares of honest parties. The distribution of **Hybrid**$_1$ is identical to that of **Hybrid**$_0$.

Hybrid$_2$: In this hybrid, $\mathcal{S}$ uses corrupted parties' shares of $([\lambda_\alpha \cdot \mathbf{1}]_{n-k}, \langle \Delta \cdot \lambda_\alpha \rangle)$ for each output wire α of an input gate or a multiplication gate to compute their shares of $([\lambda_\alpha \cdot \mathbf{1}]_{n-k}, \langle \Delta \cdot \lambda_\alpha \rangle)$ for *every* wire α in the circuit.

Then $\mathcal{S}$ computes the shares of corrupted parties in Step 5 and Step 6(a) in $\mathcal{F}_{\mathsf{PrepMal}}$. Observe that $\mathcal{S}$ does not change the behaviors of honest parties and just follows the protocol to compute the shares of corrupted parties. Therefore, the distribution of **Hybrid**$_2$ is identical to that of **Hybrid**$_1$.

Hybrid$_3$: In this hybrid, $\mathcal{S}$ simulates Step 4 as described above. We consider two cases:

- Case 1: P_1 is honest. In this case, honest parties do not need to send any message to corrupted parties. We only need to focus on the output of P_1. In **Hybrid**$_2$, P_1 reconstructs the secrets of $[\boldsymbol{\lambda_\alpha} - \boldsymbol{a}]_{n-1}$ from the shares received from all parties. In **Hybrid**$_3$, $\mathcal{S}$ computes $[\boldsymbol{\delta_\alpha}]_{n-1}$ which is equal to the difference between the sharing where the shares of corrupted parties are those received from them and the sharing where the shares of corrupted parties are those they should hold. Therefore, the secrets $\boldsymbol{\delta_\alpha}$ correspond to the additive errors due to the malicious behaviors of corrupted parties. $\mathcal{S}$ passes $\boldsymbol{\delta_\alpha}$ to $\mathcal{F}_{\mathsf{PrepMal}}$, who will add the errors to $\boldsymbol{\lambda_\alpha} - \boldsymbol{a}$ and return to the honest party P_1. Then the output of P_1 is identically distributed in both worlds.
- Case 2: P_1 is corrupted. We only need to show that the messages sent from honest parties to corrupted parties are identically distributed. In **Hybrid**$_2$, these messages are the shares of $[\boldsymbol{\lambda_\alpha} - \boldsymbol{a}]_{n-1}$ of honest parties. Note that they are uniformly distributed given the shares of corrupted parties and the secrets $\boldsymbol{\lambda_\alpha}$ since the whole sharing is masked by a random degree-$(n-1)$ packed Shamir sharing of $\mathbf{0}$, $[\boldsymbol{o}^{(1)}]_{n-1}$. In **Hybrid**$_3$, $\mathcal{S}$ computes the shares of $[\boldsymbol{\lambda_\alpha} - \boldsymbol{a}]_{n-1}$

of corrupted parties and receives the secrets $\boldsymbol{\lambda_\alpha} - \boldsymbol{a}$ from $\mathcal{F}_{\mathsf{PrepMal}}$. Then $\mathcal{S}$ uniformly samples the shares of $[\boldsymbol{\lambda_\alpha} - \boldsymbol{a}]_{n-1}$ of honest parties based on the shares of corrupted parties and the secrets. Therefore, the messages sent from honest parties to corrupted parties are identically distributed.

In both cases, $\mathcal{S}$ finally follows the protocol to compute the shares of $\{\langle \Delta \cdot (\lambda_{\alpha_i} - a_i)\rangle\}_{i=1}^k$ of corrupted parties and then provides them to $\mathcal{F}_{\mathsf{PrepMal}}$. In **Hybrid**$_2$, the additive sharings $\{\langle \Delta \cdot (\lambda_{\alpha_i} - a_i)\rangle\}_{i=1}^k$ are locally refreshed by all parties. The refresh step ensures that, given the shares of corrupted parties and the secret, the distribution of the shares of honest parties is computationally indistinguishable from that of a random additive sharing. Thus, the shares of honest parties in **Hybrid**$_2$ are computationally indistinguishable from those generated by $\mathcal{F}_{\mathsf{PrepMal}}$. Thus, the distribution of **Hybrid**$_3$ is computationally indistinguishable to that of **Hybrid**$_2$.

Observe that **Hybrid**$_3$ is the execution in the ideal world. Therefore, Π_{PrepMal} securely computes $\mathcal{F}_{\mathsf{PrepMal}}$ in the $\mathcal{F}_{\mathsf{PrepIndMal}}$-hybrid model against a malicious adversary who controls t out of n parties. $\square$

D Missing Details for the Circuit-Independent Offline Phase

D.1 Preparing Random Sharings for a Given Linear Secret Sharing Scheme

Let Σ be a linear secret sharing scheme in $\mathbb{F}$. To prepare random Σ-sharings, we follow a similar approach to that in [DN07] and describe the procedure in π_{RandSh}.

Procedure 5: $\pi_{\mathsf{Randsh}}(\Sigma)$

1. All parties agree on a Vandermonde matrix $\boldsymbol{M}^{\mathrm{T}}$ of size $n \times (n-t)$ in $\mathbb{F}$.
2. Each party P_i randomly samples a random Σ-sharing $\boldsymbol{S}^{(i)}$ and distributes the shares to other parties.
3. All parties locally compute $(\boldsymbol{R}^{(1)}, \ldots, \boldsymbol{R}^{(n-t)})^{\mathrm{T}} = \boldsymbol{M}(\boldsymbol{S}^{(1)}, \ldots, \boldsymbol{S}^{(n)})^{\mathrm{T}}$ and output $(\boldsymbol{R}^{(1)}, \ldots, \boldsymbol{R}^{(n-t)})$.

Note that each output sharing $\boldsymbol{R}^{(i)}$ is a linear combination of $\{\boldsymbol{S}^{(j)}\}_{j=1}^n$. The correctness follows from the fact that Σ is a linear secret sharing scheme. Thus, all parties will output valid Σ-sharings in the above approach. The security follows from the fact that any sub-matrix of size $(n-t) \cdot (n-t)$ of an $n \times (n-t)$ Vandermonde matrix is invertible. Therefore, given the random sharings prepared by corrupted parties, there is a one-to-one map from the random sharings prepared by honest parties and the output sharings. Thus, the output sharings are also random.

Using Pseudorandom Generators to Reduce the Communication. We adapt the approach of using pseudorandom generators (PRGs) [LN17,NV18,BBC+19,GLO+21], [GSY21,RS22] to reduce the communication complexity when preparing random sharings. Concretely, each pair of parties agree on a random PRG seed at the beginning of the protocol. Whenever a party needs to send a random field element to the other party, these two parties evaluate the PRG and obtain a common random field element.

In the following we discuss how to prepare the random sharings we need in our construction and analyze the communication complexity.

Preparing Random Sharings in the Form of $[r \cdot \mathbf{1}]_{n-k}$. Considering the following generating process.

1. The dealer samples a random value r and sets the secret to be $r \cdot \mathbf{1}$.
2. The dealer sets the shares of the first t parties to be random values.
3. For the underlying polynomial of $[r \cdot \mathbf{1}]_{n-k}$, $t + k = n - k + 1$ evaluation points are fixed. The dealer reconstructs the whole sharing by using the shares of the first t parties and the secrets $r \cdot \mathbf{1}$.
4. Finally, the dealer distributes the shares to all other parties.

Observe that except the shares of $[r \cdot \mathbf{1}]_{n-k}$ of the last $(n-t)$ parties, the rest of shares are uniform elements. Therefore, when using PRG, the dealer only needs to send $(n-t)$ elements to distribute $[r \cdot \mathbf{1}]_{n-k}$. The communication complexity of preparing random sharings in the form of $[r \cdot \mathbf{1}]_{n-k}$ using the above approach is $(n-t) \cdot n/(n-t) = n$ elements per sharing.

Preparing Random Degree-$(n-1)$ *Packed Shamir Sharings of* $\mathbf{0} \in \mathbb{F}_p^k$. Considering the following generating process.

1. The dealer sets the secret to be $\mathbf{0}$.
2. The dealer sets the shares of the first $n-k$ parties to be random values.
3. For the underlying polynomial of $[\mathbf{0}]_{n-1}$, $n-k+k=n$ evaluation points are fixed. The dealer reconstructs the whole sharing by using the shares of the first $n-k$ parties and the secrets $\mathbf{0}$.
4. Finally, the dealer distributes the shares to all other parties.

Observe that except the shares of $[\mathbf{0}]_{n-1}$ of the last k parties, the rest of shares are uniform elements. Therefore, when using PRG, the dealer only needs to send k elements to distribute $[\mathbf{0}]_{n-k}$. The communication complexity of preparing random packed Shamir sharings of $\mathbf{0}$ using the above approach is $k \cdot n/(n-t) \approx n/2$ elements per sharing. Here we use the fact that $k=(n-t+1)/2$.

Preparing Random Additive Sharings of 0. In our construction, we will need to refresh an additive sharing by adding a random additive sharing of 0.

Consider the following generating process:

1. The dealer first samples random values as the shares of $\langle 0 \rangle$ of the rest $n-1$ parties.
2. The dealer locally computes his own share such that all shares together form an additive sharing of 0.
3. The dealer distributes the shares to all other parties.

Relying on PRGs, the dealer does not need to send any message to other parties. Thus, all parties can *locally* prepare additive sharings of 0. This also means that parties can locally refresh their additive sharings.

Preparing Random degree-$(n-k)$ *Packed Shamir Sharings* $[\boldsymbol{r}]_{n-k}$. In our construction, we will use a random degree-$(n-k)$ packed Shamir sharing to transform k additive sharings to a degree-$(n-k)$ packed Shamir sharing. Consider the following generating process:

1. The dealer sets the shares of $[\boldsymbol{r}]_{n-k}$ of the first $n-k+1$ parties to be random values.
2. For the underlying polynomial of $[\boldsymbol{r}]_{n-k}$, $n-k+1$ evaluation points are fixed. The dealer reconstructs the whole sharing by using the shares of the first $n-k+1$ parties.
3. Finally, the dealer distributes the shares to all other parties.

Observe that except the shares of $[\boldsymbol{r}]_{n-k}$ of the last $k-1$ parties, the rest of shares are uniform elements. Therefore, when using PRG, the dealer only needs to send $k-1$ elements to distribute a random degree-$(n-k)$ packed Shamir sharing $[\boldsymbol{r}]_{n-k}$. The communication complexity of preparing random degree-$(n-k)$ packed Shamir sharings using the above approach is $(k-1) \cdot n/(n-t) = n/2$ elements per sharing. Here we use the fact that $k=(n-t+1)/2$.

Preparing Random Sharings in the Form of $([\boldsymbol{r}]_{n-k}, [\boldsymbol{r}]_{n-1})$. In our construction, we use a pair of random sharings $([\boldsymbol{r}]_{n-k}, [\boldsymbol{r}]_{n-1})$ to reduce the degree of a degree-$(n-1)$ packed Shamir sharing. Consider the following generating process:

1. The dealer first samples random values as the shares of $[\boldsymbol{r}]_{n-1}$ for all parties.
2. The dealer reconstructs the secret $\boldsymbol{r}$.
3. The dealer sets the shares of $[\boldsymbol{r}]_{n-k}$ of the first t parties to be random values.
4. For the underlying polynomial of $[\boldsymbol{r}]_{n-k}$, $t+k=n-k+1$ evaluation points are fixed. The dealer reconstructs the whole sharing by using the shares of the first t parties and the secrets $\boldsymbol{r}$.
5. Finally, the dealer distributes the shares to all other parties.

Observe that except the shares of $[\boldsymbol{r}]_{n-k}$ of the last $(n-t)$ parties, the rest of shares are uniform elements. Therefore, when using PRG, the dealer only needs to send $n-t$ elements to distribute a pair of random sharings $([\boldsymbol{r}]_{n-k}, [\boldsymbol{r}]_{n-1})$. The communication complexity of preparing random sharings in the form of $([\boldsymbol{r}]_{n-k}, [\boldsymbol{r}]_{n-1})$ is $(n-t) \cdot n/(n-t) = n$ elements per sharing.

Preparing Random Sharings in the Form of $\{[r|_i]_t\}_{i=1}^k$. In our construction, we will only use one tuple of such random sharings to transform the MAC key to the desired form. The cost of preparing only one tuple of such random sharings does not affect the efficiency of the whole construction. Thus, we do not try to optimize this step. All parties simply follow the above approach to prepare random sharings in the form of $\{[r|_i]_t\}_{i=1}^k$.

D.2 Preparing Random Packed Triples with Authentications

Here we present details of the building blocks in Section 6 for the generation of random packed triples with authentication, which is instantiated by π_{Triple} (Procedure 3 in page 20). There, we make use of the following two functionalities $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$, $\mathcal{F}_{\mathsf{nVOLE}}$ from [RS22].

Functionality 4: $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$

Parameters: An expansion function $\texttt{Expand} : S \to \mathbb{F}_p^m$ with seed space S and output length m. The functionality runs between parties P_A and P_B.

On receiving s_a from P_A and s_b from P_B, where $s_a, s_b \in S$:

1. Computing $\boldsymbol{u} = \texttt{Expand}(s_a), \boldsymbol{x} = \texttt{Expand}(s_b)$ and sample $\boldsymbol{v} \leftarrow \mathbb{F}_p^m$.
2. Output $\boldsymbol{w} = \boldsymbol{u} * \boldsymbol{x} - \boldsymbol{v}$ to P_A and $\boldsymbol{v}$ to P_B.

Corrupted Party: If P_B is corrupted, $\boldsymbol{v}$ may be chosen by $\mathcal{A}$. If P_A is corrupted, $\boldsymbol{w}$ can be chosen by $\mathcal{A}$ (and $\boldsymbol{v}$ is computed by $\boldsymbol{u} * \boldsymbol{x} - \boldsymbol{w}$).

Functionality 5: $\mathcal{F}_{\mathsf{nVOLE}}$

Parameters: An expansion function $\texttt{Expand} : S \to \mathbb{F}_p^m$ with seed space S and output length m. The functionality runs between parties $P_1, P_2, \ldots, P_n$.

Initialize: On receiving $\texttt{Init}$ from P_i for $i \in \{1, \ldots, n\}$, sample $\Delta^i \leftarrow \mathbb{F}$, send it to P_i, and ignore all subsequent $\texttt{Init}$ command from P_i.
Extend: On receiving $\texttt{Extend}$ from every $P_i \in \{P_1, \ldots, P_n\}$:

1. Sample seed $\texttt{seed}^i \leftarrow S$ for all P_i.
2. Compute $\boldsymbol{u}^i = \texttt{Expand}(\texttt{seed}^i)$.
3. Sample $\boldsymbol{v}_i^j \leftarrow \mathbb{F}_p^m$ for all P_i and $j \neq i$. Retrieve Δ^j and compute $\boldsymbol{w}_j^i = \boldsymbol{u}^i \cdot \Delta^j - \boldsymbol{v}_i^j$.
4. Output $(\texttt{seed}^i, (\boldsymbol{w}_j^i, \boldsymbol{v}_j^i)_{j \neq i})$ to P_i for all $P_i \in \{P_1, \ldots, P_n\}$.

Corrupted Party: A corrupted party P_i can choose Δ^i and $\texttt{seed}^i$. It can also choose $\boldsymbol{w}_j^i$ (and $\boldsymbol{v}_i^j$ is computed by $\boldsymbol{u}^i \cdot \Delta^j - \boldsymbol{w}_j^i$) and $\boldsymbol{v}_j^i$.
Global Key Query: If P_i is corrupted, receive $(\texttt{query}, \boldsymbol{\Delta}')$ from $\mathcal{A}$ with $\boldsymbol{\Delta}' \in \mathbb{F}_p^n$. If $\boldsymbol{\Delta}' = \boldsymbol{\Delta}$, where $\boldsymbol{\Delta} = (\Delta^1, \ldots, \Delta^n)$, send $\texttt{success}$ to P_i and ignore any subsequent global key query. Else, send $(\texttt{abort}, \boldsymbol{\Delta})$ to P_i, $\texttt{abort}$ to P_j and abort.

In [RS22], Rachuri and Scholl use these two functionalities to prepare authenticated Beaver triples in the form of *additive sharings*. Concretely,

1. All parties first use $\mathcal{F}_{\mathsf{nVOLE}}$ to prepare authenticated random additive sharings. In particular,
   - $\boldsymbol{\Delta} = (\Delta^1, \ldots, \Delta^n)$ is viewed as an additive sharing of the MAC key $\Delta = \sum_{i=1}^n \Delta^i$.
   - For each party P_i, $\boldsymbol{u}^i = \texttt{Expand}(\texttt{seed}^i)$ are the additive shares held by P_i.
   - Since for every P_i, P_j, they together hold an additive sharing of $\boldsymbol{u}^i \cdot \Delta^j$, all parties can locally transform them to additive sharings such that the secrets are $(\sum_{i=1}^n \boldsymbol{u}^i) \cdot \Delta$, which are interpreted as the MACs of the additive sharings $(\boldsymbol{u}^1, \ldots, \boldsymbol{u}^n)$.
2. After preparing two vectors of additive sharings, say $\langle \boldsymbol{a} \rangle, \langle \boldsymbol{b} \rangle$ together with their MACs, every ordered pair of parties invoke $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ to compute additive sharings of $\boldsymbol{a}^i * \boldsymbol{b}^j$. Then, all parties can locally transform them to additive sharings of secrets $\boldsymbol{a} * \boldsymbol{b}$, which are interpreted as $\langle \boldsymbol{c} \rangle$. Note that the MACs of $\langle \boldsymbol{c} \rangle$ are *not* computed in this step, which we will discuss later.

To use these two building blocks in our protocol, we encounter the following two problems: First, the natural output of these two building blocks corresponds to authenticated Beaver triples in the form of additive sharings. However, our online protocol requires to prepare *packed* Beaver triples. Second, the MACs of $\langle \boldsymbol{c} \rangle$ are NOT prepared in the preprocessing phase. Instead, Rachuri and Scholl compute the MACs of $\langle \boldsymbol{c} \rangle$ in the online phase. Ideally, we want to have them prepared in the preprocessing phase rather than the online phase.

To address these two problems, a straightforward solution is to transform the additive sharings to packed Shamir sharings and move the computation of the MACs of $\langle \boldsymbol{c} \rangle$ to the preprocessing phase. However, doing it directly would incur a large communication complexity in the preprocessing phase:

- Computing the MACs of $\langle \boldsymbol{c} \rangle$ requires to communicate $2 \cdot n$ elements per Beaver triple.
- As we will discuss later, transforming a group of k additive sharings to a packed Shamir sharing would require to communicate $k \cdot n$ elements. Since we need to transform each additive sharing and each MAC sharing in the Beaver triples, this step costs $6 \cdot k \cdot n$ elements.
- As we will mention later, the Beaver triples prepared by $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ may not be correct. In [RS22], each Beaver triple is verified by another (possibly incorrect) Beaver triple. The verification costs another $6 \cdot n$ elements per Beaver triple.

Note that to obtain a packed Beaver triple with authentications, the first step and the third step needs to be done for k authenticated Beaver triples in the form of additive sharings. Summing them up, the communication complexity in the preprocessing phase would be as large as $14 \cdot k \cdot n$ elements per packed Beaver triple. In our work, instead, we prepare the desired packed Beaver triples with an amortized cost of $4 \cdot k \cdot n$ elements.

Obtaining Packed Shamir Sharings with Authentications. Recall that all parties can prepare authenticated random additive sharings by using $\mathcal{F}_{\mathsf{nVOLE}}$. The output from $\mathcal{F}_{\mathsf{nVOLE}}$ is used as the first two sharings in the authenticated Beaver triples in [RS22]. Since our construction needs to prepare authenticated random packed Shamir sharings, instead of first computing the authenticated random additive sharings and then transforming them to packed Shamir sharings, we consider to generate authenticated random packed Shamir sharings directly from $\mathcal{F}_{\mathsf{nVOLE}}$.

