# TurboPack: Honest Majority MPC with Constant Online Communication

Daniel Escudero,[1] Vipul Goyal,[2,3] Antigoni Polychroniadou[1] and Yifan Song[2]

[1] J.P. Morgan AI Research, NY, USA
[2] Carnegie Mellon University
[3] NTT Research

**Abstract.** We present a novel approach to honest majority secure multiparty computation in the preprocessing model with information theoretic security that achieves the best online communication complexity. The online phase of our protocol requires $12$ elements in total per multiplication gate with circuit-dependent preprocessing, or $20$ elements in total with circuit-independent preprocessing. Prior works achieved linear online communication complexity in $n$, the number of parties, with the best prior existing solution involving $1.5n$ elements per multiplication gate. Only one recent work (Goyal *et al*, CRYPTO'22) achieves constant online communication complexity, but the constants are large ($108$ elements for passive security, and twice that for active security). That said, our protocol offers a very efficient information theoretic online phase for any number of parties.
The total end-to-end communication cost with the preprocessing phase is linear in $n$, i.e., $10n + 44$, which is larger than the $4n$ complexity of the state-of-the-art protocols. The gap is not significant when the online phase must be optimized as a priority and a reasonably large number of parties is involved. Unlike previous works based on packed secret-sharing to reduce communication complexity, we further reduce the communication by avoiding the use of complex and expensive network routing or permutations tools. Furthermore, we also allow for a maximal honest majority adversary, while most previous works require the set of honest parties to be strictly larger than a majority.
Our protocol is simple and offers concrete efficiency. To illustrate this we present a full-fledged implementation together with experimental results that show improvements in online phase runtimes that go up to $5\times$ in certain settings (*e.g.* $45$ parties, LAN network, circuit of depth $10$ with 1M gates).

## 1 Introduction

Secure multiparty computation (MPC) enables a set of parties, each having its own input, to compute a given function on these without leaking anything besides the output while only involving communication among each other, *i.e.* without relying on a central third party. Security requires that, even if an adversary corrupts $t$ out of the $n$ parties, this adversary learns nothing about the inputs of non-corrupt/honest parties. Different protocols exist depending on the ratio $t/n$. If $t < n/2$, which is referred to as the honest majority setting, it is known that information-theoretic security is possible to achieve, whereby the adversary does not learn anything about the honest parties' inputs regardless of his/her computational power. In contrast, if $t \geq n/2$, which is known as the dishonest majority setting, computational assumptions are required to achieve security.

The focus of this work is the honest majority case, where the adversary corrupts at most a minority of the parties, and information-theoretic security can be achieved. A crucial metric for the performance of an MPC protocol is its communication complexity, meaning the amount of messages that must be sent across the parties. Recent constructions such as [DN07, GIP+14, CGH+18, NV18, GSZ20, BGIN20, GLO+21] show that an arithmetic circuit $C$ can be evaluated with overall communication complexity $O(|C| \cdot n)$ elements, where $|C|$ is the number of multiplication gates in the circuit. Furthermore, some of these works achieve very small concrete constants: [GSZ20] and [GLO+21] require $5.5n$ and $4n$ elements per multiplication gate, respectively. Moreover, different works suggest that we cannot design MPC protocols with $o(|C| \cdot n)$ communication [DNPR16, DLN19].

With the aim of further improving the communication complexity of honest majority MPC protocols, we consider the widely used offline/online paradigm, in which the execution of the protocol is split into two phases: an *offline phase*, which is independent of the parties' inputs and hence can be run in advance before these are known, and an *online phase*, which requires knowledge of the inputs, and is typically much more lightweight than the offline phase. For example, in the dishonest majority setting it is common to use the offline phase to preprocess the so-called authenticated Beaver triples, which makes use of computational assumptions and expensive cryptographic tools, and are then consumed in an online phase that is highly efficient and information-theoretically secure. This is the trend followed in BeDOZa [BDOZ11], SPDZ [DPSZ12a, DKL+13], and all the subsequent works in secret-sharing-based dishonest majority MPC. The idea behind this approach is that, even though the end-to-end protocol may not provide certain level of efficiency, since the online phase dictates the latency from the moment in which the parties provide inputs to the moment they receive output, having a fast online phase may be enough for a wide range of applications, especially if the parties can afford to pre-compute the offline phase (*e.g.* while they are idle).

State-of-the-art protocols such as ATLAS [GLO+21] aim at minimizing total communication complexity, achieving $4n$ elements per multiplication gate, distributed as $2n$ in the offline phase, and $2n$ in the online phase. However, it is natural to wonder how much can the *online phase* alone be optimized, while still achieving comparable overall efficiency as state-of-the-art work. In this case, the protocol that offers the most lightweight online phase, *asymptotically*, is the recent work of [GPS22]. This protocol focuses in the more general dishonest majority setting with $t < (1-\epsilon)n$ for some $\epsilon > 0$, and it achieves an online phase whose communication complexity is $O(|C|)$ (*i.e.* it is independent of the number of parties). Setting $\epsilon = 1/2$, we obtain the honest majority setting as a particular case. However, one drawback of the construction from [GPS22] is the large constant hidden in the big-$O$ notation:

- For the online phase, the analysis of [GPS22] shows that their protocol requires $14 \cdot n/k$ elements of communication per multiplication and addition gate,[4] where, for the honest majority case, $k \approx n/4$ and $t \leq n - 2k + 1$. Thus, the cost per addition and multiplication gate is about $56$ elements. Assuming the number of addition gates is the same as the number of multiplication gates, the effective cost per multiplication gate becomes $108$ elements.
- For the preprocessing phase, the analysis of [GPS22] shows that their protocol requires $12 \cdot n^2/k^2$ elements of preprocessing data, where $k \approx n/4$ in the honest majority setting. The circuit size of computing the preprocessing data would be $192|C|$. When directly instantiating it by an IT MPC in the standard honest majority setting, it costs at least $192|C| \cdot n$ elements of communication.
- To achieve the malicious security, the work [GPS22] uses information-theoreitc MACs, which at least double the cost in both the preprocessing phase and the online communication.

In contrast, the protocol with the next best online phase is DN07 [DN07] with the optimization from [GSZ20], which requires $1.5n$ elements per multiplication gate in the online phase, but $4n$ elements in the offline phase. Unless $n > 72$ for passive security, or $n > 144$ for active security, [GPS22] does not necessarily offer a better online phase.

## 1.1 Our Contribution

In this work, we present a new honest majority MPC protocol, TurboPack, which has an online phase whose total communication per multiplication gate is *constant*, irrespectively of the number of parties, and furthermore, unlike [GPS22], this constant is concretely small. To better understand the communication complexity of TurboPack, we split the computation into three phases:[5]

① A circuit and input-independent phase, requiring $10n + 24$ elements per multiplication gate
② A circuit-dependent but input-independent phase, requiring $8$ elements per multiplication gate
③ A circuit and input-dependent phase, requiring $12$ elements per multiplication gate.

More concrete theorem statements can be found in Thm. 1 and 2 for passive security, and 3 for active security. In the circuit-dependent preprocessing model (**CD**), the offline phase is allowed to depend on the circuit to be computed, and in this case our online phase (③) requires only $12$ elements per multiplication gate. In the circuit-independent preprocessing model (**CI**), where the offline phase cannot depend on the circuit, our online phase (②+③) is only slightly larger, namely $20$

[4] Most protocols, including TurboPack, do not require communication for addition gates. The protocol of [GPS22] is an exception.

[5] We note that such a splitting of three phases has been considered in the line of works focusing on constant-round MPC with malicious security such as [NST17, WRK17b, WRK17a, LPSY19].

field elements per multiplication gate. Either way, it only takes a small value of $n$ for our online phase to become more efficient than that of DN07, and the more parties involved, the better TurboPack becomes. Notice that the CD prep. model makes sense in many practical settings where the circuit is already known in advance (e.g. perhaps it is fixed for certain task, like training a neural network of a pre-determined architecture on data provided by the clients), and recent works have shown some benefits of knowing the circuit in advance for the preprocessing [BENO19, ESV21, ACE$^+$21]. Nonetheless, the CI prep. model also has its advantages, such as allowing external parties or services to produce preprocessing data "as a service", while being agnostic to which computation is being carried out.