The main observation is that the shares of a random degree-$(n-1)$ packed Shamir sharing are uniformly distributed. This is because a random degree-$(n-1)$ packed Shamir sharing corresponds to a random degree-$(n-1)$ polynomial, which satisfies that any n evaluations are uniformly distributed. On the other hand, the shares of a random additive sharing are also uniformly distributed. Thus, we may naturally view the random additive sharings prepared in $\mathcal{F}_{\mathsf{nVOLE}}$ as degree-$(n-1)$ packed Shamir sharings. Concretely, for each random additive sharing $(u^1, \ldots, u^n)$, let $\boldsymbol{u}$ denote the secrets of the degree-$(n-1)$ packed Shamir sharing when the shares are $(u^1, \ldots, u^n)$. Then we may view that all parties hold the packed Shamir sharing $[\boldsymbol{u}]_{n-1}$.

To obtain a degree-$(n-k)$ packed Shamir sharing of $\boldsymbol{u}$, we will perform a sharing transformation with the help of a pair of random packed Shamir sharings $([\boldsymbol{r}]_{n-k}, [\boldsymbol{r}]_{n-1})$ as described in $\pi_{\mathsf{DegReduce}}$. This transformation requires to communicate $3n$ elements (including the cost of $([\boldsymbol{r}]_{n-k}, [\boldsymbol{r}]_{n-1})$).

Procedure 6: $\pi_{\mathsf{DegReduce}}$

Input: All parties hold a degree-$(n-1)$ packed Shamir sharing $[\boldsymbol{u}]_{n-1}$ and a pair of random sharings $([\boldsymbol{r}]_{n-k}, [\boldsymbol{r}]_{n-1})$.

1. All parties locally compute $[\boldsymbol{u}+\boldsymbol{r}]_{n-1} = [\boldsymbol{u}]_{n-1} + [\boldsymbol{r}]_{n-1}$ and send their shares to P_1.
2. P_1 locally reconstructs $\boldsymbol{u}+\boldsymbol{r}$ and reshares the secrets by $[\boldsymbol{u}+\boldsymbol{r}]_{2k-2}$.
3. All parties locally compute $[\boldsymbol{u}]_{n-k} = [\boldsymbol{u}+\boldsymbol{r}]_{2k-2} - [\boldsymbol{r}]_{n-k}$.

Now the problem is to prepare the MACs for $\boldsymbol{u}$. We observe that in $\mathcal{F}_{\mathsf{nVOLE}}$, for every ordered pair of parties (P_i, P_j), P_i, P_j together hold an additive sharing of $u^i \cdot \Delta^j$. Since each secret u_ℓ in $\boldsymbol{u}$ is a linear combination of $(u^1, \ldots, u^n)$, all parties can locally compute an additive sharing of $\Delta \cdot u_\ell$. To obtain the MACs $[\Delta \cdot \boldsymbol{u}]_{n-k}$, we will perform a sharing transformation with the help of a random degree-$(n-k)$ packed Shamir sharing $[\boldsymbol{r}]_{n-k}$ as described in π_{AddTran}.

Procedure 7: π_{AddTran}

Input: All parties hold k additive sharings $\{\langle \varDelta \cdot u_\ell \rangle\}_{\ell=1}^{k}$ and a random sharing $[\boldsymbol{r}]_{n-k}$.

1. For all $\ell \in \{1, \dots, k\}$, all parties locally transform $[\boldsymbol{r}]_{n-k}$ to an additive sharing $\langle r_\ell \rangle$ and locally refresh the obtained additive sharing.
2. For all $\ell \in \{1, \dots, k\}$, all parties locally compute $\langle \varDelta \cdot u_\ell + r_\ell \rangle = \langle \varDelta \cdot u_\ell \rangle + \langle r_\ell \rangle$ and send their shares to P_1.
3. For all $\ell \in \{1, \dots, k\}$, P_1 locally reconstructs $\varDelta \cdot u_\ell + r_\ell$. Then P_1 reshares the secrets by $[\varDelta \cdot \boldsymbol{u} + \boldsymbol{r}]_{2k-2}$.
4. All parties locally compute $[\varDelta \cdot \boldsymbol{u}]_{n-k} = [\varDelta \cdot \boldsymbol{u} + \boldsymbol{r}]_{2k-2} - [\boldsymbol{r}]_{n-k}$.

In summary, to obtain packed Shamir sharings with authentications, we interpret the output of $\mathcal{F}_{\mathsf{nVOLE}}$ as packed Shamir sharings, and apply sharing transformations to reduce the degree and obtain packed Shamir sharings for the MACs. It is worth to note that *our approach also reduces the output size of* $\mathcal{F}_{\mathsf{nVOLE}}$: In [RS22], each additive sharing output by $\mathcal{F}_{\mathsf{nVOLE}}$ is only used to store one secret. In our approach, each additive sharing output by $\mathcal{F}_{\mathsf{nVOLE}}$ is interpreted as a packed Shamir sharing which stores k secrets. Effectively, we reduce the output size of $\mathcal{F}_{\mathsf{nVOLE}}$ by a factor of k.

Communication complexity of Procedure $\pi_{\mathsf{DegReduce}}$. The communication involved in $\pi_{\mathsf{DegReduce}}$ is the following:

- (Step 1) $n-1$ elements from the parties to P_1
- (Step 2) $n-(k-1)$ sharings from P_1 to the parties (for a degree-$2(k-1)$ sharing of a k-dimensional vector, $k-1$ shares can be set to zero)

This leads to a total of $2n-k$.

Communication complexity of Procedure π_{AddTran}. In π_{AddTran} the communication is the following:

- (Step 2) $k \cdot (n-1)$ shares from the parties to P_1,
- (Step 3) $n-(k-1)$ sharings from P_1 to the parties,

The total is then $k \cdot (n-2) + n + 1$.

Obtaining Shamir Sharings of the Authentication Key. In $\mathcal{F}_{\mathsf{PrepIndMal}}$, we want to obtain k degree-t Shamir sharings $\{[\varDelta|_i]_t\}_{i=1}^{k}$. Note that all parties hold an additive sharing $\langle \varDelta \rangle$ received from $\mathcal{F}_{\mathsf{nVOLE}}$. We transform $\langle \varDelta \rangle$ to $\{[\varDelta|_i]_t\}_{i=1}^{k}$ by using a tuple of random sharings $\{[r|_i]_t\}_{i=1}^{k}$ as described in π_{MACKey}. Since π_{MACKey} is only invoked once, the cost of π_{MACKey} does not affect the overall communication complexity.

Procedure 8: π_{MACKey}

Input: All parties hold an additive sharing $\langle \varDelta \rangle$ and a tuple of random sharings $\{[r|_i]_t\}_{i=1}^{k}$.

1. All parties locally transform $[r|_1]_t$ to an additive sharing $\langle r \rangle$ and locally refresh the obtained additive sharing.
2. All parties locally compute $\langle \varDelta + r \rangle = \langle \varDelta \rangle + \langle r \rangle$ and send their shares to P_1.
3. P_1 reconstructs the secret $\varDelta + r$ and reshares the secret by $[(\varDelta + r) \cdot \mathbf{1}]_{2k-2}$. Note that $[(\varDelta + r) \cdot \mathbf{1}]_{2k-2}$ can be viewed as $[\varDelta + r|_i]_{2k-2}$ for all $i \in \{1, \dots, k\}$.
4. For all $i \in \{1, \dots, k\}$, all parties locally compute $[\varDelta|_i]_t = [(\varDelta + r) \cdot \mathbf{1}]_{2k-2} - [r|_i]_t$.

Obtaining Packed Beaver Triples with Authentications. Recall that our procedure for triple generation is π_{Triple} (Procedure 3 in page 20). As [RS22], we continue to use $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ to obtain the third sharing in a Beaver triple. After preparing two random packed Shamir sharings $[\boldsymbol{a}]_{n-1}, [\boldsymbol{b}]_{n-1}$, every ordered pair of parties (P_i, P_j) invoke $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ to compute $a^i \cdot b^j$, where a_i is the i-th share of $[\boldsymbol{a}]_{n-1}$ and b^j is the j-th share of $[\boldsymbol{b}]_{n-1}$. Note that this step is *identical* to that in [RS22]: we only interpret $(a^1, \dots, a^n), (b^1, \dots, b^n)$ as packed Shamir sharings while they are interpreted as additive sharings in [RS22].

From $\{a^i \cdot b^j\}_{i,j}$, all parties can locally compute an additive sharing of each $c_\ell = a_\ell \cdot b_\ell$ for all $\ell \in \{1, \dots, k\}$. To obtain a packed Shamir sharing of $\boldsymbol{c}$ with authentications, we first prepare a random degree-$(n-1)$ packed Shamir sharing with authentications using $\mathcal{F}_{\mathsf{nVOLE}}$ as described above. This time we keep the MAC part in the form of additive sharings, i.e., $([\boldsymbol{r}]_{n-1}, \{\langle \Delta \cdot r_\ell \rangle\}_{\ell=1}^k)$. The missing block for the description of π_{Triple} is the procedure π_{Auth} below, which produces precisely these random sharings.

Procedure 9: π_{Auth}

Input: All parties hold k additive sharings $\{\langle c_\ell \rangle\}_{\ell=1}^k$ and an authenticated random sharing $([\boldsymbol{r}]_{n-1}, \{\langle \Delta \cdot r_\ell \rangle\}_{\ell=1}^k)$. All parties also hold $\{[\Delta|_\ell]_t\}_{\ell=1}^k$.

1. For all $\ell \in \{1, \dots, k\}$, all parties locally transform $[\boldsymbol{r}]_{n-1}$ to an additive sharing $\langle r_\ell \rangle$ and refresh the obtained additive sharing.
2. For all $\ell \in \{1, \dots, k\}$, all parties locally compute $\langle c_\ell + r_\ell \rangle = \langle c_\ell \rangle + \langle r_\ell \rangle$ and send their shares to P_1.
3. For all $\ell \in \{1, \dots, k\}$, P_1 locally reconstructs $c_\ell + r_\ell$. Then P_1 reshares the secrets by $[\boldsymbol{c} + \boldsymbol{r}]_{2k-2}$.
4. All parties locally compute $[\boldsymbol{c}]_{n-1} = [\boldsymbol{c} + \boldsymbol{r}]_{2k-2} - [\boldsymbol{r}]_{n-1}$.
5. For all $\ell \in \{1, \dots, k\}$, all parties locally compute $[\Delta \cdot (c_\ell + r_\ell)|_\ell]_{n-1} = [\Delta|_\ell]_t \cdot [\boldsymbol{c} + \boldsymbol{r}]_{2k-2}$.
6. All parties locally transform $[\Delta \cdot (c_\ell + r_\ell)|_\ell]_{n-1}$ to an additive sharing $\langle \Delta \cdot (c_\ell + r_\ell) \rangle$ and compute $\langle \Delta \cdot c_\ell \rangle = \langle \Delta \cdot (c_\ell + r_\ell) \rangle - \langle \Delta \cdot r_\ell \rangle$. Finally, all parties locally refresh $\langle \Delta \cdot c_\ell \rangle$.

Communication complexity of π_{Auth}. In the protocol the following interaction happens:

- (Step 2) $k \cdot (n-1)$ shares from the parties to P_1,
- (Step 3) $n - (k-1)$ shares from P_1 to the parties

This results in $k \cdot (n-2) + n + 1$.

D.3 Preparing Random Sharings with Authentications

In this section, we show how to prepare random sharings in the form of $([r \cdot \mathbf{1}]_{n-k}, \langle \Delta \cdot r \rangle)$. We make use of the functionality $\mathcal{F}_{\mathsf{nVOLE}}$ from [RS22].

The idea is to first use $\mathcal{F}_{\mathsf{nVOLE}}$ to prepare random sharings in the form $([\boldsymbol{r}]_{n-1}, \{\langle \Delta \cdot r_i \rangle\}_{i=1}^k)$. Then for all $i \in \{1, \dots, k\}$, we transform $([\boldsymbol{r}]_{n-1}, \langle \Delta \cdot r_i \rangle)$ to $([r_i \cdot \mathbf{1}]_{n-k}, \langle \Delta \cdot r_i \rangle)$ by using a random packed Shamir sharing $[\rho \cdot \mathbf{1}]_{n-k}$, which can be prepared by using π_{RandSh}. We describe the protocol π_{RandAuth} as follows. The amortized communication complexity per sharing is $2n + n/k$ elements.

Procedure 10: π_{RandAuth}

1. Each party P_i calls $\mathcal{F}_{\mathsf{nVOLE}}$ with input `Extend` and receives invokes the seed s^i. Use the outputs to define degree-$(n-1)$ packed Shamir sharings $\{[\boldsymbol{r}_\ell]_{n-1}\}_{\ell=1}^m$, where m is the output length of the expansion function defined in $\mathcal{F}_{\mathsf{nVOLE}}$, such that the i-th shares of $\{[\boldsymbol{r}_\ell]_{n-1}\}_{\ell=1}^m$ are $\mathtt{Expand}(s^i)$. All parties locally compute and refresh $\{(\langle \Delta \cdot r_{\ell,1} \rangle, \dots, \langle \Delta \cdot r_{\ell,k} \rangle)\}_{\ell=1}^m$.
2. All parties invoke π_{RandSh} to prepare $k \cdot m$ random sharings in the form of $[\rho_i \cdot \mathbf{1}]_{n-k}$. For all $\ell \in \{1, \dots, m\}$ consume k fresh random sharings $\{[\rho_i \cdot \mathbf{1}]_{n-k}\}_{i=1}^k$ and run the following steps.
   (a) All parties locally compute $[\boldsymbol{r}_\ell + \boldsymbol{\rho}]_{n-1} = [\boldsymbol{r}_\ell]_{n-1} + \sum_{i=1}^k \boldsymbol{e}_i * [\rho_i \cdot \mathbf{1}]_{n-k}$, where $\boldsymbol{\rho} = (\rho_1, \dots, \rho_k)$. Then all parties send their shares of $[\boldsymbol{r}_\ell + \boldsymbol{\rho}]_{n-1}$ to P_1.
   (b) P_1 reconstructs the secrets $\boldsymbol{r}_\ell + \boldsymbol{\rho}$ and sends the secrets to all parties.
   (c) For all $i \in \{1, \dots, k\}$, all parties locally compute $[r_{\ell,i} \cdot \mathbf{1}]_{n-k} = (r_{\ell,i} + \rho_i) - [\rho_i \cdot \mathbf{1}]_{n-k}$. Recall that all parties hold $\langle \Delta \cdot r_{\ell,i} \rangle$. Therefore all parties hold $([r_{\ell,i} \cdot \mathbf{1}]_{n-k}, \langle \Delta \cdot r_{\ell,i} \rangle)$.
3. Finally all parties check whether P_1 sends the same values to all parties:
   (a) Suppose P_1 distributes $\boldsymbol{\tau}_1, \dots, \boldsymbol{\tau}_m$ to all parties.
   (b) All parties call $\mathcal{F}_{\mathsf{Coin}}$ to obtain random values $\chi_1, \dots, \chi_m \in \mathbb{F}$.
   (c) All parties locally compute $\boldsymbol{\tau} = \sum_{\ell=1}^m \chi_\ell \cdot \boldsymbol{\tau}_\ell$.
   (d) All parties exchange their results $\boldsymbol{\tau}$ and check whether all results are identical. If not, all parties abort.

Communication complexity of π_{RandAuth}. In Procedure π_{RandAuth} the following is communicated per authenticated random sharing produced. We ignore the cost of Step 3, as it is independent of the amount of random sharings desired.

– (Step 2) One call to π_{RandSh} to generate k sharings of the form $[\rho \cdot \mathbf{1}]_{n-k}$. This costs $k \cdot n$.
– (Step 2a) $n-1$ shares reconstructed to P_1
– (Step 2b) $k \cdot (n-1)$ reconstructed secrets sent by P_1 to the parties.

In total, the communication equals $k(2n-1)+n-1$.

D.4 Verification of Packed Beaver Triples

In [RS22], the Beaver triples obtained from $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ may be incorrect. This is because a corrupted party P_i may use different shares $a^i, \tilde{a}^i$ when invoking $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ with different honest parties $P_j, P_{j'}$. And P_i may also locally change his share of $\langle c \rangle$ before computing the MAC of c.

To ensure that the online protocol uses correct Beaver triples, Rachuri and Scholl [RS22] follow the idea in [CGH⁺18] and evaluate a randomized version of the target circuit in addition to the standard version. Concretely, for each wire value w, all parties will compute an authenticated additive sharing of w and a randomized version $r \cdot w$, where r is a hidden secret generated at the beginning of the computation. At the end of the protocol, all parties together verify whether the randomized wire values correctly correspond to the real wire values before reconstructing the final output. Note that the evaluation of a randomized version of the target circuit requires to use fresh Beaver triples. Therefore, equivalently, the protocol in [RS22] uses two Beaver triples per multiplication gate.

Our idea is to verify the correctness of (packed) Beaver triples in the preprocessing phase following from the technique of sacrificing [DKL⁺13]: We will prepare two possibly incorrect packed Beaver triples and use one packed Beaver triple to verify the other one. We describe the procedure $\pi_{\mathsf{Sacrifice}}$ as follows.

Procedure 11: $\pi_{\mathsf{Sacrifice}}$

1. Suppose all parties hold m packed Beaver triples with authentications, denoted by

$$\{([\boldsymbol{a}_\ell]_{n-k}, [\Delta \cdot \boldsymbol{a}_\ell]_{n-k}), ([\boldsymbol{b}_\ell]_{n-k}, [\Delta \cdot \boldsymbol{b}_\ell]_{n-k}), \\ ([\boldsymbol{c}_\ell]_{n-1}, \{\langle \Delta \cdot c_{\ell,i} \rangle\}_{i=1}^k)\}_{\ell=1}^m.$$

2. All parties run Step 1, 2, 5 of **Generation** in π_{Triple} to prepare m packed Beaver triples with authentications in the following form:

$$\{([\tilde{\boldsymbol{a}}_\ell]_{n-1}, \{\langle \Delta \cdot \tilde{a}_{\ell,i} \rangle\}_{i=1}^k), ([\tilde{\boldsymbol{b}}_\ell]_{n-1}, \{\langle \Delta \cdot \tilde{b}_{\ell,i} \rangle\}_{i=1}^k), \\ ([\tilde{\boldsymbol{c}}_\ell]_{n-1}, \{\langle \Delta \cdot \tilde{c}_{\ell,i} \rangle\}_{i=1}^k)\}_{\ell=1}^m.$$

3. All parties invoke $\mathcal{F}_{\mathsf{Coin}}$ and generate a random element $\rho \in \mathbb{F}$.
4. For all $\ell \in \{1, \ldots, m\}$, all parties locally compute $[\tilde{\boldsymbol{a}}_\ell - \rho \cdot \boldsymbol{a}_\ell]_{n-1} = [\tilde{\boldsymbol{a}}_\ell]_{n-1} - \rho \cdot [\boldsymbol{a}_\ell]_{n-k}$ and $[\tilde{\boldsymbol{b}}_\ell - \boldsymbol{b}_\ell]_{n-1} = [\tilde{\boldsymbol{b}}_\ell]_{n-1} - [\boldsymbol{b}_\ell]_{n-k}$ and send their shares to P_1.
5. For all $j \in \{1, \ldots, m\}$, P_1 locally reconstructs the secrets $\tilde{\boldsymbol{a}}_\ell - \rho \cdot \boldsymbol{a}_\ell$ and $\tilde{\boldsymbol{b}}_\ell - \boldsymbol{b}_\ell$ and reshares the secrets by $[\tilde{\boldsymbol{a}}_\ell - \rho \cdot \boldsymbol{a}_\ell]_{k-1}, [\tilde{\boldsymbol{b}}_\ell - \boldsymbol{b}_\ell]_{k-1}$.
6. For all $j \in \{1, \ldots, m\}$, all parties locally compute

$$[\boldsymbol{\theta}_\ell]_{n-1} = [\tilde{\boldsymbol{a}}_\ell - \rho \cdot \boldsymbol{a}_\ell]_{k-1} * [\tilde{\boldsymbol{b}}_\ell - \boldsymbol{b}_\ell]_{k-1} + [\tilde{\boldsymbol{a}}_\ell - \rho \cdot \boldsymbol{a}_\ell]_{k-1} * [\boldsymbol{b}_\ell]_{n-k} \\ + \rho \cdot [\tilde{\boldsymbol{b}}_\ell - \boldsymbol{b}_\ell]_{k-1} * [\boldsymbol{a}_\ell]_{n-k} + \rho \cdot [\boldsymbol{c}_\ell]_{n-1} - [\tilde{\boldsymbol{c}}_\ell]_{n-1}.$$

7. For all $i \in \{1, \ldots, k\}$ and for all $\ell \in \{1, \ldots, m\}$,
   (a) Recall that all parties hold $\{[\Delta|_i]_t\}_{i=1}^k$. All parties locally compute $[\Delta \cdot (\tilde{a}_{\ell,i} - \rho \cdot a_{\ell,i}) \cdot (\tilde{b}_{\ell,i} - b_{\ell,i})|_i]_{n-1} = [\Delta|_i]_t \cdot [\tilde{\boldsymbol{a}}_\ell - \rho \cdot \boldsymbol{a}_\ell]_{k-1} \cdot [\tilde{\boldsymbol{b}}_\ell - \boldsymbol{b}_\ell]_{k-1}$. Then, all parties locally transform $[\Delta \cdot (\tilde{a}_{\ell,i} - \rho \cdot a_{\ell,i}) \cdot (\tilde{b}_{\ell,i} - b_{\ell,i})|_i]_{n-1}$ to an additive sharing $\langle \Delta \cdot (\tilde{a}_{\ell,i} - \rho \cdot a_{\ell,i}) \cdot (\tilde{b}_{\ell,i} - b_{\ell,i}) \rangle$.
   (b) Recall that all parties hold $[\Delta \cdot \boldsymbol{a}_\ell]_{n-k}, [\Delta \cdot \boldsymbol{b}_\ell]_{n-k}$. All parties locally compute $[\Delta \cdot (\tilde{\boldsymbol{a}}_\ell - \rho \cdot \boldsymbol{a}_\ell) * \boldsymbol{b}_\ell]_{n-1} = [\tilde{\boldsymbol{a}}_\ell - \rho \cdot \boldsymbol{a}_\ell]_{k-1} * [\Delta \cdot \boldsymbol{b}_\ell]_{n-k}$ and $\rho \cdot [\Delta \cdot (\tilde{\boldsymbol{b}}_\ell - \boldsymbol{b}_\ell) * \boldsymbol{a}_\ell]_{n-1} = \rho \cdot [\tilde{\boldsymbol{b}}_\ell - \boldsymbol{b}_\ell]_{k-1} * [\Delta \cdot \boldsymbol{a}_\ell]_{n-k}$. Then, all parties locally transform these two packed Shamir sharings to additive sharings $\langle \Delta \cdot (\tilde{a}_{\ell,i} - \rho \cdot a_{\ell,i}) \cdot b_{\ell,i} \rangle$ and $\langle \Delta \cdot \rho \cdot (\tilde{b}_{\ell,i} - b_{\ell,i}) \cdot a_{\ell,i} \rangle$.

(c) Recall that all parties hold additive sharings $\langle \Delta \cdot c_{\ell,i}\rangle$ and $\langle \Delta \cdot \tilde{c}_{\ell,i}\rangle$.
(d) All parties finally compute an additive sharing $\langle \Delta \cdot \theta_{\ell,i}\rangle$ by

$$\langle \Delta \cdot (\tilde{a}_{\ell,i} - \rho \cdot a_{\ell,i}) \cdot (\tilde{b}_{\ell,i} - b_{\ell,i})\rangle + \langle \Delta \cdot (\tilde{a}_{\ell,i} - \rho \cdot a_{\ell,i}) \cdot b_{\ell,i}\rangle$$
$$+ \langle \Delta \cdot \rho \cdot (\tilde{b}_{\ell,i} - b_{\ell,i}) \cdot a_{\ell,i}\rangle + \rho \cdot \langle \Delta \cdot c_{\ell,i}\rangle - \langle \Delta \cdot \tilde{c}_{\ell,i}\rangle.$$

Note that when $\boldsymbol{c}_\ell = \boldsymbol{a}_\ell * \boldsymbol{b}_\ell$ and $\tilde{\boldsymbol{c}}_\ell = \tilde{\boldsymbol{a}}_\ell * \tilde{\boldsymbol{b}}_\ell$, we have

$$\begin{aligned}\boldsymbol{\theta}_\ell &= (\tilde{\boldsymbol{a}}_\ell - \rho \cdot \boldsymbol{a}_\ell) * (\tilde{\boldsymbol{b}}_\ell - \boldsymbol{b}_\ell) + (\tilde{\boldsymbol{a}}_\ell - \rho \cdot \boldsymbol{a}_\ell) * \boldsymbol{b}_\ell \\ &\quad + \rho \cdot (\tilde{\boldsymbol{b}}_\ell - \boldsymbol{b}_\ell) * \boldsymbol{a}_\ell + \rho \cdot \boldsymbol{c}_\ell - \tilde{\boldsymbol{c}}_\ell \\ &= \tilde{\boldsymbol{a}}_\ell * \tilde{\boldsymbol{b}}_\ell - \tilde{\boldsymbol{c}}_\ell \\ &= \boldsymbol{0}.\end{aligned}$$

Thus, it is sufficient to verify that $\boldsymbol{\theta}_\ell$ is an all-0 vector. A subtle issue with the sacrifice approach given above is that we also must verify that P_1 distributes correct degree-$(k-1)$ packed Shamir sharings. For this, we use Procedure $\pi_{\mathsf{VerifyDeg}}$ described below.

Procedure 12: $\pi_{\mathsf{VerifyDeg}}$

Input: All parties hold m degree-$(k-1)$ packed Shamir sharings distributed by P_1, denoted by

$$[\boldsymbol{x}_1]_{k-1}, \ldots, [\boldsymbol{x}_m]_{k-1}.$$

All parties hold an additive sharing $\langle \Delta \rangle$. For each $\ell \in \{1,\ldots,m\}$, all parties also hold $\{\langle \Delta \cdot x_{\ell,i}\rangle\}_{i=1}^k$

1. All parties call $\mathcal{F}_{\mathsf{Coin}}$ to obtain random values $\chi_1,\ldots,\chi_m \in \mathbb{F}$.
2. All parties locally compute $[\boldsymbol{x}]_{k-1} = \sum_{\ell=1}^m \chi_\ell \cdot [\boldsymbol{x}_\ell]_{k-1}$ and compute $\langle \Delta \cdot x_i\rangle = \sum_{\ell=1}^m \chi_\ell \cdot \langle x_{\ell,i}\rangle_{k-1}$ for all $i \in \{1,\ldots,k\}$.
3. All parties exchange their shares of $[\boldsymbol{x}]_{k-1}$ and check that the shares of $[\boldsymbol{x}]_{k-1}$ lie on a degree-$(k-1)$ polynomial. If false, all parties abort.
4. All parties reconstruct the secrets $\boldsymbol{x}$. For all $i \in \{1,\ldots,k\}$, all parties locally compute $\langle \delta_i \rangle = \langle \Delta \cdot x_i\rangle - x_i \cdot \langle \Delta \rangle$ and refresh the additive sharing $\langle \delta_i\rangle$.
5. For all $i \in \{1,\ldots,k\}$, all parties call $\mathcal{F}_{\mathsf{Commit}}$ to commit their shares of $\langle \delta_i\rangle$ towards each other.
6. For all $i \in \{1,\ldots,k\}$, all parties open the commitments of the shares of $\langle \delta_i \rangle$ and check whether it is an additive sharing of 0. If not, all parties abort.

Finally, we verify that $\boldsymbol{\theta}_\ell$ is an all-0 vector by making use of the procedure $\pi_{\mathsf{CheckZero}}$ below.

Procedure 13: $\pi_{\mathsf{CheckZero}}$

Input: All parties hold m degree-$(n-1)$ packed Shamir sharings, denoted by

$$[\boldsymbol{\theta}_1]_{n-1}, \ldots, [\boldsymbol{\theta}_m]_{n-1}.$$

All parties hold an additive sharing $\langle \Delta \rangle$. For each $\ell \in \{1,\ldots,m\}$, all parties also hold $\{\langle \Delta \cdot \theta_{\ell,i}\rangle\}_{i=1}^k$

1. All parties call $\mathcal{F}_{\mathsf{Coin}}$ to obtain random values $\chi_1,\ldots,\chi_m \in \mathbb{F}$.
2. All parties invoke π_{RandSh} to prepare a random degree-$(n-1)$ packed Shamir sharing of $\boldsymbol{0}$, denoted by $[\boldsymbol{0}]_{n-1}$.
3. All parties locally compute $[\boldsymbol{\theta}]_{n-1} = [\boldsymbol{0}]_{n-1} + \sum_{\ell=1}^m \chi_\ell \cdot [\boldsymbol{\theta}_\ell]_{n-1}$ and compute $\langle \Delta \cdot \theta_i\rangle = \sum_{\ell=1}^m \chi_\ell \cdot \langle \theta_{\ell,i}\rangle_{k-1}$ for all $i \in \{1,\ldots,k\}$. Then all parties locally refresh the additive sharings $\{\langle \Delta \cdot \theta_i\rangle\}_{i=1}^k$.
4. All parties call $\mathcal{F}_{\mathsf{Commit}}$ to commit their shares of $[\boldsymbol{\theta}]_{n-1}$ towards each other.
5. All parties open the commitments of the shares of $[\boldsymbol{\theta}]_{n-1}$ and check whether it is a degree-$(n-1)$ packed Shamir sharing of $\boldsymbol{0}$. If not, all parties abort.
6. For all $i \in \{1,\ldots,k\}$, all parties call $\mathcal{F}_{\mathsf{Commit}}$ to commit their shares of $\langle \Delta \cdot \theta_i\rangle$ towards each other.
7. For all $i \in \{1,\ldots,k\}$, all parties open the commitments of the shares of $\langle \Delta \cdot \theta_i \rangle$ and check whether it is an additive sharing of 0. If not, all parties abort.

The security of the above verification follows from a similar idea in [RS22] which we will discuss in the proof.

Communication complexity of $\pi_{\mathsf{Sacrifice}}$. We first notice that the communication of $\pi_{\mathsf{VerifyDeg}}$ and $\pi_{\mathsf{CheckZero}}$ are independent of the amount of multiplications, so we ignore their costs. Regarding $\pi_{\mathsf{Sacrifice}}$, the communication is discussed below. We ignore calls to $\mathcal{F}_{\mathsf{Coin}}$.

– (Step 2) Parties run steps 1, 2 and 5 in π_{Triple}. Among these, only the last step involves interaction that amounts to $k(n-2)+n+1$.
– (Step 4) $2(n-1)$ shares sent by the parties to P_1.
– (Step 6) $2(n-1)$ shares sent by P_1 to the parties.

The total is $k(n-2)+5n-3$.

D.5 Full Description of the Circuit-Independent Preprocessing Protocol

We are ready to present the protocol $\Pi_{\mathsf{PrepIndMal}}$ that realizes $\mathcal{F}_{\mathsf{PrepIndMal}}$ below.

Protocol 3: $\Pi_{\mathsf{PrepIndMal}}$

Initialization: For every ordered pair of parties (P_i, P_j), P_i sends a random PRG seed $s_{i,j}$ to P_j. Whenever P_i needs to sample a random value and sends it to P_j, P_i and P_j locally evaluate the PRG with random seed $s_{i,j}$ to obtain the common random value.

1. **Settling Authentication Keys**: All parties invoke **Initialization** in π_{Triple} to prepare random sharings $\{[\Delta|_i]_t\}_{i=1}^k$.
2. **Preparing Random Packed Sharings**: Let N_1 be the number of input gates and multiplication gates. All parties invoke π_{RandAuth} to prepare N_1 random sharings in the form of $([r\cdot\mathbf{1}]_{n-k}, \langle\Delta\cdot r\rangle)$.
3. **Preparing Packed Beaver Triples with Authentications**: Let N_2 denote the number of groups of multiplication/input/output gates. Repeat the following, until $\geq N_2$ triples are generated.
   (a) All parties invoke **Generation** in π_{Triple} to prepare m packed Beaver triples with authentications, where m is the output length in π_{Triple}.
   (b) All parties invoke $\pi_{\mathsf{Sacrifice}}$, $\pi_{\mathsf{VerifyDeg}}$, and $\pi_{\mathsf{CheckZero}}$ to verify the correctness of the triples prepared in the last step.
4. **Preparing Random Masked Sharings for Multiplication Gates**: Let N_3 denote the number of groups of multiplication gates. All parties invoke π_{RandSh} to prepare $3N_3$ random sharings in the form of $[\mathbf{0}]_{n-1}$.
5. **Preparing Random Masked Sharings for Input and Output Gates**: Let N_4 be the number of groups of input gates and output gates. Then we have $N_2 = N_3 + N_4$. All parties invoke π_{RandSh} to prepare N_4 random sharings in the form of $[\mathbf{0}]_{n-1}$.

Communication complexity of $\Pi_{\mathsf{PrepIndMal}}$. The communication of $\Pi_{\mathsf{PrepIndMal}}$ per multiplication group is divided in the following:

– (Step 2) One call to π_{RandAuth}, which costs $k(2n-1)+n-1$
– (Step 3) One call to π_{Triple}, which costs $k\cdot(3n-8)+10n+3$, and one call to $\pi_{\mathsf{Sacrifice}}$, which costs $k(n-2)+5n-3$.
– (Step 4) One call to π_{RandSh} to generate 3 random sharings $[\mathbf{0}]_{n-1}$. This costs $3(n/2)$.

Summing up, we obtain $k(6n-11)+17.5n-1$, for every group of k multiplication gates. If we divide by k, we obtain $6n-11+(17.5n-1)/k$ per multiplication gate, and if we reall that $k=(\epsilon/2)n+1$, we obtain

$$6n-11+\frac{17.5n-1}{\frac{\epsilon}{2}n+1}.$$

We can upper bound this by $6n+\frac{35}{\epsilon}$.

D.6 Proof of Lemma 2

Lemma 2. *Protocol $\Pi_{\mathsf{PrepIndMal}}$ securely computes $\mathcal{F}_{\mathsf{PrepIndMal}}$ in the $\{\mathcal{F}_{\mathsf{OLE}}^{prog}, \mathcal{F}_{\mathsf{nVOLE}}, \mathcal{F}_{\mathsf{Commit}}, \mathcal{F}_{\mathsf{Coin}}\}$-hybrid model against a malicious adversary who controls t out of n parties.*

Proof. We will construct a simulator $\mathcal{S}$ to simulate the behaviors of honest parties. Let $Corr$ denote the set of corrupted parties and $\mathcal{H}$ denote the set of honest parties. The simulator $\mathcal{S}$ works as follows.

For every ordered pair (P_i, P_j) where P_i is honest and P_j is corrupted, $\mathcal{S}$ samples a random PRG seed and sends it to P_j. For every ordered pair (P_i, P_j) where P_i is corrupted and P_i is honest, $\mathcal{S}$ receives a PRG seed from P_i.

Simulating π_{RandSh}. We first describe a general strategy of simulating π_{RandSh}. In our construction, we will only use π_{RandSh} to generate random sharings which satisfy that the shares of corrupted parties are independent of the secrets. Recall that we only focus on the case where $t \geq n/2$. It means that the number of honest parties is upper bounded by the number of corrupted parties.

At a high level, whenever an honest party P_i needs to generate and distribute a random Σ-sharing, for every corrupted party P_j,

- If the protocol requires P_i and P_j to obtain their share by evaluating the PRG with the common seed, $\mathcal{S}$ honestly evaluates the PRG and obtains the share of P_j.
- Otherwise, $\mathcal{S}$ samples a random element as the share of P_j and sends it to P_j.

Whenever a corrupted party P_i distributes a random Σ-sharing, $\mathcal{S}$ receives the shares of honest parties (by either evaluating the PRG with the common seeds or receiving from P_i). Then $\mathcal{S}$ samples a random Σ-sharing based on the shares of honest parties and views it as the Σ-sharing distributed by P_i. Note that, corrupted parties can locally change their shares to any arbitrary values. This way, we have defined the correct shares that corrupted parties should hold.

In details,

1. In Step 1, $\mathcal{S}$ follows the protocol to agree on a Vandermonde matrix $\boldsymbol{M}^{\mathrm{T}}$.
2. In Step 2, for each honest party P_i, $\mathcal{S}$ distributes the shares of $\boldsymbol{S}^{(i)}$ to corrupted parties as described above. For each corrupted party P_i, $\mathcal{S}$ receives the shares of $\boldsymbol{S}^{(i)}$ of honest parties. $\mathcal{S}$ randomly samples a Σ-sharing based on the shares of honest parties and view it as the sharing generated by P_i.
3. In Step 3, $\mathcal{S}$ computes the shares of corrupted parties for each Σ-sharing $\boldsymbol{R}^{(i)}$.

Simulating the Main Protocol. No we describe the simulation of $\Pi_{\mathsf{PrepIndMal}}$.

1. In Step 1, $\mathcal{S}$ simulates **Initialization** in π_{Triple} as follows:
   (a) $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{nVOLE}}$ and receives Δ^i for each corrupted party P_i.
   (b) $\mathcal{S}$ simulates π_{RandSh} when preparing $\{[r|_i]_t\}_{i=1}^t$ as described above and learns the shares of corrupted parties. Then $\mathcal{S}$ simulates π_{MACKey} as follows.
   (c) $\mathcal{S}$ follows the protocol when transforming $[r|_1]_t$ to $\langle r\rangle$ and computes the shares of corrupted parties. When refreshing $\langle r\rangle$, recall that it is done by first invoking π_{RandSh} to prepare a random additive sharing $\langle 0\rangle$ and then adding it with $\langle r\rangle$. $\mathcal{S}$ computes the corrupted parties' shares of $\langle r\rangle$ after refreshing.
   (d) $\mathcal{S}$ computes the shares of $\langle \Delta + r\rangle$ of corrupted parties and samples random values as the shares of honest parties. $\mathcal{S}$ reconstructs the secrets $\Delta + r$. Then $\mathcal{S}$ honestly sends the shares of $\langle \Delta + r\rangle$ of honest parties to P_1.
   (e) Depending on whether P_1 is honest or not, there are two cases:
      - If P_1 is honest, $\mathcal{S}$ receives the shares of corrupted parties and reconstructs the secret $\overline{\Delta + r}$ (which can be different from $\Delta + r$ since corrupted parties may send wrong shares to P_1). $\mathcal{S}$ honestly follows the protocol and distributes $[(\overline{\Delta + r}) \cdot \mathbf{1}]_{2k-2}$.
        For all $i \in \{1, \ldots, k\}$, $\mathcal{S}$ computes the shares of $[(\overline{\Delta + r}) - r|_i]_t$ of corrupted parties. Note that for a degree-t Shamir sharing, when $t \geq n/2$, corrupted parties can locally change their shares to change the secret from $(\overline{\Delta + r}) - r$ to $(\Delta + r) - r = \Delta$. We may equivalently think that the shares of corrupted parties are changed so that the secret is Δ (and they can then change their shares to any arbitrary values). Thus, $\mathcal{S}$ adjust the shares of corrupted parties as follows:

$\mathcal{S}$ generates a degree-t Shamir sharing $[\Delta + r - (\overline{\Delta + r})|_i]_t$ such that the shares of honest parties are 0s. Then $\mathcal{S}$ computes the shares of $[\Delta|_i]_t = [\Delta + r - (\overline{\Delta + r})|_i]_t + [(\overline{\Delta + r}) - r|_i]_t$ of corrupted parties.

  - If P_1 is corrupted, $\mathcal{S}$ receives the shares of $[(\Delta + r) \cdot \mathbf{1}]_{2k-2}$ of honest parties and reconstructs the whole sharing and the secrets $\overline{(\Delta + r) \cdot \mathbf{1}}$. Then $\mathcal{S}$ computes $\boldsymbol{\delta} = \overline{(\Delta + r) \cdot \mathbf{1}} - \Delta \cdot \mathbf{1}$.
    For all $i \in \{1, \ldots, k\}$, $\mathcal{S}$ computes the shares of $[(\Delta + r) \cdot \mathbf{1}]_{2k-2} - [r|_i]_t$ of corrupted parties. Note that the secret is equal to $\Delta + \delta_i$. $\mathcal{S}$ adjust the shares of corrupted parties as follows:
    $\mathcal{S}$ generates a degree-t Shamir sharing $[\delta_i|_i]_t$ such that the shares of honest parties are 0s. Then $\mathcal{S}$ computes the shares of $[\Delta|_i]_t = [(\Delta + r) \cdot \mathbf{1}]_{2k-2} - [r|_i]_t - [\delta_i|_i]_t$ of corrupted parties.