While we achieve the best possible online phase, our overall costs are higher than ATLAS, or even DN07. However, this gap is not very large: the total communication of TurboPack is $10n + 44$ elements per multiplication gate, a factor of $2.5\times$ worse than ATLAS (the protocol with the best overall communication) and $\approx 1.8\times$ worse than DN07 (the protocol with the best online communication). For many application settings, this overhead can be considered to be small taking into account the gains in the online phase.

We also remark that we consider both the passive and active security (with abort) settings, but for most of the main body of this paper we focus on passive security. Our results for active security are obtained by making use of the passively secure construction as a starting point, and using existing techniques in distributed zero-knowledge proofs [BBCG$^+$19] in a black-box manner to verify the correctness of the computation. The overhead in terms of communication can be made negligible, so the communication complexity of our actively secure protocol remains essentially the same as that of the passive protocol.

We achieve constant communication in the online phase with the help of packed secret-sharing, a technique to distribute and operate on multiple secrets simultaneously while only paying the cost of a single secret (we refer the reader to Section 2 where we provide a detailed overview of TurboPack). This tool has been used in several previous works [FY92, DIK10, GIP15, GIOZ17, BGJK21a, GPS21], however, these works typically (1) only tolerate a smaller corruption threshold $t < (1/2 - \epsilon) \cdot n$, and (2) require complex network routing or permutation-based techniques to make packed secret-sharing, which is more suitable for SIMD computations, compatible with less structured circuits. Unlike the works mentioned above, TurboPack tolerates the optimal honest majority adversary $n = 2t+1$, and it is concretely simple and efficient, avoiding complex network routing or permutation tools. The only exception is the recent work of [GPS22], which uses packed secret-sharing in the dishonest majority setting, which includes honest majority as a particular case. However, as we have mentioned already, TurboPack is simpler and achieves much better concrete constants. A more detailed description and comparison to this and several other relevant related works appears in Section 1.2.

We have fully implemented the passive version of TurboPack, and we carried out a series of experiments that assess the improvement of our online phase with respect to DN07, the protocol with the most efficient online phase for a reasonable number of parties. We experimentally observe that, over a LAN network and in the CD prep. model, our online phase starts outperforming that of DN07 for small values of $n$ such as $n = 13$, and for larger values of $n$, such as $n = 45$, and for circuits of moderate width (100k), our online phase takes only 22% the time than that of DN07. In the CI prep. model our improvement is slightly smaller but still noticeable: for $n = 45$ and 100k width, our (circuit-independent) online phase is almost twice as fast as that of DN07. Other scenarios lead to even better improvement factors of $6.7\times$, such as the localhost setting with at least 69-77 parties, as evaluated in Table 5 in Section F in the Appendix. From the above, we regard TurboPack as an important step towards achieving practical honest majority MPC for any number of parties.

## 1.2 Related Work

*Honest-majority with maximal adversary.* There is a long line of works studying the efficiency of honest majority information-theoretic MPC in our settings of interest: passive security and active security with abort, for an arbitrary number of parties (hence we do not consider protocols restricted to small number of parties such as [BGIN19]). The first protocol in achieving linear communication complexity in this setting is DN07 [DN07], whose concrete constants were improved in [GSZ20], achieving, per multiplication gate, $4n$ elements in the offline phase and $1.5n$ elements in the online phase. This is both for passive security and active security with abort, ignoring logarithmic terms in $|C|$ for the latter case. The protocol of [CGH$^+$18] showed that active security (with abort) could be obtained with an overhead of only $2\times$ with respect to passive security. In [DE21a], where the authors aim at minimizing the online phase costs for general secret-sharing schemes, but when instantiated with Shamir secret-sharing the same online complexity of $1.5n$ as above is obtained. This can be brought down to $1n$ field elements by using MACs, as shown in [DE21a], at the expense of increasing the communication complexity of the offline phase.