2. In Step 2, $\mathcal{S}$ simulates π_{RandAuth} as follows.
   (a) $\mathcal{S}$ first emulates $\mathcal{F}_{\mathsf{nVOLE}}$ and receives the shares of corrupted parties. For any global key query made by a corrupted party, $\mathcal{S}$ randomly samples Δ^i for each honest party P_i, sends back $(\mathtt{abort}, \Delta)$, and aborts.
   (b) $\mathcal{S}$ simulates π_{RandSh} when preparing random sharings in the form of $[\rho \cdot \mathbf{1}]_{n-k}$ as described above and learns the shares of corrupted parties. For all $\ell \in \{1, \ldots, m\}$, $\mathcal{S}$ computes the shares of $[\boldsymbol{r}_\ell + \boldsymbol{\rho}]_{n-1}$ of corrupted parties. $\mathcal{S}$ samples random values as the shares of $[\boldsymbol{r}_\ell + \boldsymbol{\rho}]_{n-1}$ of honest parties and reconstructs the secrets $\boldsymbol{r}_\ell + \boldsymbol{\rho}$. Depending on whether P_1 is honest, there are two cases.
     - If P_1 is honest, $\mathcal{S}$ receives the shares of $[\boldsymbol{r}_\ell + \boldsymbol{\rho}]_{n-1}$ of corrupted parties and reconstructs the secrets $\overline{\boldsymbol{r}_\ell + \boldsymbol{\rho}}$. Then P_1 honestly sends back $\overline{\boldsymbol{r}_\ell + \boldsymbol{\rho}}$ to all parties.
     - If P_1 is corrupted, $\mathcal{S}$ sends the shares of $[\boldsymbol{r}_\ell + \boldsymbol{\rho}]_{n-1}$ of honest parties to P_1 and receives $\overline{\boldsymbol{r}_\ell + \boldsymbol{\rho}}$ from P_1.
       If P_1 sends different values to honest parties, $\mathcal{S}$ views the values sent to the first honest party as the actual values distributed by P_1. Later on, $\mathcal{S}$ will abort in this case.

   For all $i \in \{1, \ldots, k\}$, $\mathcal{S}$ computes the shares of $[(\overline{r_{\ell,i} + \rho_i} - \rho_i) \cdot \mathbf{1}]_{n-k} = \overline{r_{\ell,i} + \rho_i} - [\rho_i \cdot \mathbf{1}]_{n-k}$ of corrupted parties. Note that for a degree-$(n-k)$ packed Shamir sharing, when $t \geq n/2$, corrupted parties can locally change their shares to change the secrets from $(\overline{r_{\ell,i} + \rho_i} - \rho_i) \cdot \mathbf{1}$ to $r_{\ell,i} \cdot \mathbf{1}$. We may equivalently think that the shares of corrupted parties are changed so that the secret is $r_{\ell,i} \cdot \mathbf{1}$ (and they can then change their shares to any arbitrary values). Thus, $\mathcal{S}$ adjust the shares of corrupted parties as follows:
   $\mathcal{S}$ generates a degree-$(n-k)$ packed Shamir sharing $[r_{\ell,i} + \rho_i - (\overline{r_{\ell,i} + \rho_i})]_{n-k}$ such that the shares of honest parties are 0s. Then $\mathcal{S}$ computes the shares of $[r_{\ell,i} \cdot \mathbf{1}]_{n-k} = [(\overline{r_{\ell,i} + \rho_i} - \rho_i) \cdot \mathbf{1}]_{n-k} + [r_{\ell,i} + \rho_i - (\overline{r_{\ell,i} + \rho_i})]_{n-k}$ of corrupted parties.
   (c) $\mathcal{S}$ honestly emulates $\mathcal{F}_{\mathsf{Coin}}$ and honestly follows the protocol to check whether P_1 distributes the same values to all parties. If P_1 *did not* distributes the same values to all honest parties but the check passes, $\mathcal{S}$ aborts.
3. In Step 3(a), $\mathcal{S}$ simulates π_{Triple} as follows.
   (a) $\mathcal{S}$ first emulates $\mathcal{F}_{\mathsf{nVOLE}}$ and receives the shares of corrupted parties. For any global key query made by a corrupted party, $\mathcal{S}$ randomly samples Δ^i for each honest party P_i, sends back $(\mathtt{abort}, \Delta)$, and aborts.
   Let s_a^i, s_b^i denote the seeds of corrupted parties, and let $\boldsymbol{a}^i = \mathtt{Expand}(s_a^i), \boldsymbol{b}^i = \mathtt{Expand}(s_b^i)$ denote the shares of corrupted parties.
   (b) For every ordered pair of parties (P_i, P_j), $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ as follows: Initially, for all $i, j \in \{1, \ldots, n\}$, $\mathcal{S}$ sets $\boldsymbol{\delta}_a^{i,j} = \boldsymbol{\delta}_b^{i,j} = \mathbf{0}$.
     - If P_i and P_j are both corrupted, $\mathcal{S}$ randomly samples $\boldsymbol{u}^{i,j}$ and $\boldsymbol{v}^{j,i}$ such that $\boldsymbol{u}^{i,j} + \boldsymbol{v}^{j,i} = \mathtt{Expand}(s_a^i) * \mathtt{Expand}(s_b^j)$.
     - If P_i is corrupted and P_j is honest, $\mathcal{S}$ receives $\bar{s}_a^i$ and $\boldsymbol{u}^{i,j}$. Then $\mathcal{S}$ sets $\boldsymbol{\delta}_a^{i,j} = \mathtt{Expand}(\bar{s}_a^i) - \boldsymbol{a}^i$.
     - If P_i is honest and P_j is corrupted, $\mathcal{S}$ receives $\bar{s}_b^j$ and $\boldsymbol{v}^{j,i}$. Then $\mathcal{S}$ sets $\boldsymbol{\delta}_b^{i,j} = \mathtt{Expand}(\bar{s}_b^j) - \boldsymbol{b}^j$.

   Then for all $\ell \in \{1, \ldots, m\}$, (P_i, P_j) holds $u_\ell^{i,j}, v_\ell^{i,j}$ such that $u_\ell^{i,j} + v_\ell^{i,j} = (a_\ell^i + \delta_{a,\ell}^{i,j}) \cdot (b_\ell^j + \delta_{b,\ell}^{i,j})$.
   For all $i \in \{1, \ldots, k\}$, when transforming to $\langle c_{\ell,i} \rangle$, there exists $\eta_{a,\ell,i}^j, \eta_{b,\ell,i}^j$ such that $c_{\ell,i} = a_{\ell,i} \cdot b_{\ell,i} + \sum_{j=1}^{n} (\eta_{a,\ell,i}^j \cdot b_\ell^j + a_\ell^j \cdot \eta_{b,\ell,i}^j)$.

$\mathcal{S}$ computes $\{(\eta^j_{a,\ell,i}, \eta^j_{b,\ell,i})\}_{j=1}^n$ for all $\ell \in \{1, \dots, m\}$ and $i \in \{1, \dots, k\}$. $\mathcal{S}$ also computes the shares of $\{(\langle c_{\ell,1}\rangle, \dots, \langle c_{\ell,k}\rangle)\}_{\ell=1}^m$ of corrupted parties.

(c) $\mathcal{S}$ simulates π_{RandSh} when preparing random sharings in the form of $([\boldsymbol{r}]_{n-k}, [\boldsymbol{r}]_{n-1})$ as described above and learns the shares of corrupted parties. Then $\mathcal{S}$ simulates $\pi_{\mathsf{DegReduce}}$: $\mathcal{S}$ computes the shares of $[\boldsymbol{u}+\boldsymbol{r}]_{n-1}$ of corrupted parties and samples random values as the shares of $[\boldsymbol{u}+\boldsymbol{r}]_{n-1}$ of honest parties. Then $\mathcal{S}$ reconstructs the secrets $\boldsymbol{u}+\boldsymbol{r}$. Depending on whether P_1 is honest, there are two cases.
   - If P_1 is honest, $\mathcal{S}$ receives the shares of $[\boldsymbol{u}+\boldsymbol{r}]_{n-1}$ of corrupted parties and reconstructs the secrets $\overline{\boldsymbol{u}+\boldsymbol{r}}$. Then P_1 honestly distributes $[\overline{\boldsymbol{u}+\boldsymbol{r}}]_{2k-2}$ to all parties.
   - If P_1 is corrupted, $\mathcal{S}$ sends the shares of $[\boldsymbol{u}+\boldsymbol{r}]_{n-1}$ of honest parties to P_1 and receives the shares of $[\overline{\boldsymbol{u}+\boldsymbol{r}}]_{2k-2}$ of honest parties from P_1.
     $\mathcal{S}$ reconstructs the whole sharing $[\overline{\boldsymbol{u}+\boldsymbol{r}}]_{2k-2}$ and computes the secrets $\overline{\boldsymbol{u}+\boldsymbol{r}}$.

   $\mathcal{S}$ computes the shares of $[\overline{\boldsymbol{u}+\boldsymbol{r}}-\boldsymbol{r}]_{n-k} = [\overline{\boldsymbol{u}+\boldsymbol{r}}]_{2k-2} - [\boldsymbol{r}]_{n-k}$ of corrupted parties. Note that for a degree-$(n-k)$ packed Shamir sharing, when $t \geq n/2$, corrupted parties can locally change their shares to change the secrets from $\overline{\boldsymbol{u}+\boldsymbol{r}}-\boldsymbol{r}$ to $\boldsymbol{u}$. We may equivalently think that the shares of corrupted parties are changed so that the secret is $\boldsymbol{u}$ (and they can then change their shares to any arbitrary values). Thus, $\mathcal{S}$ adjust the shares of corrupted parties as follows:
   $\mathcal{S}$ generates a degree-$(n-k)$ packed Shamir sharing $[\boldsymbol{u}+\boldsymbol{r}-\overline{\boldsymbol{u}+\boldsymbol{r}}]_{n-k}$ such that the shares of honest parties are 0s. Then $\mathcal{S}$ computes the shares of $[\boldsymbol{u}]_{n-k} = [\overline{\boldsymbol{u}+\boldsymbol{r}}-\boldsymbol{r}]_{n-k} + [\boldsymbol{u}+\boldsymbol{r}-\overline{\boldsymbol{u}+\boldsymbol{r}}]_{n-k}$ of corrupted parties.

(d) $\mathcal{S}$ simulates π_{RandSh} when preparing random sharings in the form of $[\boldsymbol{r}]_{n-k}$ as described above and learns the shares of corrupted parties. Then $\mathcal{S}$ simulates π_{AddTran} as follows: For all $\ell \in \{1, \dots, k\}$, $\mathcal{S}$ computes the corrupted parties' shares of $\langle r_\ell \rangle$ after refreshing. Then $\mathcal{S}$ computes the shares of $\langle \Delta \cdot u_\ell + r_\ell \rangle$ of corrupted parties and samples random values as the shares of $\langle \Delta \cdot u_\ell + r_\ell \rangle$ of honest parties. Next $\mathcal{S}$ reconstructs the secret $\Delta \cdot u_\ell + r_\ell$. Depending on whether P_1 is honest, there are two cases.
   - If P_1 is honest, $\mathcal{S}$ receives the shares of $\langle \Delta \cdot u_\ell + r_\ell \rangle$ of corrupted parties and reconstructs the secret $\overline{\Delta \cdot u_\ell + r_\ell}$. Then P_1 honestly distributes $[\overline{\Delta \cdot \boldsymbol{u}+\boldsymbol{r}}]_{2k-2}$ to all parties.
   - If P_1 is corrupted, $\mathcal{S}$ sends the shares of $\langle \Delta \cdot u_\ell + r_\ell \rangle$ of honest parties to P_1 and receives the shares of $[\overline{\Delta \cdot \boldsymbol{u}+\boldsymbol{r}}]_{2k-2}$ of honest parties from P_1.
     $\mathcal{S}$ reconstructs the whole sharing $[\overline{\Delta \cdot \boldsymbol{u}+\boldsymbol{r}}]_{2k-2}$ and computes the secrets $\overline{\Delta \cdot \boldsymbol{u}+\boldsymbol{r}}$.

   $\mathcal{S}$ computes the shares of $[\overline{\Delta \cdot \boldsymbol{u}+\boldsymbol{r}}-\boldsymbol{r}]_{n-k} = [\overline{\Delta \cdot \boldsymbol{u}+\boldsymbol{r}}]_{2k-2} - [\boldsymbol{r}]_{n-k}$ of corrupted parties. Note that for a degree-$(n-k)$ packed Shamir sharing, when $t \geq n/2$, corrupted parties can locally change their shares to change the secrets from $\overline{\Delta \cdot \boldsymbol{u}+\boldsymbol{r}}-\boldsymbol{r}$ to $\Delta \cdot \boldsymbol{u}$. We may equivalently think that the shares of corrupted parties are changed so that the secret is $\Delta \cdot \boldsymbol{u}$ (and they can then change their shares to any arbitrary values). Thus, $\mathcal{S}$ adjust the shares of corrupted parties as follows:
   $\mathcal{S}$ generates a degree-$(n-k)$ packed Shamir sharing $[\Delta \cdot \boldsymbol{u}+\boldsymbol{r}-\overline{\Delta \cdot \boldsymbol{u}+\boldsymbol{r}}]_{n-k}$ such that the shares of honest parties are 0s. Then $\mathcal{S}$ computes the shares of $[\Delta \cdot \boldsymbol{u}]_{n-k} = [\overline{\Delta \cdot \boldsymbol{u}+\boldsymbol{r}}-\boldsymbol{r}]_{n-k} + [\Delta \cdot \boldsymbol{u}+\boldsymbol{r}-\overline{\Delta \cdot \boldsymbol{u}+\boldsymbol{r}}]_{n-k}$ of corrupted parties.

(e) $\mathcal{S}$ follows the simulation of Step 1 in π_{Triple} described above when preparing random sharings in the form of $([\boldsymbol{r}]_{n-1}, \{\langle \Delta \cdot r_i \rangle\}_{i=1}^k)$ and learns the shares of corrupted parties. Then $\mathcal{S}$ simulates π_{Auth} as follows: For all $\ell \in \{1, k\}$, $\mathcal{S}$ computes the corrupted parties' shares of $\langle r_\ell \rangle$ after refreshing. Then $\mathcal{S}$ computes the shares of $\langle c_\ell + r_\ell \rangle$ of corrupted parties and samples random values as the shares of $\langle c_\ell + r_\ell \rangle$ of honest parties. Next $\mathcal{S}$ reconstructs the secret $c_\ell + r_\ell$. Depending on whether P_1 is honest, there are two cases.
   - If P_1 is honest, $\mathcal{S}$ receives the shares of $\langle c_\ell + r_\ell \rangle$ of corrupted parties and reconstructs the secret $\overline{c_\ell + r_\ell}$. Then P_1 honestly distributes $[\overline{\boldsymbol{c}+\boldsymbol{r}}]_{2k-2}$ to all parties.
   - If P_1 is corrupted, $\mathcal{S}$ sends the shares of $\langle c_\ell + r_\ell \rangle$ of honest parties to P_1 and receives the shares of $[\overline{\boldsymbol{c}+\boldsymbol{r}}]_{2k-2}$ of honest parties from P_1.
     $\mathcal{S}$ reconstructs the whole sharing $[\overline{\boldsymbol{c}+\boldsymbol{r}}]_{2k-2}$ and computes the secrets $\overline{\boldsymbol{c}+\boldsymbol{r}}$.

   $\mathcal{S}$ computes the additive errors $\boldsymbol{\epsilon} = \overline{\boldsymbol{c}+\boldsymbol{r}} - (\boldsymbol{c}+\boldsymbol{r})$. Then $\mathcal{S}$ computes the shares of $[\boldsymbol{c}+\boldsymbol{\epsilon}]_{n-1} = [\overline{\boldsymbol{c}+\boldsymbol{r}}]_{2k-2} - [\boldsymbol{r}]_{n-1}$ of corrupted parties.
   $\mathcal{S}$ also computes the shares of $[\Delta \cdot (\overline{c_\ell + r_\ell})|_\ell]_{n-1} = [\Delta|_\ell]_t \cdot [\overline{\boldsymbol{c}+\boldsymbol{r}}]_{2k-2}$ of corrupted parties.
   $\mathcal{S}$ follows the protocol and computes the shares of $\langle \Delta \cdot (\overline{c_\ell + r_\ell}) \rangle$ of corrupted parties. Finally, $\mathcal{S}$ computes the shares of $\langle \Delta \cdot (c_\ell + \epsilon_\ell) \rangle = \langle \Delta \cdot (\overline{c_\ell + r_\ell}) \rangle - \langle \Delta \cdot r_\ell \rangle$ of corrupted parties.

4. In Step 3(b), $\mathcal{S}$ simulates $\pi_{\mathsf{Sacrifice}}$ as follows.
   (a) Recall that for every packed Beaver triple
   $$([\boldsymbol{a}_\ell]_{n-k}, [\Delta \cdot \boldsymbol{a}_\ell]_{n-k}), ([\boldsymbol{b}_\ell]_{n-k}, [\Delta \cdot \boldsymbol{b}_\ell]_{n-k}), ([\boldsymbol{c}_\ell]_{n-1}, \{\langle \Delta \cdot c_{\ell,i} \rangle\}_{i=1}^k),$$
   $\mathcal{S}$ learns the shares of corrupted parties. $\mathcal{S}$ also learns the errors $\{\eta_{a,\ell,i}^j, \eta_{b,\ell,i}^j\}_{j=1}^n$ and $\epsilon_{\ell,i}$ for all $\ell \in \{1, \ldots, m\}$ and $i \in \{1, \ldots, k\}$ which satisfy that
   $$c_{\ell,i} = a_{\ell,i} \cdot b_{\ell,i} + \sum_{j=1}^n (\eta_{a,\ell,i}^j \cdot b_\ell^j + \eta_{b,\ell,i}^j \cdot a_\ell^j) + \epsilon_{\ell,i}.$$
   Here a_ℓ^j, b_ℓ^j are P_j's shares of $[\boldsymbol{a}_\ell]_{n-1}, [\boldsymbol{b}_\ell]_{n-1}$ (i.e., P_j's shares before the sharing transformation, or equivalently, the ℓ-th outputs of $\mathtt{Expand}(s_a^j)$ and $\mathtt{Expand}(s_b^j)$).
   (b) In Step 2, $\mathcal{S}$ follows the simulation strategy of π_{Triple}. In particular, for every packed Beaver triple
   $$([\tilde{\boldsymbol{a}}_\ell]_{n-1}, \{\langle \Delta \cdot \tilde{a}_{\ell,i} \rangle\}_{i=1}^k), ([\tilde{\boldsymbol{b}}_\ell]_{n-1}, \{\langle \Delta \cdot \tilde{b}_{\ell,i} \rangle\}_{i=1}^k), ([\tilde{\boldsymbol{c}}_\ell]_{n-1}, \{\langle \Delta \cdot \tilde{c}_{\ell,i} \rangle\}_{i=1}^k),$$
   $\mathcal{S}$ learns the shares of corrupted parties. $\mathcal{S}$ also learns the errors $\{\tilde{\eta}_{a,\ell,i}^j, \tilde{\eta}_{b,\ell,i}^j\}_{j=1}^n$ and $\tilde{\epsilon}_{\ell,i}$ for all $\ell \in \{1, \ldots, m\}$ and $i \in \{1, \ldots, k\}$ which satisfy that
   $$\tilde{c}_{\ell,i} = \tilde{a}_{\ell,i} \cdot \tilde{b}_{\ell,i} + \sum_{j=1}^n (\tilde{\eta}_{a,\ell,i}^j \cdot \tilde{b}_\ell^j + \tilde{\eta}_{b,\ell,i}^j \cdot \tilde{a}_\ell^j) + \tilde{\epsilon}_{\ell,i}.$$
   (c) In Step 3, $\mathcal{S}$ honestly emulates $\mathcal{F}_{\mathsf{Coin}}$ and samples ρ.
   (d) In Step 4, $\mathcal{S}$ computes the shares of $[\tilde{\boldsymbol{a}}_\ell - \rho \cdot \boldsymbol{a}_\ell]_{n-1}$ and $[\tilde{\boldsymbol{b}}_\ell - \boldsymbol{b}_\ell]_{n-1}$ of corrupted parties. $\mathcal{S}$ samples random values as the shares of honest parties and reconstructs the secrets $\tilde{\boldsymbol{a}}_\ell - \rho \cdot \boldsymbol{a}_\ell$ and $\tilde{\boldsymbol{b}}_\ell - \boldsymbol{b}_\ell$. Depending on whether P_1 is honest, there are two cases:
      – If P_1 is honest, $\mathcal{S}$ receives the shares of $[\tilde{\boldsymbol{a}}_\ell - \rho \cdot \boldsymbol{a}_\ell]_{n-1}$ and $[\tilde{\boldsymbol{b}}_\ell - \boldsymbol{b}_\ell]_{n-1}$ of corrupted parties. Then $\mathcal{S}$ reconstructs $\tilde{\boldsymbol{a}}_\ell - \rho \cdot \boldsymbol{a}_\ell$ and $\tilde{\boldsymbol{b}}_\ell - \boldsymbol{b}_\ell$. $\mathcal{S}$ honestly distributes $[\tilde{\boldsymbol{a}}_\ell - \rho \cdot \boldsymbol{a}_\ell]_{k-1}, [\tilde{\boldsymbol{b}}_\ell - \boldsymbol{b}_\ell]_{k-1}$ to all parties.
      – If P_1 is corrupted, $\mathcal{S}$ sends the shares of $[\tilde{\boldsymbol{a}}_\ell - \rho \cdot \boldsymbol{a}_\ell]_{n-1}$ and $[\tilde{\boldsymbol{b}}_\ell - \boldsymbol{b}_\ell]_{n-1}$ of honest parties to P_1 and receives the shares of $[\tilde{\boldsymbol{a}}_\ell - \rho \cdot \boldsymbol{a}_\ell]_{k-1}$ and $[\tilde{\boldsymbol{b}}_\ell - \boldsymbol{b}_\ell]_{k-1}$ of honest parties. $\mathcal{S}$ reconstructs the whole sharings and computes the secrets $\tilde{\boldsymbol{a}}_\ell - \rho \cdot \boldsymbol{a}_\ell$ and $\tilde{\boldsymbol{b}}_\ell - \boldsymbol{b}_\ell$ by using the shares of the first k honest parties.
   (e) In Step 6, $\mathcal{S}$ follows the protocol and computes the shares of $[\boldsymbol{\theta}_\ell]_{n-1}$ of corrupted parties.
   (f) In Step 7, $\mathcal{S}$ follows the protocol and computes the shares of $\langle \Delta \cdot \theta_{\ell,i} \rangle$ of corrupted parties for all $\ell \in \{1, \ldots, m\}$ and $i \in \{1, \ldots, k\}$.

   Then $\mathcal{S}$ simulates $\pi_{\mathsf{VerifyDeg}}$ as follows.
   (a) In Step 1, $\mathcal{S}$ honestly emulates $\mathcal{F}_{\mathsf{Coin}}$ and samples random values $\chi_1, \ldots, \chi_m$.
   (b) In Step 2, recall that $\mathcal{S}$ learns the shares of $[\boldsymbol{x}_\ell]_{k-1}$ of honest parties when simulating $\pi_{\mathsf{Sacrifice}}$. $\mathcal{S}$ computes the shares of $[\boldsymbol{x}]_{k-1}$ of honest parties. $\mathcal{S}$ also computes the shares of $\{\langle \Delta \cdot x_i \rangle\}_{i=1}^k$ of corrupted parties.
   (c) In Step 3, $\mathcal{S}$ honestly exchange the shares of honest parties with corrupted parties. If the shares of $[\boldsymbol{x}]_{k-1}$ does not form a valid degree-$(k-1)$ packed Shamir sharing, $\mathcal{S}$ aborts. If there exists ℓ such that the shares of $[\boldsymbol{x}_\ell]_{k-1}$ of honest parties do not lie on a degree-$(k-1)$ polynomial but the check passes, $\mathcal{S}$ also aborts.
   $\mathcal{S}$ reconstructs the secrets $\overline{\boldsymbol{x}}$, which is equal to $\sum_{\ell=1}^m \chi_\ell \cdot \overline{\boldsymbol{x}_\ell}$. On the other hand, $\mathcal{S}$ also learns $\boldsymbol{x}_\ell$ when simulating $\pi_{\mathsf{Sacrifice}}$. $\mathcal{S}$ computes $\boldsymbol{x} = \sum_{\ell=1}^m \chi_\ell \cdot \boldsymbol{x}_\ell$.
   (d) In Step 4, for all $i \in \{1, \ldots, k\}$, $\mathcal{S}$ computes the shares of $\langle \delta_i \rangle$ of corrupted parties.
   (e) In Step 5, $\mathcal{S}$ honestly emulates $\mathcal{F}_{\mathsf{Commit}}$ for corrupted parties.
   If $\boldsymbol{x} \neq \overline{\boldsymbol{x}}$, $\mathcal{S}$ randomly samples Δ^i for each honest party P_i and computes $\Delta = \sum_{j=1}^n \Delta^i$. $\mathcal{S}$ sets $\delta_i = \Delta \cdot (x_i - \overline{x_i})$ and randomly samples the shares $\langle \delta_i \rangle$ of honest parties based on the shares of corrupted parties and the secret. Then $\mathcal{S}$ honestly emulates $\mathcal{F}_{\mathsf{Commit}}$ for honest parties.
   (f) $\mathcal{S}$ honestly follows Step 6. If $\boldsymbol{x} \neq \overline{\boldsymbol{x}}$ but the check passes, $\mathcal{S}$ aborts.

Next $\mathcal{S}$ simulates $\pi_{\mathsf{CheckZero}}$ as follows.

(a) In Step 1, $\mathcal{S}$ honestly emulates $\mathcal{F}_{\mathsf{Coin}}$ and samples random values $\chi_1, \ldots, \chi_m$.

(b) In Step 2, $\mathcal{S}$ simulates π_{RandSh} when preparing $[\mathbf{0}]_{n-1}$ as desribed above and learns the shares of corrupted parties.

(c) In Step 3, $\mathcal{S}$ computes the shares of $[\boldsymbol{\theta}]_{n-1}$ and $\{\langle \Delta \cdot \theta_i \rangle\}_{i=1}^{k}$ of corrupted parties.

(d) In Step 4, $\mathcal{S}$ honestly emulates $\mathcal{F}_{\mathsf{Commit}}$ for corrupted parties.
Recall that $\pi_{\mathsf{CheckZero}}$ is invoked only when all parties accept the check in $\pi_{\mathsf{VerifyDeg}}$, which guarantees that all degree-$(k-1)$ packed Shamir sharings distributed by P_1 in $\pi_{\mathsf{Sacrifice}}$ are correct. Then for all $\ell \in \{1, \ldots, m\}$ and for all $i \in \{1, \ldots, k\}$,

$$\begin{aligned} \theta_{\ell,i} = \rho \cdot (\sum_{j=1}^{n} (\eta^{j}_{a,\ell,i} \cdot b^{j}_{\ell} + \eta^{j}_{b,\ell,i} \cdot a^{j}_{\ell}) + \epsilon_{\ell,i}) \\ - \sum_{j=1}^{n} (\tilde{\eta}^{j}_{a,\ell,i} \cdot \tilde{b}^{j}_{\ell} + \tilde{\eta}^{j}_{b,\ell,i} \cdot \tilde{a}^{j}_{\ell}) - \tilde{\epsilon}_{\ell,i}. \end{aligned}$$

If any of $\{\eta^{j}_{a,\ell,i}, \eta^{j}_{b,\ell,i}\}_{\ell \in \{1,\ldots,m\}, i \in \{1,\ldots,k\}}$ or any of $\{\tilde{\eta}^{j}_{a,\ell,i}, \tilde{\eta}^{j}_{b,\ell,i}\}_{\ell \in \{1,\ldots,m\}, i \in \{1,\ldots,k\}}$ is not 0, $\mathcal{S}$ randomly samples the shares of $[\boldsymbol{a}_\ell]_{n-1}, [\boldsymbol{b}_\ell]_{n-1}, [\tilde{\boldsymbol{a}}_\ell]_{n-1}, [\tilde{\boldsymbol{b}}_\ell]_{n-1}$ of honest parties. Then $\mathcal{S}$ computes $\theta_{\ell,i}$ and θ_i accordingly. Next, $\mathcal{S}$ randomly samples the shares of $[\boldsymbol{\theta}]_{n-1}$ of honest parties based on the shares of corrupted parties and the secrets $\boldsymbol{\theta}$. $\mathcal{S}$ honestly emulates $\mathcal{F}_{\mathsf{Commit}}$ for honest parties.

(e) In Step 5, $\mathcal{S}$ honestly follows the protocol. If any of $\{\eta^{j}_{a,\ell,i}, \eta^{j}_{b,\ell,i}\}_{\ell \in \{1,\ldots,m\}, i \in \{1,\ldots,k\}}$ or any of $\{\tilde{\eta}^{j}_{a,\ell,i}, \tilde{\eta}^{j}_{b,\ell,i}\}_{\ell \in \{1,\ldots,m\}, i \in \{1,\ldots,k\}}$ is not 0 but the check passes, $\mathcal{S}$ aborts.

(f) In Step 6, $\mathcal{S}$ honestly emulates $\mathcal{F}_{\mathsf{Commit}}$ for corrupted parties.
If any of $\{\epsilon_{\ell,i}\}_{\ell \in \{1,\ldots,m\}, i \in \{1,\ldots,k\}}$ or any of $\{\tilde{\epsilon}_{\ell,i}\}_{\ell \in \{1,\ldots,m\}, i \in \{1,\ldots,k\}}$ is not 0, $\mathcal{S}$ randomly samples Δ^i for each honest party P_i and computes $\Delta = \sum_{j=1}^{n} \Delta^i$. $\mathcal{S}$ sets $\Delta \cdot \theta_{\ell,i} = \Delta \cdot (\rho \cdot \epsilon_{\ell,i} - \tilde{\epsilon}_{\ell,i})$ and computes $\Delta \cdot \theta_i$ accordingly. Then $\mathcal{S}$ randomly samples the shares $\langle \Delta \cdot \theta_i \rangle$ of honest parties based on the shares of corrupted parties and the secret. Then $\mathcal{S}$ honestly emulates $\mathcal{F}_{\mathsf{Commit}}$ for honest parties.

(g) In Step 7, $\mathcal{S}$ honestly follows the protocol. If any of $\{\epsilon_{\ell,i}\}_{\ell \in \{1,\ldots,m\}, i \in \{1,\ldots,k\}}$ or any of $\{\tilde{\epsilon}_{\ell,i}\}_{\ell \in \{1,\ldots,m\}, i \in \{1,\ldots,k\}}$ is not 0 but the check passes, $\mathcal{S}$ aborts.

5. In Step 4 and Step 5, $\mathcal{S}$ simulates π_{RandSh} when preparing random sharings $[\mathbf{0}]_{n-1}$ as described above and learns the shares of corrupted parties.
6. Finally, $\mathcal{S}$ provides the shares of corrupted parties to $\mathcal{F}_{\mathsf{PrepIndMal}}$.

This completes the description of $\mathcal{S}$.

Now we use hybrid arguments to prove the security of $\Pi_{\mathsf{PrepIndMal}}$.

Hybrid$_0$: In this hybrid, $\mathcal{S}$ honestly follows the protocol. This corresponds to the real world.

Hybrid$_1$: In this hybrid, for every ordered pair of honest parties (P_i, P_j), when P_i needs to send a random element to P_j, instead of evaluating the PRG on their common random seed, they use a real random element. By the security of the PRG, the distribution of **Hybrid$_1$** and distribution of **Hybrid$_0$** are computationally indistinguishable.

Hybrid$_2$: In this hybrid, π_{RandSh} is simulated by $\mathcal{S}$ using the general strategy described above. Note that the simulation does not require $\mathcal{S}$ to sample the shares of honest parties. $\mathcal{S}$ pushes the sampling of Δ^i to whenever it is needed. When needed, $\mathcal{S}$ samples random Σ-sharings and computes the shares of honest parties based on the shares of corrupted parties. Now we analyze the distribution of **Hybrid$_2$** and the distribution of **Hybrid$_1$**.

In **Hybrid$_1$**, whenever an honest party P_i needs to distribute a random Σ-sharing, since $t \geq n/2$ and the random Σ-sharing we need to prepare satisfies that the shares of corrupted parties are independent of the secrets, therefore the shares of corrupted parties are uniformly random (which are either computed by PRG on the common random seeds or received from P_i). Thus, the distribution of the messages sent by P_i in **Hybrid$_1$** is identical to that in **Hybrid$_2$**.

Now we show that the distribution of the shares of honest parties in **Hybrid$_1$** is identical to that in **Hybrid$_2$**. Let $\boldsymbol{M}_{\mathcal{C}orr}$ denote the submatrix of $\boldsymbol{M}$ that contains the columns corresponding to corrupted parties' indices. Similarly, let $\boldsymbol{M}_{\mathcal{H}}$ denote the submatrix of $\boldsymbol{M}$ that contains the columns corresponding to honest parties' indices. Then we have

$$\begin{aligned}(\boldsymbol{R}^{(1)}, \ldots, \boldsymbol{R}^{(n-t)})^{\mathrm{T}} &= \boldsymbol{M}(\boldsymbol{S}^{(1)}, \ldots, \boldsymbol{S}^{(n)})^{\mathrm{T}} \\ &= \boldsymbol{M}_{\mathcal{C}orr} \cdot \left((\boldsymbol{S}^{(j)})_{j \in \mathcal{C}orr}\right)^{\mathrm{T}} + \boldsymbol{M}_{\mathcal{H}} \cdot \left((\boldsymbol{S}^{(j)})_{j \in \mathcal{H}}\right)^{\mathrm{T}}.\end{aligned}$$

Let $(\boldsymbol{R}^{(1)}_{\mathcal{C}orr}, \ldots, \boldsymbol{R}^{(n-t)}_{\mathcal{C}orr})^{\mathrm{T}} = \boldsymbol{M}_{\mathcal{C}orr} \cdot \left((\boldsymbol{S}^{(j)})_{j \in \mathcal{C}orr}\right)^{\mathrm{T}}$ and $(\boldsymbol{R}^{(1)}_{\mathcal{H}}, \ldots, \boldsymbol{R}^{(n-t)}_{\mathcal{H}})^{\mathrm{T}} = \boldsymbol{M}_{\mathcal{H}} \cdot \left((\boldsymbol{S}^{(j)})_{j \in \mathcal{H}}\right)^{\mathrm{T}}$. Then each $\boldsymbol{R}^{(i)}_{\mathcal{H}}$ is a random Σ-sharing given the shares of corrupted parties. As for $\boldsymbol{R}^{(i)}_{\mathcal{C}orr}$, $\mathcal{S}$ receives the shares of honest parties from corrupted parties. For any fixed $\boldsymbol{R}^{(i)}_{\mathcal{C}orr}$, $\boldsymbol{R}^{(i)}_{\mathcal{C}orr} + \boldsymbol{R}^{(i)}_{\mathcal{H}}$ is a random Σ-sharing given the shares of corrupted parties. In **Hybrid**$_2$, $\mathcal{S}$ generates the shares of $\boldsymbol{R}^{(i)}_{\mathcal{C}orr}$ of corrupted parties by randomly sampling their shares of $(\boldsymbol{S}^{(j)})_{j \in \mathcal{C}orr}$ based on the shares of honest parties. Then the shares of $\boldsymbol{R}^{(i)} = \boldsymbol{R}^{(i)}_{\mathcal{C}orr} + \boldsymbol{R}^{(i)}_{\mathcal{H}}$ of honest parties are randomly sampled based on the shares of corrupted parties. Thus, the distribution of the shares of honest parties in **Hybrid**$_1$ is identical to that in **Hybrid**$_2$

Therefore, the distribution of **Hybrid**$_2$ is identical to that of **Hybrid**$_1$.

Hybrid$_3$: In this hybrid, $\mathcal{S}$ simulates **Initialization** in π_{Triple} as described above. Note that the simulation does not require $\mathcal{S}$ to sample Δ^i for each honest party and the shares of honest parties. $\mathcal{S}$ pushes the sampling of Δ^i and the shares of honest parties to whenever it is needed. When needed, $\mathcal{S}$ samples a random value as Δ^i and randomly generates the shares of honest parties based on the shares of corrupted parties (in the same way as $\mathcal{F}_{\mathsf{PrepIndMal}}$). Now we analyze the distribution of **Hybrid**$_3$ and the distribution of **Hybrid**$_2$.

Note that the only steps that require communication are Step 2 and Stpe 3 in π_{MACKey}. In **Hybrid**$_2$, all parties compute additive sharings $\langle \Delta + r \rangle = \langle \Delta \rangle + \langle r \rangle$ and send their shares to P_1. Since $\langle r \rangle$ is a random additive sharing, $\langle \Delta + r \rangle$ is also a random additive sharing. In particular, each share of $\langle \Delta + r \rangle$ is uniformly distributed. In **Hybrid**$_3$, $\mathcal{S}$ simply samples random values as the shares of honest parties. Thus, the distribution of the messages sent to P_1 is identical in both hybrids. Then P_1 reconstructs the secret $\Delta + r$ and distributes $[(\Delta + r) \cdot \mathbf{1}]_{2k-2}$ to all parties. We consider two cases:

- If P_1 is honest, then P_1 honestly follow the protocol in both cases.
- If P_1 is corrupted, since P_1 receives values with the same distribution in both hybrids, it performs identically in both hybrids when distributing $[(\Delta + r) \cdot \mathbf{1}]_{2k-2}$.

Thus, the messages exchanged between honest parties and corrupted parties in both hybrids are identically distributed.

Now we show that the distribution of the shares of honest parties in **Hybrid**$_3$ is identical to that in **Hybrid**$_2$. Regarding Δ^i of each honest party P_i, it is uniformly distributed in both **Hybrid**$_2$ and **Hybrid**$_3$. Regarding the shares of $\{[\Delta|_i]_t\}_{i=1}^k$ of honest parties, observe that they are fully decided by the shares of corrupted parties and the secret. In **Hybrid**$_2$, due to the malicious behaviors of corrupted parties, the secret of $[\Delta|_i]_t$ may not be Δ. Instead, there may be an additive error δ_i. In **Hybrid**$_3$, $\mathcal{S}$ computes the additive error δ_i. Then the shares of honest parties is determined by the shares of corrupted parties and $\Delta + \delta_i$. Since $t \geq n/2$, there is a degree-t Shamir sharing $[\delta_i|_i]_t$ such that the shares of honest parties are all 0s. Therefore, we may adjust the shares of corrupted parties such that the secret is Δ without changing the shares of honest parties. This is exactly the process done by $\mathcal{S}$. Therefore, the shares of $\{[\Delta|_i]_t\}_{i=1}^k$ of honest parties are identically distributed in both hybrids.

Therefore, the distribution of **Hybrid**$_3$ is identical to that of **Hybrid**$_2$.

Hybrid$_4$: In this hybrid, for an honest party P_i, we replace the output of $\mathtt{Expand}(\mathtt{seed}^i)$ in $\mathcal{F}_{\mathsf{nVOLE}}$ and $\mathcal{F}^{\mathsf{prog}}_{\mathsf{OLE}}$ by uniform values. By the security of the PRG, the distribution of **Hybrid**$_4$ and distribution of **Hybrid**$_3$ are computationally indistinguishable.

Hybrid$_5$: In this hybrid, $\mathcal{S}$ simulates π_{RandAuth} as described above. Note that the simulation does not require $\mathcal{S}$ to generate the shares of honest parties. $\mathcal{S}$ pushes the sampling of the shares of honest parties to whenever it is needed. When needed, $\mathcal{S}$ randomly generates the shares of honest parties based on the shares of corrupted parties (in the same way as $\mathcal{F}_{\mathsf{PrepIndMal}}$). Now we analyze the distribution of **Hybrid**$_5$ and the distribution of **Hybrid**$_4$.

Compared with $\mathbf{Hybrid}_4$, $\mathcal{S}$ always abort when it receives a global key query. Observe that $\mathcal{S}$ only generates $\{\Delta^i\}_{i\in\mathcal{H}}$ when corrupted parties make a global key query. The probability that the query makes a successful guess is negligible due to the exponential size of $\mathbb{F}$.