The protocol with the best overall communication complexity is ATLAS [GLO$^+$21], which uses $4n$ elements in total per multiplication gate, distributed as $2n$ elements in the offline phase and $2n$ elements in the online phase. As we have already pointed out, achieving $o(|C|n)$ communication complexity for the maximal adversary $n = 2t + 1$ is believed to be impossible. However, none of the works above have achieved $o(|C|n)$ even for the online phase of the protocol only. In our work, we achieve an online phase with $O(|C|)$ communication complexity, with actual small hidden constants. The overall communication complexity is linear in $n$, and it is only a factor of $< 2\times$ worse than state-of-the-art.

Finally, the work of [GPS22] considers packed secret-sharing in the context of dishonest majority MPC, which in particular includes the case $n = 2t + 1$. However, as we explained in Section 1, their work requires larger constants and only becomes practical for a very large number of parties.

*Honest-majority with sub-maximal adversary.* When the corruption threshold satisfies $t < (1/2-\epsilon)\cdot n$ for some $\epsilon > 0$, multiple works have made use of packed secret-sharing techniques to achieve a *total* communication complexity that is independent of the number of parties [FY92, DIK10, GIP15, GIOZ17, BGJK21a, GSY21, BGJK21b, GPS21]. Since packed secret-sharing is intended to operate on vectors, performing the same operation at an entry-wise level, different techniques are needed to accomodate for the fact that, for typical circuits, values must be re-ordered from one batch to the next.

Different works tackle this difficulty in different ways. A prominent method is to preprocess different permutation pairs that are used in the online phase to re-arrange secret-shared data. This is the approach followed by the pioneering work of [DIK10]. Other works, such as [GSY21] and [BGJK21b], use packed secret-sharing for circuits that are more “SIMD-friendly” ([GSY21] uses it to preprocess multiplication triples, [BGJK21b] uses it on circuits having wide-enough blocks that appear with high enough frequency). Most of the works mentioned above achieve a communication complexity per multiplication gate of $O(\log|C|)$. The work [GPS21] first shows that an arithmetic circuit $C$ can be evaluated with overall communication complexity $O(1)$ elements per multiplication gate, but again, this is still in the $t < (1/2 - \epsilon) \cdot n$ setting. Furthermore, this requires complex techniques related to network-routing in order to support general non-SIMD circuits, and the hidden constants in the big-$O$ notation are relatively large.

Our work is not comparable to these related works, given that we tolerate the maximal adversary $n = 2t + 1$, while they require a gap between $t$ and $(n - 1)/2$. With this gap, it is possible to obtain $O(|C|)$ communication complexity overall [GPS21], which we cannot achieve in our setting. However, it is important to mention that the hidden constants in works like [GPS21], coupled with the complexities of impementing network routing, may make of TurboPack a better option in practice, especially if $2t + 1$ is close to $n$. For example, if $t = (n - 1)/2 + 1 - k$, the protocol in [GPS21] can be used with a “packing parameter” (the amount of secrets packed in one share) of $k$. TurboPack, in contrast, has a packing parameter of $\approx n/4$. As a result, unless $k > n/4$, our packing parameter—and hence our online phase—is better. Furthermore, asymptotically (in $n$) the overall communication in [GPS21] may be better than ours, but the hidden constants in big-$O$ notation make it so that such improvement is only noticeable for very large values of $n$.

## 2 Technical Overview

In this section, we give an overview of our techniques. In the following, we will use a bold letter to represent a vector.

### 2.1 Starting Idea: Efficient Online Protocol Based on [BBG$^+$21, GPS22]

When the corruption threshold is sub-optimal (either with honest majority or dishonest majority), the generic approach of reducing the communication complexity is to use the packed secret sharing technique introduced in [FY92]. Let $k$ denote the packing parameter, which looking ahead, will be set equal to $k = (n - t + 1)/2$. The idea of the packed secret sharing technique is to store $k$ secrets within a single secret sharing. In this way, we can evaluate a group of $k$ (addition or multiplication) gates in parallel. In particular, the cost of evaluating a group of $k$ gates by using packed secret sharings is the same as the cost of evaluating a single gate by using standard secret sharings. Ideally, the overall communication complexity can be reduced by a factor of $k$.