Then in Step 2(a) of π_{RandAuth}, all parties compute $[\boldsymbol{r}_\ell+\boldsymbol{\rho}]_{n-1}=[\boldsymbol{r}_\ell]_{n-1}+\sum_{i=1}^k \boldsymbol{e}_i*[\rho_i\cdot\mathbf{1}]_{n-k}$ and send their shares to P_1. Since $[\boldsymbol{r}_\ell]_{n-1}$ is a random degree-$(n-1)$ packed Shamir sharing, it satisfies that all shares are uniformly distributed. In $\mathbf{Hybrid}_5$, $\mathcal{S}$ simply samples random values as the shares of honest parties. Therefore, the messages sent to P_1 have the same distribution in both hybrids. Next in Step 2(b) of π_{RandAuth}, P_1 reconstructs the secrets $\boldsymbol{r}_\ell+\boldsymbol{\rho}$ and sends $\boldsymbol{r}_\ell+\boldsymbol{\rho}$ to all parties. We consider two cases:

- If P_1 is honest, then P_1 honestly follow the protocol in both cases.
- If P_1 is corrupted, since P_1 receives values with the same distribution in both hybrids, it performs identically in both hybrids when sending $\boldsymbol{r}_\ell+\boldsymbol{\rho}$.

Thus, the messages sent by P_1 in both hybrids are identically distributed.

In Step 3 of π_{RandAuth}, all parties check whether P_1 sends the same values to all parties. $\mathcal{S}$ simply follows the protocol. The only difference is that if P_1 does not send the same values to all honest parties but the check passes, $\mathcal{S}$ aborts. Observe that it happens only with negligible probability.

Now we show that the distribution of the shares of honest parties in $\mathbf{Hybrid}_5$ is identical to that in $\mathbf{Hybrid}_4$. Regarding $\langle\Delta\cdot r_{\ell,i}\rangle$, the shares of corrupted parties are generated by $\mathcal{F}_{\mathsf{nVOLE}}$, which is emulated by $\mathcal{S}$ in $\mathbf{Hybrid}_5$. Since $r_{\ell,i}$ is uniformly distributed in both hybrids and the distribution of the shares of honest parties is determined by the shares of corrupted parties and the secret $\Delta\cdot r_{\ell,i}$, the shares of $\langle\Delta\cdot r_{\ell,i}\rangle$ of honest parties are identically distributed in both hybrids. Regarding the shares of $[r_{\ell,i}\cdot\mathbf{1}]_{n-k}$ of honest parties, observe that their distribution is determined by the shares of corrupted parties and the secret. In $\mathbf{Hybrid}_4$, due to the malicious behaviors of corrupted parties, the secret of $[r_{\ell,i}\cdot\mathbf{1}]_{n-k}$ may not be $r_{\ell,i}\cdot\mathbf{1}$. Instead, there may be an additive error. Following the same argument as that in $\mathbf{Hybrid}_3$, $\mathcal{S}$ can adjust the shares of corrupted parties such that the secret is $r_{\ell,i}\cdot\mathbf{1}$ without changing the shares of honest parties. This is exactly the process done by $\mathcal{S}$. Therefore, the shares of $[r_{\ell,i}\cdot\mathbf{1}]_{n-k}$ of honest parties are identically distributed in both hybrids.

Therefore, the distribution of $\mathbf{Hybrid}_5$ is statistically close to that of $\mathbf{Hybrid}_4$.

$\mathbf{Hybrid}_6$: In this hybrid, $\mathcal{S}$ simulates π_{Triple} as described above. Note that the simulation does not require $\mathcal{S}$ to generate the shares of honest parties. $\mathcal{S}$ pushes the sampling of the shares of honest parties to whenever it is needed. When needed, $\mathcal{S}$ generates the shares of honest parties as follows:

1. $\mathcal{S}$ randomly samples Δ^i for each honest party P_i if it has not been generated.
2. $\mathcal{S}$ samples random values as the shares of $[\boldsymbol{a}_\ell]_{n-1},[\boldsymbol{b}_\ell]_{n-1}$ of honest parties. Then $\mathcal{S}$ reconstructs the secrets $\boldsymbol{a}_\ell,\boldsymbol{b}_\ell$.
3. For $([\boldsymbol{a}_\ell]_{n-k},[\Delta\cdot\boldsymbol{a}_\ell]_{n-k})$, $\mathcal{S}$ computes $\Delta\cdot\boldsymbol{a}_\ell$. Then based on the shares of corrupted parties and the secrets, $\mathcal{S}$ computes the shares of honest parties. $\mathcal{S}$ computes the shares of $([\boldsymbol{b}_\ell]_{n-k},[\Delta\cdot\boldsymbol{b}_\ell]_{n-k})$ of honest parties similarly.
4. For $([\boldsymbol{c}_\ell]_{n-1},\{\langle\Delta\cdot c_{\ell,i}\rangle\}_{i=1}^k)$, $\mathcal{S}$ computes $c_{\ell,i}=a_{\ell,i}\cdot b_{\ell,i}+\sum_{j=1}^n(\eta_{a,\ell,i}^j\cdot b_\ell^j+a_\ell^j\cdot\eta_{b,\ell,i}^j)+\epsilon_{\ell,i}$ and $\Delta\cdot c_{\ell,i}$. Here a_ℓ^j,b_ℓ^j are P_j's shares of $[\boldsymbol{a}_\ell]_{n-1},[\boldsymbol{b}_\ell]_{n-1}$. Then based on the shares of corrupted parties, $\mathcal{S}$ randomly samples the shares of honest parties.

Now we analyze the distribution of $\mathbf{Hybrid}_6$ and the distribution of $\mathbf{Hybrid}_5$.

The first two steps in π_{Triple} requires no interactions between honest parties and corrupted parties. From Step 3 to Step 5, $\mathcal{S}$ can simulate $\pi_{\mathsf{DegReduce}}$, π_{AddTran}, and π_{Auth} similarly to π_{MACKey} and π_{RandAuth}: Whenever $\mathcal{S}$ needs to prepare the shares of honest parties and send them to P_1, $\mathcal{S}$ samples random values as their shares. In particular, $\mathcal{S}$ can adjust the shares of corrupted parties in $\pi_{\mathsf{DegReduce}}$ and π_{AddTran} to cancel the additive errors introduced due to the malicious behaviors of corrupted parties. In π_{Auth}, $\mathcal{S}$ extracts the additive errors $\boldsymbol{\epsilon}_\ell$ for all $\ell\in\{1,\ldots,m\}$.

Now we show that the distribution of the shares of honest parties in $\mathbf{Hybrid}_6$ is identical to that in $\mathbf{Hybrid}_5$. For $[\boldsymbol{a}_\ell]_{n-1},[\boldsymbol{b}_\ell]_{n-1}$, the shares of honest parties are uniformly random in $\mathbf{Hybrid}_5$. In $\mathbf{Hybrid}_6$, $\mathcal{S}$ just samples random values as the shares of honest parties. This also implies that the secrets $\boldsymbol{a}_\ell,\boldsymbol{b}_\ell$ are identically distributed in both hybrids. For $([\boldsymbol{a}_\ell]_{n-k},[\Delta\cdot\boldsymbol{a}_\ell]_{n-k})$ and $([\boldsymbol{b}_\ell]_{n-k},[\Delta\cdot\boldsymbol{b}_\ell]_{n-k})$, the shares of honest parties are determined by the shares of corrupted parties and the secrets. Since $\mathcal{S}$ perfectly simulates the behaviors of honest parties, the shares of corrupted parties are identically distributed in both hybrids. Together with the fact that $\boldsymbol{a}_\ell, \boldsymbol{b}_\ell$ are also identically distributed, we conclude that the shares of $([\boldsymbol{a}_\ell]_{n-k}, [\Delta \cdot \boldsymbol{a}_\ell]_{n-k})$ of honest parties are identically distributed. For $([\boldsymbol{c}_\ell]_{n-1}, \{\langle \Delta \cdot c_{\ell,i} \rangle\}_{i=1}^k)$, $\mathcal{S}$ computes the actual values $\boldsymbol{c}_\ell$ and $\Delta \cdot \boldsymbol{c}_\ell$ using the sharings $[\boldsymbol{a}_\ell]_{n-1}, [\boldsymbol{b}_\ell]_{n-1}$ and the errors introduced by corrupted parties' behaviors. In **Hybrid**$_5$, $([\boldsymbol{c}_\ell]_{n-1}, \{\langle \Delta \cdot c_{\ell,i} \rangle\}_{i=1}^k)$ are random sharings given the shares of corrupted parties and the secrets. $\mathcal{S}$ generates the shares of honest parties in the same way. Therefore, the shares of $([\boldsymbol{a}_\ell]_{n-k}, [\Delta \cdot \boldsymbol{a}_\ell]_{n-k})$ of honest parties are identically distributed.

Thus, the distribution of **Hybrid**$_6$ is identical to that of **Hybrid**$_5$.

Hybrid$_7$: In this hybrid, $\mathcal{S}$ simulates $\pi_{\mathsf{Sacrifice}}$, $\pi_{\mathsf{VerifyDeg}}$, and $\pi_{\mathsf{CheckZero}}$ as described above. Note that any values generated in these three protocols are not needed in the further steps. It is sufficient to only show that the messages sent to corrupted parties have the same distributions in both hybrids.

In $\pi_{\mathsf{Sacrifice}}$, $\mathcal{S}$ simulates Step 2 in the same way as that in π_{Triple}. As we argued above, the messages sent to corrupted parties have the same distributions in both hybrids. In Step 4, since $[\tilde{\boldsymbol{a}}_\ell]_{n-1}, [\tilde{\boldsymbol{b}}_\ell]_{n-1}$ are random degree-$(n-1)$ packed Shamir sharings, the shares of $[\tilde{\boldsymbol{a}}_\ell - \rho \cdot \boldsymbol{a}_\ell]_{n-1}, [\tilde{\boldsymbol{b}}_\ell - \boldsymbol{b}_\ell]_{n-1}$ of honest parties are uniform in **Hybrid**$_6$. In **Hybrid**$_7$, $\mathcal{S}$ samples random values as shares of honest parties. Therefore, the shares of $[\tilde{\boldsymbol{a}}_\ell - \rho \cdot \boldsymbol{a}_\ell]_{n-1}, [\tilde{\boldsymbol{b}}_\ell - \boldsymbol{b}_\ell]_{n-1}$ of honest parties are identically distributed in both hybrids. Depending on whether P_1 is honest, there are two cases:

- If P_1 is honest, then P_1 honestly follow the protocol in both cases.
- If P_1 is corrupted, since P_1 receives values with the same distribution in both hybrids, it performs identically in both hybrids when distributing $[\tilde{\boldsymbol{a}}_\ell - \rho \cdot \boldsymbol{a}_\ell]_{k-1}, [\tilde{\boldsymbol{b}}_\ell - \boldsymbol{b}_\ell]_{k-1}$.

In $\pi_{\mathsf{VerifyDeg}}$, $\mathcal{S}$ honestly follows the protocol in the first 3 steps. The only difference is that if there exists ℓ such that the shares of $[\boldsymbol{x}_\ell]_{k-1}$ of honest parties do not lie on a degree-$(k-1)$ polynomial but the check passes, $\mathcal{S}$ also aborts. Note that it only happens with negligible probability. In Step 5, if $\boldsymbol{x} \neq \overline{\boldsymbol{x}}$, $\mathcal{S}$ samples Δ^i for each honest party and then follows the protocol. Note that the distribution of Δ^i is identical in both hybrids. In Step 6, $\mathcal{S}$ follows the protocol. The only difference is that $\mathcal{S}$ will abort if $\boldsymbol{x} \neq \overline{\boldsymbol{x}}$ but the check passes. It only happens if the secret of $\langle \delta_i \rangle$ is 0 for all $i \in \{1, \ldots, k\}$. However, the secret $\delta_i = \Delta \cdot (x_i - \overline{x_i})$, and what corrupted parties can do is to add additive error on δ_i. Since Δ is not known to corrupted parties, this happens with negligible probability.

In $\pi_{\mathsf{CheckZero}}$, $\mathcal{S}$ honestly follows the protocol in the first 3 steps.

- In Step 4, if any of $\{\eta^j_{a,\ell,i}, \eta^j_{b,\ell,i}\}_{\ell \in \{1,\ldots,m\}, i \in \{1,\ldots,k\}}$ or any of $\{\tilde{\eta}^j_{a,\ell,i}, \tilde{\eta}^j_{b,\ell,i}\}_{\ell \in \{1,\ldots,m\}, i \in \{1,\ldots,k\}}$ is not 0, $\mathcal{S}$ randomly samples the shares of $[\boldsymbol{a}_\ell]_{n-1}, [\boldsymbol{b}_\ell]_{n-1}, [\tilde{\boldsymbol{a}}_\ell]_{n-1}, [\tilde{\boldsymbol{b}}_\ell]_{n-1}$ of honest parties. Then $\mathcal{S}$ computes $\theta_{\ell,i}$ and θ_i accordingly. Next, $\mathcal{S}$ randomly samples the shares of $[\boldsymbol{\theta}]_{n-1}$ of honest parties based on the shares of corrupted parties and the secrets $\boldsymbol{\theta}$. Observe that the shares of honest parties generated by $\mathcal{S}$ have the same distribution as that in **Hybrid**$_6$.
- In Step 5, $\mathcal{S}$ follows the protocol. The only difference is that if any of $\{\eta^j_{a,\ell,i}, \eta^j_{b,\ell,i}\}_{\ell \in \{1,\ldots,m\}, i \in \{1,\ldots,k\}}$ or any of $\{\tilde{\eta}^j_{a,\ell,i}, \tilde{\eta}^j_{b,\ell,i}\}_{\ell \in \{1,\ldots,m\}, i \in \{1,\ldots,k\}}$ is not 0 but the check passes, $\mathcal{S}$ aborts. Recall that
$$\begin{aligned}\theta_{\ell,i} = \rho \cdot (&\sum_{j=1}^n (\eta^j_{a,\ell,i} \cdot b^j_\ell + \eta^j_{b,\ell,i} \cdot a^j_\ell) + \epsilon_{\ell,i}) \\ &- \sum_{j=1}^n (\tilde{\eta}^j_{a,\ell,i} \cdot \tilde{b}^j_\ell + \tilde{\eta}^j_{b,\ell,i} \cdot \tilde{a}^j_\ell) - \tilde{\epsilon}_{\ell,i}.\end{aligned}$$
If any of $\{\eta^j_{a,\ell,i}, \eta^j_{b,\ell,i}\}_{\ell \in \{1,\ldots,m\}, i \in \{1,\ldots,k\}}$ or any of $\{\tilde{\eta}^j_{a,\ell,i}, \tilde{\eta}^j_{b,\ell,i}\}_{\ell \in \{1,\ldots,m\}, i \in \{1,\ldots,k\}}$ is not 0, with overwhelming probability, there exists ℓ and i such that $\theta_{\ell,i}$ is not 0. Also recall that only when P_j is an honest party, can the values $\eta^j_{a,\ell,i}, \eta^j_{b,\ell,i}, \tilde{\eta}^j_{a,\ell,i}, \tilde{\eta}^j_{b,\ell,i}$ be non-zero. Since $a^j_\ell, b^j_\ell, \tilde{a}^j_\ell, \tilde{b}^j_\ell$ are random values and unknown to corrupted parties, $\theta_{\ell,i}$ is statistically close to a uniform value and unknown to corrupted parties. Therefore, with overwhelming probability, θ_i is statistically close to a uniform value and unknown to corrupted parties. Thus, the probability that the secret of $[\boldsymbol{\theta}]_{n-1}$ is an all-0 vector is negligible. Therefore, the probability that $\mathcal{S}$ aborts in Step 5 in **Hybrid**$_7$ but not in the same step in **Hybrid**$_6$ is negligible.
- In Step 6, if any of $\{\epsilon_{\ell,i}\}_{\ell \in \{1,\ldots,m\}, i \in \{1,\ldots,k\}}$ or any of $\{\tilde{\epsilon}_{\ell,i}\}_{\ell \in \{1,\ldots,m\}, i \in \{1,\ldots,k\}}$ is not 0, $\mathcal{S}$ samples Δ^i for each honest party and then follows the protocol. Note that the distribution of Δ^i is

identical in both hybrids. In Step 7, $\mathcal{S}$ follows the protocol. The only difference is that $\mathcal{S}$ will abort if any of $\{\epsilon_{\ell,i}\}_{\ell\in\{1,\dots,m\},i\in\{1,\dots,k\}}$ or any of $\{\tilde{\epsilon}_{\ell,i}\}_{\ell\in\{1,\dots,m\},i\in\{1,\dots,k\}}$ is not 0 but the check passes. It only happens if the secret of $\langle \Delta \cdot \theta_i \rangle$ is 0 for all $i \in \{1,\dots,k\}$. However, if any of $\{\epsilon_{\ell,i}\}_{\ell\in\{1,\dots,m\},i\in\{1,\dots,k\}}$ or any of $\{\tilde{\epsilon}_{\ell,i}\}_{\ell\in\{1,\dots,m\},i\in\{1,\dots,k\}}$ is not 0, with overwhelming probability, there exists i such that $\theta_i \neq 0$. Then the secret $\Delta \cdot \theta_i$ is statistically close to a uniform value and unknown to corrupted parties. Since what corrupted parties can do is to add additive error on $\Delta \cdot \theta_i$, the probability that the secret of $\langle \Delta \cdot \theta_i \rangle$ is 0 is negligible. Therefore, the probability that $\mathcal{S}$ aborts in Step 7 in **Hybrid**$_7$ but not in the same step in **Hybrid**$_6$ is negligible.

In summary, the distribution of **Hybrid**$_7$ is statistically close to **Hybrid**$_6$.

Hybrid$_8$: Note that in **Hybrid**$_7$, $\mathcal{S}$ has already simulated all the messages sent from honest parties to corrupted parties and computed the shares of corrupted parties. In **Hybrid**$_8$, instead of generating the shares of honest parties by $\mathcal{S}$, $\mathcal{S}$ provides the shares of corrupted parties to $\mathcal{F}_{\mathsf{PrepIndMal}}$. The distribution of **Hybrid**$_8$ is identical to that of **Hybrid**$_7$.

Observe that **Hybrid**$_8$ is the execution in the ideal world. Therefore, $\Pi_{\mathsf{PrepIndMal}}$ securely computes $\mathcal{F}_{\mathsf{PrepIndMal}}$ in the $\{\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}, \mathcal{F}_{\mathsf{nVOLE}}, \mathcal{F}_{\mathsf{Commit}}, \mathcal{F}_{\mathsf{Coin}}\}$-hybrid model against a malicious adversary who controls t out of n parties. □

E Best Existing Dishonest Majority Protocol

In this section we present what, to the best of our knowledge, would constitute the protocol with the smallest online communication in the setting $t = n(1-\epsilon) - 1$, for $0 \le \epsilon < 1/2$. Most existing works in the dishonest majority setting are not able to exploit a gap larger than 1 between the corruption threshold t and the number of parties n. The only exception is [GPS22], which is unfortunately not concretely efficient as we discussed in Section 1. From the above, our starting point is to use a maximal-adversary dishonest majority protocol, but using a reduced set of $n' = t + 1 = n(1-\epsilon)$ parties instead of the full set of n parties.

Regarding the most online-efficient dishonest majority protocol to date, to the best of our knowledge this would be the Turbospeedz protocol [BNO19], which modifies the online phase of the SPDZ [DKL+13] family of protocols so that their online communication is reduced by a factor of $2\times$, assuming circuit-dependent preprocessing, as we do in our work. This circuit-dependent preprocessing consists to one multiplication triple per multiplication gate in the circuit whose factors are uniformly random, but tied to the topology of the circuit. This preprocessing material can be easily generated in a standard way assuming traditional circuit-independent multiplication triples, as proposed in [BNO19].

The instantiation of the circuit-independent offline phase, which consists of generating multiplication triples, can be done in a variety of ways. In our setting, we compare against OLE-based protocols since our protocol SuperPack already uses these tools, which allows for a more fair comparison. In this context, the most efficient protocol to the best of our knowledge is that of Le Mans [RS22], which enables the use of pseudo-random correlator generators (PCGs) to save in communication. Furthermore, we already use ideas from this protocol in SuperPack to instantiate our circuit-independent preprocessing.

As a remark, we notice that the preprocessing in Le Mans produces triples where the two factors are authenticated, but the product itself is not, which means the triple itself could be faulty. This is dealt with in the online phase by making use of the techniques from [CGH+18], which verifies the correctness of the multiplications by executing in parallel a randomized version of the circuit. Since this resutls in a communication increase in the online phase, which we aim at minimizing, we consider instead a variant where the online phase is the one from Turbospeedz [BNO19] with no changes, and the triples are fully authenticated and verified using sacrificing techniques in the circuit-independent offline phase.

From now ownwards in this section we consider the set of parties to be $\mathcal{P}' = \{P_1, \dots, P_{n'}\}$, with $n' = t + 1 \approx n(1-\epsilon)$. We use $[x]$ to denote additive secret-sharing of x among these parties, and we use $[\![x]\!]$ to denote SPDZ-like authenticated shares, meaning $[\![x]\!] = ([x], [\Delta \cdot x], [\Delta])$, where $\Delta \in_R \mathbb{F}$ is a global uniformly random key. Below we describe the online, circuit-dependent and circuit-independent offline phases of the baseline protocol sketched above. We refer the resulting protocol as **Turbospeedz**, recalling that its offline phase is implemented using ideas from [RS22]. For the protocol, we assume the existence of a broadcast channel.

E.1 Online Phase

The online phase $\Pi_{\mathsf{TSPDZ-Online}}$ is presented as Protocol 4 is taken from [BNO19], with several modifications. The invariant is simple. Each wire α which is the output of a multiplication or input gate has associated to it a random mask $\lambda_\alpha \in_R \mathbb{F}$, and for each wire γ with is the result of adding wires α and β the value λ_γ is defined recursively as $\lambda_\gamma = \lambda_\alpha + \lambda_\beta$. For each wire α, the parties in $\mathcal{P}'$ are assumed to have authenticated shares $[\![\lambda_\alpha]\!]$, and the invariant in the online phase will be that the parties also know *in the clear* the value $\mu_\alpha = v_\alpha - \lambda_\alpha$, where v_α is the actual value stored in wire α for a given choice of input values. This invariant is maintained non-interactively for addition gates, and for multiplication gates the parties can make use of certain preprocessing together with interaction involving the reconstruction of some value.

Once the invariant has been carried down to each output wire α, the parties can reconstruct the associated mask $[\![\lambda_\alpha]\!]$ to the corresponding client owning the given output gate, and at least one of the party also sends the cleartext value μ_α. In order to verify the correctness of the computation, the parties also carry over MACs of certain messages sent during the computation, which are verified at the end of the protocol execution, before reconstructing the outputs.

Circuit-dependent preprocessing functionality. In order to maintain the invariant sketched above, the parties will need to make use of certain preprocessing functionality that distributes circuit-dependent triples. This is described as Functionality $\mathcal{F}_{\mathsf{TSPDZ-PrepMal}}$ below.

Functionality 6: $\mathcal{F}_{\mathsf{TSPDZ-PrepMal}}$

1. **Settling Authentication Keys**: $\mathcal{F}_{\mathsf{TSPDZ-PrepMal}}$ samples a random value Δ. Then $\mathcal{F}_{\mathsf{TSPDZ-PrepMal}}$ samples additive sharings $[\Delta]$ and distributes the shares to all parties.
2. **Assign Random Values to Wires in** C: $\mathcal{F}_{\mathsf{TSPDZ-PrepMal}}$ receives the circuit C from all parties. Then $\mathcal{F}_{\mathsf{TSPDZ-PrepMal}}$ assigns random values to wires in C as follows.
   (a) For each output wire α of an input gate or a multiplication gate, $\mathcal{F}_{\mathsf{TSPDZ-PrepMal}}$ samples a uniform value λ_α and associates it with the wire α. $\mathcal{F}_{\mathsf{TSPDZ-PrepMal}}$ distributes authenticated sharings $[\![\lambda_\alpha]\!] = ([\lambda_\alpha], [\Delta \cdot \lambda_\alpha])$.
   (b) Starting from the first layer of C to the last layer, for each addition gate with input wires α, β and output wire γ, $\mathcal{F}_{\mathsf{TSPDZ-PrepMal}}$ sets $\lambda_\gamma = \lambda_\alpha + \lambda_\beta$. The parties can *locally* compute sharings $[\![\lambda_\gamma]\!]$. In particular, they obtain sharings $[\![\lambda_\alpha]\!]$ for *every* wire α.
3. **Preparing Products with Authentications**: For a multiplication gate with input wires α, β, functionality $\mathcal{F}_{\mathsf{TSPDZ-PrepMal}}$ distributes sharings $[\![\lambda_\alpha \cdot \lambda_\beta]\!] = ([\lambda_\alpha \cdot \lambda_\beta], [\Delta \cdot \lambda_\alpha \cdot \lambda_\beta])$.
4. **Preparing Random Sharings for Input and Output Gates**: For each input or output gate, $\mathcal{F}_{\mathsf{TSPDZ-PrepMal}}$ prepares the following random sharings.
   (a) Let α be the output wire of the input gate, or the input wire of the output gate. $\mathcal{F}_{\mathsf{TSPDZ-PrepMal}}$ samples a uniformly random Δ_α, and distributes sharings $([\Delta_\alpha], [\Delta_\alpha \cdot \lambda_\alpha])$. This allows the input/output holder to verify the correctness of λ_α.

Corrupted Parties: When $\mathcal{F}_{\mathsf{TSPDZ-PrepMal}}$ prepares random sharings, corrupted parties can choose their shares. $\mathcal{F}_{\mathsf{TSPDZ-PrepMal}}$ then samples the random sharings based on the secret it generated and the shares chosen by the corrupted parties.

Online protocol. Protocol $\Pi_{\mathsf{TSPDZ-Online}}$ below described the online phase of the best prior dishonest majority protocol.

Protocol 4: $\Pi_{\mathsf{TSPDZ-Online}}$

The parties compute the circuit in a layer-by-layer manner. The parties call $\mathcal{F}_{\mathsf{TSPDZ-PrepMal}}$ to obtain the following:

- Sharings $[\![\lambda_\alpha]\!] = ([\lambda_\alpha], [\Delta \cdot \lambda_\alpha])$ for every wire α of the circuit;
- For a multiplication gate with input wires α and β, sharings $[\![\lambda_\alpha \cdot \lambda_\beta]\!] = ([\lambda_\alpha \cdot \lambda_\beta], [\Delta \cdot \lambda_\alpha \cdot \lambda_\beta])$.
- For every wire α that is an input wire of an output gate, or output wire of an input gate, sharings $([\Delta_\alpha], [\Delta_\alpha \cdot \lambda_\alpha])$.

Committing Authentication Shares for Output Gates

1. For every wire α that is an input wire of an output gate, all parties use $\mathcal{F}_{\mathsf{Commit}}$ to commit their shares of $([\Delta_\alpha], [\Delta_\alpha \cdot \lambda_\alpha])$ towards the `Client` who owns the gate.

The parties aim at preserving the invariant that, for every wire α in the circuit, the parties know $\mu_\alpha := v_\alpha - \lambda_\alpha$. This is achieved as follows:

Computation of Input Gates

2. Let α be an output wire of an input gate associated to a `Client`. The parties send $([\lambda_\alpha], [\Delta_\alpha], [\Delta_\alpha \cdot \lambda_\alpha])$ to `Client`.
3. `Client` reconstructs λ_α, Δ_α and $\Delta_\alpha \cdot \lambda_\alpha$, and checks that the product of the first two equals the third.
4. If the previous check passes, `Client` computes $\mu_\alpha = v_\alpha - \lambda_\alpha$ and broadcasts μ_α to the parties.

Computation of Addition Gates

5. For an addition gate with input wires α and β, and output wire γ, the parties compute locally $\mu_\gamma = \mu_\alpha + \mu_\beta$.

Computation of Multiplication Gates

6. For a multiplication gate with input wires α and β, and output wire γ, the parties compute locally

$$[\![\mu_\gamma]\!] = \mu_\beta \cdot [\![\lambda_\alpha]\!] + \mu_\alpha \cdot [\![\lambda_\beta]\!] + \mu_\alpha \cdot \mu_\beta + [\![\lambda_\alpha \cdot \lambda_\beta]\!] - [\![\lambda_\gamma]\!].$$

   Notice that $[\![\mu_\gamma]\!] = ([\mu_\gamma], [\Delta \cdot \mu_\gamma])$.
7. The parties reconstruct μ_γ from $[\mu_\gamma]$. They do this by reconstructing this to P_1, who broadcasts the result back to the parties.

Verification of Multiplication Gates

8. Let $\{(\alpha_i, \beta_i, \gamma_i)\}_{i=1}^m$ be the inputs and output wires of the m multiplication gates of the circuit. For each $i \in \{1, \ldots, m\}$, the parties computed $[\![\mu_{\gamma_i}]\!] = ([\mu_{\gamma_i}], [\Delta \cdot \mu_{\gamma_i}])$, and reconstructed μ'_{γ_i}. The parties call $\mathcal{F}_{\mathsf{Coin}}$ for each such i to obtain $\chi_i \in \mathbb{F}$.
9. The parties compute locally $[\sigma] = \sum_{i=1}^m \chi_i \cdot (\mu'_{\gamma_i} \cdot [\Delta] - [\Delta \cdot \mu_{\gamma_i}])$
10. The parties call $\mathcal{F}_{\mathsf{Commit}}$ to commit their shares of $[\sigma]$.
11. All parties open the commitments of the shares of $[\sigma]$ and check whether it is an additive sharing of 0. If not, all parties abort.

Computation of Output Gates
For an output gate with input wire α and owner `Client`:

12. All the parties send μ_α to `Client`. Then all parties open their commitments of shares $\langle \Delta_\alpha \rangle, \langle \Delta_\alpha \cdot \lambda_\alpha \rangle$ to `Client`.
13. `Client` reconstructs λ_α, Δ_α and $\Delta_\alpha \cdot \lambda_\alpha$, and checks that the product of the first two equals the third.
14. `Client` checks that the received μ_α is the same across all parties.
15. If the previous checks pass, `Client` computes $v_\alpha = \mu_\alpha + \lambda_\alpha$ and outputs this value.

In Step 1 of $\Pi_{\mathsf{TSPDZ-Online}}$, we require all parties to commit their shares associated with the output gates due to a subtle security issue in the output phase. If without this step, consider an adversary who knows the inputs of all clients (this is possible as an adversary may have any pre-knowledge about the computation). Then during the output phase, an adversary may learn μ_α (which is not kept secret) and compute λ_α. This allows corrupted parties to locally change their shares of $([\Delta_\alpha], [\Delta_\alpha \cdot \lambda_\alpha])$ before revealing to `Client` in a way such that the modified secrets $\Delta_\alpha + \eta_1$ and $\Delta_\alpha \cdot \lambda_\alpha + \eta_2$ can still pass the verification. Note that this does no harm in the real world.

However, in the ideal world when `Client` is honest, the ideal adversary only knows that the real adversary launched additive attacks on $([\Delta_\alpha], [\Delta_\alpha \cdot \lambda_\alpha])$ but cannot check whether they can still pass the verification in the real world since such a check requires the knowledge of `Client`'s output. Here we address this issue by requiring corrupted parties to commit their shares *before* the computation starts so that the adversary cannot learn μ_α, and thus cannot compute λ_α.

Communication complexity of $\Pi_{\mathsf{TSPDZ-Online}}$. Per multiplication gate, the parties send $n'-1$ shares to P_1, who sends back $n'-1$ values. The total is $2(n'-1) \leq 2n(1-\epsilon)$.

E.2 Circuit-Dependent Preprocessing Phase

Now we show how to instantiate the circuit-dependent preprocessing functionality $\mathcal{F}_{\mathsf{TSPDZ-PrepMal}}$, making use of a circuit-independent preprocessing functionality that samples multiplication triples.

Circuit-independent preprocessing functionality. The functionality for circuit-independent preprocessing is described below as Functionality $\mathcal{F}_{\mathsf{TSPDZ-PrepIndMal}}$.

Functionality 7: $\mathcal{F}_{\mathsf{TSPDZ-PrepIndMal}}$

1. **Setting Authentication Keys:** $\mathcal{F}_{\mathsf{TSPDZ-PrepIndMal}}$ samples a random value Δ. Then $\mathcal{F}_{\mathsf{TSPDZ-PrepIndMal}}$ samples additive sharings $[\Delta]$ and distributes the shares to all parties.
2. **Preparing Random Sharings**: For each output wire α of an input gate or a multiplication gate in the circuit C, $\mathcal{F}_{\mathsf{TSPDZ-PrepIndMal}}$ samples a random value as λ_α, $\mathcal{F}_{\mathsf{TSPDZ-PrepIndMal}}$ samples authenticated shares $[\![\lambda_\alpha]\!] = ([\lambda_\alpha], [\Delta \cdot \lambda_\alpha])$, and distributes the shares to all parties.
3. **Preparing Beaver Triples with Authentications**: For each multiplication gate, $\mathcal{F}_{\mathsf{TSPDZ-PrepIndMal}}$ samples a random triple with authentications as follows:
   (a) $\mathcal{F}_{\mathsf{TSPDZ-PrepIndMal}}$ samples two random values $a, b \in \mathbb{F}$ and computes $\Delta \cdot a, \Delta \cdot b$. Then $\mathcal{F}_{\mathsf{TSPDZ-PrepIndMal}}$ samples two pairs of additive sharings $[\![a]\!] = ([a], [\Delta \cdot a])$, $[\![b]\!] = ([b], [\Delta \cdot b])$.
   (b) $\mathcal{F}_{\mathsf{TSPDZ-PrepIndMal}}$ computes $c = a \cdot b$ and $\Delta \cdot c$. Then $\mathcal{F}_{\mathsf{TSPDZ-PrepIndMal}}$ samples an authenticated sharing $[\![c]\!] = ([c], [\Delta \cdot c])$.

   $\mathcal{F}_{\mathsf{TSPDZ-PrepIndMal}}$ distributes the shares of $([\![a]\!], [\![b]\!], [\![c]\!])$ to all parties.
4. **Preparing Random Sharings for Input and Output Gates**: For each input or output gate with output/input wire α, $\mathcal{F}_{\mathsf{TSPDZ-PrepIndMal}}$ prepares the following random sharings.
   (a) $\mathcal{F}_{\mathsf{TSPDZ-PrepIndMal}}$ samples a and Δ_α uniformly at random.
   (b) $\mathcal{F}_{\mathsf{TSPDZ-PrepIndMal}}$ distributes shares $([\![a]\!], [\Delta_\alpha], [a \cdot \Delta_\alpha])$.

Corrupted Parties: When $\mathcal{F}_{\mathsf{TSPDZ-PrepIndMal}}$ prepares random sharings, corrupted parties can choose their shares. $\mathcal{F}_{\mathsf{TSPDZ-PrepIndMal}}$ then samples the random sharings based on the secret it generated and the shares chosen by the corrupted parties.

Instantiation of the circuit-dependent preprocessing. Using the Functionality $\mathcal{F}_{\mathsf{TSPDZ-PrepIndMal}}$, it is possible to instantiate Functionality $\mathcal{F}_{\mathsf{TSPDZ-PrepMal}}$ using Protocol $\Pi_{\mathsf{TSPDZ-PrepMal}}$ below.

Protocol 5: $\Pi_{\mathsf{TSPDZ-PrepMal}}$

1. **Setting Authentication Keys**: Parties call $\mathcal{F}_{\mathsf{TSPDZ-PrepIndMal}}$ to get additive sharings $[\Delta]$.
2. **Assign Random Values to Wires in C**: Parties call $\mathcal{F}_{\mathsf{TSPDZ-PrepIndMal}}$ to get $[\![\lambda_\alpha]\!] = ([\lambda_\alpha], [\Delta \cdot \lambda_\alpha])$ for every wire α that is the output of either an input gate or a multiplication gate.
3. **Preparing Products with Authentications**: For a multiplication gate with input wires α, β, the parties proceed as follows.
   (a) Parties call $\mathcal{F}_{\mathsf{TSPDZ-PrepIndMal}}$ to obtain a triple $([\![a]\!], [\![b]\!], [\![c]\!])$, where $c = a \cdot b$.
   (b) Parties compute locally $[\![\tau_\alpha]\!] = [\![\lambda_\alpha]\!] - [\![a]\!]$ and $[\![\tau_\beta]\!] = [\![\lambda_\beta]\!] - [\![b]\!]$,
   (c) Parties open τ_α and τ_β by sending the shares of $[\tau_\alpha]$ and $[\tau_\beta]$ to P_1, who reconstructs and broadcasts the results back.
   (d) Parties compute locally
$$[\![\lambda_\alpha \cdot \lambda_\beta]\!] = \tau_\beta \cdot [\![a]\!] + \tau_\alpha \cdot [\![b]\!] + [\![c]\!] + \tau_\alpha \cdot \tau_\beta.$$
4. **Preparing Random Sharings for Input and Output Gates**: For each input or output gate, parties call $\mathcal{F}_{\mathsf{TSPDZ-PrepIndMal}}$ to obtain, for each input or output gate with output/input wire α, sharings $([\![a]\!], [\Delta_\alpha], [a \cdot \Delta_\alpha])$. Then:
   (a) Parties compute locally $[\![\omega_\alpha]\!] = [\![\lambda_\alpha]\!] - [\![a]\!]$ and open this value
   (b) Parties compute locally $[\Delta_\alpha \cdot \lambda_\alpha] = \omega_\alpha \cdot [\Delta_\alpha] + [a \cdot \Delta_\alpha]$.
5. **Verification of Openings**: Let $[\![\xi_1]\!], \dots, [\![\xi_m]\!]$ be the sharings opened in the previouss to phases, and assume they were opened to $\xi'_1, \dots, \xi'_m$. These correspond to the τ_α and τ_β values for multiplication gates, and ω_α for input and output gates.

(a) The parties call $\mathcal{F}_{\mathsf{Coin}}$ to obtain $\chi_1, \ldots, \chi_m \in \mathbb{F}$.
(b) The parties compute locally

$$[\sigma] = \sum_{i=1}^{m} \chi_i \cdot (\xi'_i \cdot [\Delta] - [\Delta \cdot \xi_i]).$$

(c) The parties call $\mathcal{F}_{\mathsf{Commit}}$ to commit their shares of $[\sigma]$.
(d) All parties open the commitments of the shares of $[\sigma]$ and check whether it is an additive sharing of 0. If not, all parties abort.

Communication complexity of $\Pi_{\mathsf{TSPDZ-PrepMal}}$. Per multiplication gate, the parties need to send $2(n'-1)$ shares to P_1, who sends back $2(n'-1)$ elements. This results in $4(n'-1) \leq 4(1-\epsilon)n$.

E.3 Circuit-Independent Preprocessing Phase

Finally, we show how the functionality $\mathcal{F}_{\mathsf{TSPDZ-PrepIndMal}}$ is instantiated. This is essentially a modified version of Le Mans [RS22], where the resulting multiplication triples are fully authenticated, instead of the product being left unauthenticated as in Le Mans. We make use of the functionalities $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ and $\mathcal{F}_{\mathsf{nVOLE}}$, which appear as Functionalities 4 and 5.

Protocol 6: $\Pi_{\mathsf{TSPDZ-PrepIndMal}}$

Setting Authentication Keys

1. Each party P_i inputs `Init` to $\mathcal{F}_{\mathsf{nVOLE}}$ to obtain Δ^i.

Partially Authenticated Triples

2. Each P_i calls $\mathcal{F}_{\mathsf{nVOLE}}$ twice, with input `Extend` and receives the seeds s_a^i, s_b^i. Use the outputs to define vectors of shares $(\llbracket a_\ell \rrbracket)_\ell$, $(\llbracket b_\ell \rrbracket)_\ell$ such that $\boldsymbol{a}^i = \mathtt{Expand}(s_a^i)$ and $\boldsymbol{b}^i = \mathtt{Expand}(s_b^i)$.
3. Every ordered pair (P_i, P_j) calls $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ with P_i sending s_a^i and P_j sending s_b^j. $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ sends back $\boldsymbol{u}^{i,j}$ to P_i and $\boldsymbol{v}^{j,i}$ to P_j such that $\boldsymbol{u}^{i,j} + \boldsymbol{v}^{j,i} = \mathtt{Expand}(s_a^i) * \mathtt{Expand}(s_b^j) = \boldsymbol{a}^i * \boldsymbol{b}^j$. All parties locally compute $[c_\ell]$ where $c_\ell = a_\ell \cdot b_\ell$.