However, the main issue of the above approach is to prepare input packed sharings for each layer. The issue is due to the following two facts:

1. The secrets within a single packed secret sharing may not be in the correct order. When evaluating a group of $k$ gates, the protocols only support coordinate-wise operations, which requires the two vectors of secrets to be correctly aligned.
2. The secrets within a single packed secret sharing may be scattered in different output packed secret sharings in previous layers.

This issue is referred to as network routing in [GPS21]. The reason of why it cannot be easily solved is because we need to collect the secrets and reorder them in a batch way. Doing it secret by secret will cost $O(k)$ communication per packed secret sharing, which eliminates the benefit of using the packed secret sharing technique.

We note that the solutions in [GPS21, GPS22] are costly in term of the constant. What makes it worse is that, even for a group of $k$ addition gates, all parties still need to prepare input packed secret sharings for these $k$ gates. This is different from the IT MPC protocols in the standard honest majority setting [DN07, GIP+14, CGH+18, NV18, GSZ20, BGIN20, GLO+21], where addition gates can be evaluated without interaction.

*Efficient Online Protocol in [BBG+21].* An exception is the online protocol in [BBG+21], which relies on a circuit-dependent preprocessing phase (i.e., the correlated randomness can depend on the circuit but not parties' inputs) to avoid paying cost for addition gates and doing network routing in the online phase. Despite that the work of [BBG+21] focuses on the sub-optimal corruption threshold, we try to utilize their online protocol in our setting.

In the circuit-dependent preprocessing phase, a random value $\lambda_\alpha$ is assigned to each wire $\alpha$ in the circuit such that:

- For each output wire $\alpha$ of input gates[6] and multiplication gates, $\lambda_\alpha$ is uniformly random.
- For each addition gate with input wires $\alpha, \beta$ and output wire $\gamma$, $\lambda_\gamma = \lambda_\alpha + \lambda_\beta$.

Multiplication gates in each layer of the circuit are divided into groups of size $k$. For each group of $k$ multiplication gates, the random values associated with the first input wires and the second input wires are shared by using packed secret sharings respectively.

In the online phase, if we use $v_\alpha$ to represent the actual wire value associated with the wire $\alpha$, the goal is to compute $\mu_\alpha = v_\alpha - \lambda_\alpha$. The authors in [BBG+21] observe that it is sufficient to only let a single party, say $P_1$, learn $\{\mu_\alpha\}_\alpha$. It comes with two benefits:

- For each addition gate with input wires $\alpha, \beta$ and output wire $\gamma$, since $v_\gamma = v_\alpha + v_\beta$ and $\lambda_\gamma = \lambda_\alpha + \lambda_\beta$, $P_1$ can locally compute $\mu_\gamma = \mu_\alpha + \mu_\beta$. Therefore, addition gates can be computed without interaction.
- For each group of $k$ multiplication gates, let $\boldsymbol{\alpha}$ denote the batch of the first input wires, and $\boldsymbol{\beta}$ denote the batch of the second input wires. Recall that $P_1$ learns $\{\mu_\alpha\}_\alpha$ in clear. $P_1$ shares $\boldsymbol{\mu_\alpha}$ and $\boldsymbol{\mu_\beta}$ (which are two vectors of values) to all other parties by using packed secret sharings. Since all parties hold packed secret sharings of $\boldsymbol{\lambda_\alpha}$ and $\boldsymbol{\lambda_\beta}$ prepared in the circuit-dependent preprocessing phase, they can locally compute packed secret sharings of $\boldsymbol{v_\alpha}$ and $\boldsymbol{v_\beta}$ without doing network routing in the online phase.

Thus, the main task is to evaluate a group of multiplication gates and compute $\boldsymbol{\mu_\gamma}$ for the output wires $\boldsymbol{\gamma}$. To this end, we first review the packed Shamir secret sharing scheme, which is used in our construction.