Authenticated Random Values

4. Each P_i calls $\mathcal{F}_{\mathsf{nVOLE}}$ with input `Extend` and receives the seed s_r^i. Use the outputs to define a vector of shares $(\llbracket r_\ell \rrbracket)_\ell$ such that $\boldsymbol{r}^i = \mathtt{Expand}(s_r^i)$.

Authenticating the Products

5. For every ℓ, the parties compute locally $[c_\ell] - [r_\ell]$, and the parties open $(c_\ell - r_\ell)$ by sending their shares to P_1 and this party broadcasting back the reconstruction.
6. The parties compute locally $(c_\ell - r_\ell) \cdot [\Delta] + [\Delta \cdot r_\ell]$. Notice the parties now have $\llbracket c_\ell \rrbracket = ([c_\ell], [\Delta \cdot c_\ell])$.

Verification of the Products (Sacrificing)

7. Parties repeat the previous steps to produce authenticated triples $(\llbracket \tilde{a}_\ell \rrbracket, \llbracket \tilde{b}_\ell \rrbracket, \llbracket \tilde{c}_\ell \rrbracket)_\ell$.
8. The parties invoke $\mathcal{F}_{\mathsf{Coin}}$ to sample $\rho \in \mathbb{F}$.
9. For every ℓ, the parties compute locally $[\tilde{a}_\ell] - \rho \cdot [a_\ell]$ and $[\tilde{b}_\ell] - [b_\ell]$, and the parties open $(\tilde{a}_\ell - \rho \cdot a_\ell)$ and $(\tilde{b}_\ell - b_\ell)$ by sending their shares to P_1 and this party broadcasting back the reconstructions.
10. The parties compute locally for every ℓ:

$$\begin{aligned}\llbracket \theta_\ell \rrbracket = (\tilde{a}_\ell - \rho \cdot a_\ell) \cdot (\tilde{b}_\ell - b_\ell) + (\tilde{a}_\ell - \rho \cdot a_\ell) \cdot \llbracket b_\ell \rrbracket \\ + \rho \cdot (\tilde{b}_\ell - b_\ell) \cdot \llbracket a_\ell \rrbracket + \rho \cdot \llbracket c_\ell \rrbracket - \llbracket \tilde{c}_\ell \rrbracket.\end{aligned}$$

11. The parties call $\mathcal{F}_{\mathsf{Coin}}$ to get $\chi_1, \ldots, \chi_m \in \mathbb{F}$.
12. The parties compute locally $\llbracket \sigma \rrbracket = \sum_{\ell=1}^{m} \chi_\ell \cdot \llbracket \theta_\ell \rrbracket$.
13. The parties call $\mathcal{F}_{\mathsf{Commit}}$ to commit their shares of $[\sigma]$.
14. All parties open the commitments of the shares of $[\sigma]$ and check whether it is an additive sharing of 0. If not, all parties abort.

15. The parties call $\mathcal{F}_{\mathsf{Commit}}$ to commit their shares of $[\Delta \cdot \sigma]$.
16. All parties open the commitments of the shares of $[\Delta \cdot \sigma]$ and check whether it is an additive sharing of 0. If not, all parties abort.

Random Sharings for Input and Output Gates

17. Let $\boldsymbol{\alpha}$ be the list of wires that are output of an input gate, or output of an input gate. The parties sample $[\![a]\!]$ for a random $a \in \mathbb{F}$ as in step 4 above, that is, each P_i calls $\mathcal{F}_{\mathsf{nVOLE}}$ with input `Extend` and receives the seed s_a^i.
18. Each party P_i samples locally a seed $s^i_{\Delta_\alpha}$.
19. Every ordered pair (P_i, P_j) calls $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ with P_i sending s_a^i and P_j sending $s^j_{\Delta_\alpha}$. $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ sends back $\boldsymbol{u}^{i,j}$ to P_i and $\boldsymbol{v}^{j,i}$ to P_j such that $\boldsymbol{u}^{i,j} + \boldsymbol{v}^{j,i} = \mathtt{Expand}(s_a^i) * \mathtt{Expand}(s^j_{\Delta_\alpha}) = \boldsymbol{a}^i * \boldsymbol{\Delta}^j_{\boldsymbol{\alpha}}$. All parties locally compute $[a_\ell \cdot \Delta_{\alpha_\ell}]$ for each ℓ.

Communication complexity of $\Pi_{\mathsf{TSPDZ-PrepIndMal}}$. Ignoring the calls to $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ and $\mathcal{F}_{\mathsf{nVOLE}}$, the amount of communication per multiplication gate involves the parties sending $n' - 1$ shares to P_1 who sends $n' - 1$ values back, when authenticating the products. This amounts to $2(n' - 1) \leq 2(1 - \epsilon)n$ elements. In the sacrifice stpe, the parties must open two shared values, which increases this quantity to $6(1 - \epsilon)n$. We ignore the rest of the the cost of the sacrifice step since its communication is independent of the amount of multiplications.

Number of calls to $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ and $\mathcal{F}_{\mathsf{nVOLE}}$. Protocol $\Pi_{\mathsf{TSPDZ-PrepIndMal}}$ requires the `Expand` function in the definition of Functionality $\mathcal{F}_{\mathsf{nVOLE}}$ (and $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$) to produce $|C|$ correlations, where $|C|$ is the number of multiplication gates of the circuit C. Each of these correlations is used to process a different multiplication gate. In our protocol SuperPack, we only require $|C|/k$ total correlations.

F Additional Discussion on the Communication Complexity

Let n be the number of parties Let $0 < \epsilon < 1/2$, and t be the amount of corrupt parties. Assume that $t + 1 = n(1 - \epsilon)$, that is, the percentage of corrupt parties is roughly $(1 - \epsilon) \times 100\%$, and the percentage of honest parties is roughly $\epsilon \times 100\%$.

Recall that both our protocol and the best prior work (see Section E) are split into three phases:

1. (Section 4) An online phase that is dependent on the client's inputs and the circuit topology;
2. (Section 5) A circuit-dependent offline phase that is independent of the client's inputs, but dependent on the topology of the circuit;
3. (Section 6) A circuit-independent offline phase that is independent of both the client's inputs and the topology of the circuit, and only depends on (an upper bound on) the amount of gates of each type.

We present for reference, in Table 4, a more expanded version of Table 1.

The online communication of our protocol is $6/\epsilon$, while the communication complexity of the online phase in Turbospeedz is $2(1 - \epsilon)n$, which depends linearly on n. Therefore, our online phase improves on that of Turbospeedz asymptotically, as n increases. Let us denote by ℓ the improvement factor of our protocol with respect to Turbospeedz, that is, $1/\ell = 3/(n\epsilon(1 - \epsilon))$, which means that the online communication of our protocol is $\ell\times$ better than that of Turbospeedz. Not surprisingly, this factor improves as n or ϵ increase, which is consistent with the fact that either increasing the ratio of honest parties among a fixed set of parties, or increasing the total amount of parties for a fixed honest ratio, benefits our protocol. Regarding the complete offline phase (ignoring calls to $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ and $\mathcal{F}_{\mathsf{nVOLE}}$), our complexity is $6n + 39/\epsilon - 11$, while in Turbospeedz it is $10(1 - \epsilon)$. Hence, the ratio is $10(1 - \epsilon)n/(6n + 39/\epsilon)$. This approaches $10(1 - \epsilon)/6$ as $n \to \infty$, which is a limit that ranges between $10/6 \approx 1.6$ for $\epsilon = 0$, to $5/6 \approx 0.83$ for $\epsilon = 1/2$. As a result, *in the limit* as the number of parties grows, our offline phase is only $1/0.83 = 1.2\times$ less efficient than that of Turbospeedz if ϵ is too small, and it can even be $1.6\times$ better than that of Turbospeedz if ϵ is close to $1/2$. Some concrete improvement factors for both the online and offline phases are computed in Table 5, for different values of ϵ and n.

		Online	CD Offline	CI Offline
Three phases	**SuperPack**	$6/\epsilon$	$4/\epsilon$	$6n + 35/\epsilon$
	Turbospeedz	$2(1-\epsilon)n$	$4(1-\epsilon)n$	$6(1-\epsilon)n$
Two phases	**SuperPack**	$6/\epsilon$	$6n + 39/\epsilon$	
	Turbospeedz	$2(1-\epsilon)n$	$10(1-\epsilon)n$	
End-to-end	**SuperPack**	$6n + 45/\epsilon$		
	Turbospeedz	$12(1-\epsilon)n$		

Table 4. Communication complexity in terms of field elements per multiplication gate of SuperPack, and comparison to the previous work with the best online phase, which is Turbospeedz [BNO19] (with its offline phase instantiated by Le Mans [RS22]). For reference we also add up the two offline phases and report one single offline phase, and add the three phases and report end-to-end complexity. The cost of the calls to $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ and $\mathcal{F}_{\mathsf{nVOLE}}$ in the circuit-dependent offline phase is ignored.

# Parties	Percentage of corrupt parties			
	90%	80%	70%	60%
25	0.75 / 0.42	1.33 / 0.58	1.75 / 0.62	2.0 / 0.61
50	1.5 / 0.65	2.67 / 0.81	3.5 / 0.81	4.0 / 0.75
75	2.25 / 0.8	4.0 / 0.93	5.25 / 0.91	6.0 / 0.82
100	3.0 / 0.91	5.33 / 1.01	7.0 / 0.96	8.0 / 0.86

Table 5. Improvement factor that represents how many times more (or less) efficient our protocol is with respect to Turbospeedz. This is presented as ℓ_{Onl} / ℓ_{Off}, where $\ell_{\mathsf{Onl}} = (n\epsilon(1-\epsilon))/3$ is the factor in the online phase, and $\ell_{\mathsf{Off}} = (10(1-\epsilon)n)/(6n + 39/\epsilon)$ is the factor in the offline phase.

Finally, we consider the question of, given a desired improvement factor ℓ, how many honest parties are needed in order for our protocol to achieve the desired relative efficiency? This is addressed in the following proposition, which shows that, if there are at least 4ℓ honest parties, our protocol outperforms Turbospeedz by a factor of ℓ for *any* total number of parties n, as long as $n \geq 16\ell$. This is important as it shows that the condition on whether our protocol is better than Turbospeedz and by how much depends essentially on the concrete amount of honest parties only, independently on the total amount of parties.

Proposition 1. *Given $\ell \in \mathbb{N}$, the online phase of our protocol is at least $\ell\times$ better than that of Turbospeedz as long as there are at least 6ℓ honest parties and a total amount of parties $n \geq 12\ell$, or at least 4ℓ honest parties and a total amount of parties $n \geq 16\ell$.*

Proof. Assuming $12\ell \leq n$, simple algebra shows that our improvement factor is at least ℓ if

$$\epsilon \geq \frac{1}{2} \cdot \left(1 - \sqrt{1 - \frac{12\ell}{n}}\right).$$

In terms of the amount of honest parties, we see that

$$h = \epsilon \cdot n \geq \ell \cdot \left(\frac{6}{1 + \sqrt{1 - \frac{12\ell}{n}}}\right)$$

honest parties are required for our online phase to be $\ell\times$ that of Turbospeedz. Since $6/(1 + \sqrt{1 - \frac{12\ell}{n}}) \leq 6$, we have that $h = 6 \cdot \ell$ honest parties suffice for our online protocol to be $\ell\times$ better than that of Turbospeedz, *for any number of parties* $n \geq 12\ell$. Furthermore, if $n \geq 16\ell$, then $12\ell/n \geq 3/4$, and in this case we can check that $6/(1 + \sqrt{1 - \frac{12\ell}{n}}) \leq 4$, so $h = 4\ell$ parties suffice. $\square$

F.1 Supplementary Data for Table 3

Table 3 shows the improvement factor of our online protocol with respect to the online phase in Turbospeedz, for a varying number of parties, ϵ and network bandwidth. In Table 6, we provide the running time of our online protocol (in seconds) with the same parameter settings.

Bandwidth	**# Parties**	**Percentage of corrupt parties**			
		90%	80%	70%	60%
500 mbps	**16**	1.22	1.18	1.12	0.74
	32	1.88	1.35	1.08	1.03
	48	2.60	1.62	1.16	0.96
	64	2.74	1.87	1.21	1.16
	80	2.63	1.71	1.14	1.15
100 mbps	**16**	2.17	1.67	1.71	1.20
	32	3.00	2.38	1.81	1.57
	48	4.51	2.48	1.89	1.40
	64	4.60	2.80	2.03	1.70
	80	4.74	2.73	2.00	1.64
50 mbps	**16**	3.84	2.70	2.65	1.98
	32	5.41	4.00	2.75	2.48
	48	8.28	4.18	3.05	2.28
	64	8.56	4.80	3.17	2.77
	80	8.60	4.65	3.50	2.82
10 mbps	**16**	18.91	12.57	12.57	9.34
	32	27.00	19.70	13.12	11.19
	48	39.87	20.00	14.94	10.82
	64	41.43	23.83	15.27	11.54
	80	41.76	23.22	16.28	12.59

Table 6. Running time (in seconds) of our online protocol for a varying number of parties, ϵ and network bandwidth. The network delay is 1ms for the simulation of LAN network. The circuits have depth 10 and width 10k.

G Comparison with Our Semi-honest Variant

In this section, we compare the cost of our protocol with the semi-honest variant of our protocol.

Communication Complexity of the Semi-honest Variant of Our Protocol. When focusing on semi-honest security, we do not need to authenticate the packed Shamir secret sharings by using MACs in our construction. For simplicity, we focus on the communication complexity per multiplication gate:

– In Π_{Online}, the communication complexity of π_{Mult} remains to be $\frac{6}{\epsilon}$ elements per multiplication gate (Step 2 and Step 6 in π_{Mult}).
– In Π_{PrepMal}, the communication complexity per multiplication gate remains to be $\frac{4}{\epsilon}$ elements (Step 4.(a) in Π_{PrepMal}).
– In $\Pi_{\mathsf{PrepIndMal}}$, we may save the communication complexity from the following aspects.
  • In Step 2, we may use π_{RandSh} rather than π_{RandAuth} to prepare random sharings in the form of $[r \cdot \mathbf{1}]_{n-k}$.
  • In Step 3(a), we may omit Step 4 in π_{Triple}.
  • We may omit Step 3(b) entirely.

  As a result, the communication cost is reduced to $2n + 17/\epsilon$ elements per multiplication gate.

As we can see, the communication complexity of our protocol is the same as that of our semi-honest variant in the online phase and circuit-dependent preprocessing phase. Concretely,

– For the online phase, both protocols require $\frac{6}{\epsilon}$ elements per multiplication gate.
– For the circuit-dependent preprocessing phase, both protocols require $\frac{4}{\epsilon}$ elements per multiplication gate.

For the circuit-independent preprocessing phase, our malicious protocol requires $6n+\frac{35}{\epsilon}$ elements per multiplication gate. And our semi-honest variant requires $2n+\frac{17}{\epsilon}$ elements per multiplication gate. Thus, we conclude that, the *additive* cost to upgrade from semi-honest security to malicious security is $4n+\frac{18}{\epsilon}$ elements per multiplication gate, in the circuit-independent preprocessing phase.

Protocols, Procedures and Functionalities

Figure 1 lists our procedures (blue boxes), protocols (grey boxes) and functionalities (yellow boxes), together with their dependencies and positions in the text. A procedure is like a protocol, except that it does not implement a functionality but rather is used as a building block inside another protocol.

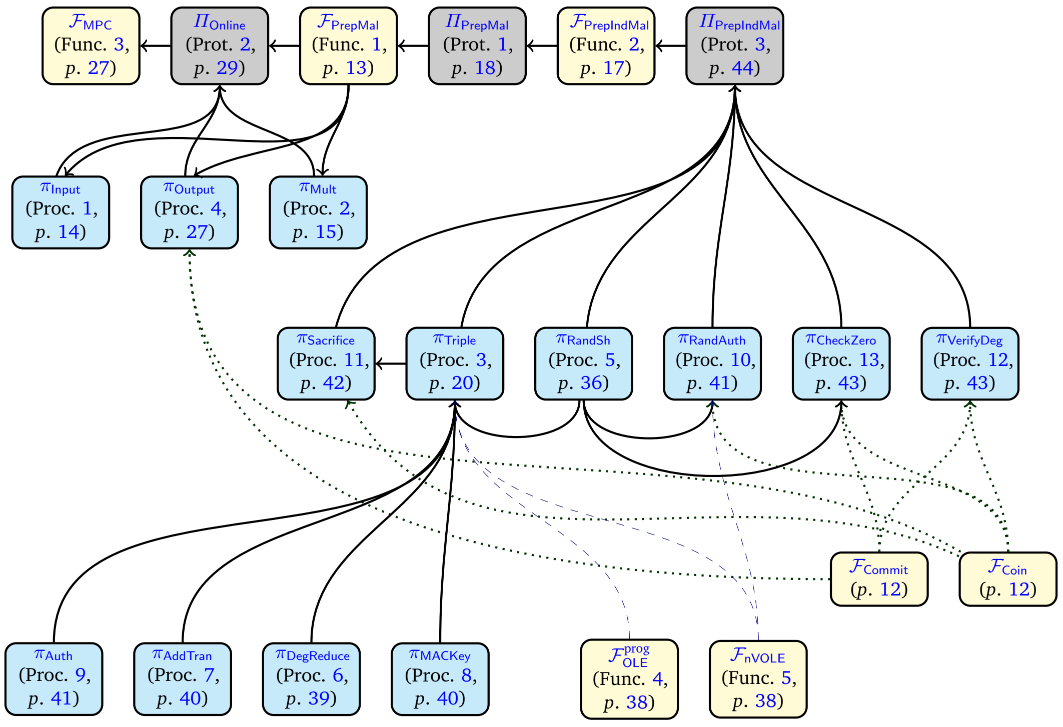
$\mathcal{F}_{\mathsf{MPC}}$ (Func. 3, p. 27)
Π_{Online} (Prot. 2, p. 29)
$\mathcal{F}_{\mathsf{PrepMal}}$ (Func. 1, p. 13)
Π_{PrepMal} (Prot. 1, p. 18)
$\mathcal{F}_{\mathsf{PrepIndMal}}$ (Func. 2, p. 17)
$\Pi_{\mathsf{PrepIndMal}}$ (Prot. 3, p. 44)
π_{Input} (Proc. 1, p. 14)
π_{Output} (Proc. 4, p. 27)
π_{Mult} (Proc. 2, p. 15)
$\pi_{\mathsf{Sacrifice}}$ (Proc. 11, p. 42)
π_{Triple} (Proc. 3, p. 20)
π_{RandSh} (Proc. 5, p. 36)
π_{RandAuth} (Proc. 10, p. 41)
$\pi_{\mathsf{CheckZero}}$ (Proc. 13, p. 43)
$\pi_{\mathsf{VerifyDeg}}$ (Proc. 12, p. 43)
$\mathcal{F}_{\mathsf{Commit}}$ (p. 12)
$\mathcal{F}_{\mathsf{Coin}}$ (p. 12)
π_{Auth} (Proc. 9, p. 41)
π_{AddTran} (Proc. 7, p. 40)
$\pi_{\mathsf{DegReduce}}$ (Proc. 6, p. 39)
π_{MACKey} (Proc. 8, p. 40)
$\mathcal{F}^{\mathsf{prog}}_{\mathsf{OLE}}$ (Func. 4, p. 38)
$\mathcal{F}_{\mathsf{nVOLE}}$ (Func. 5, p. 38)

Fig. 1. Dependency graph of our procedures, protocols and functionalities. An arrow from A to B means that A is used to implement B. Dashed lines correspond to constructions using the $\mathcal{F}^{\mathsf{prog}}_{\mathsf{OLE}}$ and $\mathcal{F}_{\mathsf{nVOLE}}$ functionalities, which we do not instantiate in our work, and dotted lines correspond to constructions using $\mathcal{F}_{\mathsf{Commit}}$ and $\mathcal{F}_{\mathsf{Coin}}$, whose instantiation is simple and is discussed in Section 3.