*Review: Packed Shamir Secret Sharing Scheme and Multiplication-Friendliness.* The packed Shamir secret sharing scheme [FY92] is a natural generalization of the standard Shamir secret sharing scheme [Sha79]. It allows to secret-share a batch of secrets within a single Shamir sharing. For a vector $\boldsymbol{x} \in \mathbb{F}^k$, we use $[\![\boldsymbol{x}]\!]_d$ to denote a degree-$d$ packed Shamir sharing, where $k-1 \leq d \leq n-1$. It requires $d+1$ shares to reconstruct the whole sharing, and any $d-k+1$ shares are independent of the secrets. The packed Shamir secret sharing scheme has the following nice properties:

[6] Output wires of input gates are just the input wires of the circuit.

– Linear Homomorphism: For all $d \geq k-1$ and $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}^k$, $[\![\boldsymbol{x}+\boldsymbol{y}]\!]_d = [\![\boldsymbol{x}]\!]_d + [\![\boldsymbol{y}]\!]_d$.
– Multiplicative: For all $d_1, d_2 \geq k-1$ subject to $d_1 + d_2 < n$, and for all $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}^k$, $[\![\boldsymbol{x} * \boldsymbol{y}]\!]_{d_1+d_2} = [\![\boldsymbol{x}]\!]_{d_1} * [\![\boldsymbol{y}]\!]_{d_2}$, where the multiplications are performed on the corresponding shares.

As noted in [GPS22], when $d \leq n-k$, all parties can locally multiply a public vector $\boldsymbol{c} \in \mathbb{F}^k$ with a degree-$d$ packed Shamir sharing $[\![\boldsymbol{x}]\!]_d$:

1. All parties first locally compute a degree-$(k-1)$ packed Shamir sharing of $\boldsymbol{c}$, denoted by $[\![\boldsymbol{c}]\!]_{k-1}$. Note that for a degree-$(k-1)$ packed Shamir sharing, all shares are determined by the secrets.
2. All parties then locally compute $[\![\boldsymbol{c} * \boldsymbol{x}]\!]_{n-1} = [\![\boldsymbol{c}]\!]_{k-1} * [\![\boldsymbol{x}]\!]_{n-k}$.

We simply write $[\![\boldsymbol{c} * \boldsymbol{x}]\!]_{n-1} = \boldsymbol{c} * [\![\boldsymbol{x}]\!]_{n-k}$ to denote the above process. This property is referred to as multiplication-friendliness in [GPS22].

To make sure that the packed Shamir secret sharing scheme is secure against $t$ corrupted parties, we also require $d \geq t+k-1$. When $d = n-k$ and $k = (n-t+1)/2 = (n+3)/4$, the degree-$(n-k)$ packed Shamir secret sharing scheme is both multiplication-friendly and secure against $t$ corrupted parties.

*Using the Multiplication Protocols of [GPS22] in Our Setting.* Recall that the work of [BBG+21] focuses on the sub-optimal corruption threshold. Let $t'$ denote the corruption threshold in [BBG+21]. They use a degree-$t$ (where $t = (n-1)/2$) packed Shamir sharing to store $k' = t - t' + 1$ secrets, which allows all parties to locally compute a degree-$(n-1)$ packed Shamir sharing of $\boldsymbol{\mu}_\gamma$. To see this, note that $\boldsymbol{\mu}_\gamma = \boldsymbol{v}_\gamma - \boldsymbol{\lambda}_\gamma = \boldsymbol{v}_\alpha * \boldsymbol{v}_\beta - \boldsymbol{\lambda}_\gamma$. Since all parties hold $[\![\boldsymbol{v}_\alpha]\!]_t, [\![\boldsymbol{v}_\beta]\!]_t, [\![\boldsymbol{\lambda}_\gamma]\!]_t$, they can locally compute

$$[\![\boldsymbol{\mu}_\gamma]\!]_{n-1} = [\![\boldsymbol{v}_\alpha]\!]_t * [\![\boldsymbol{v}_\beta]\!]_t - [\![\boldsymbol{\lambda}_\gamma]\!]_t.$$

The resulting sharing has degree $(n-1)$ because $n-1 = 2 \cdot t$. Then they can reconstruct $\boldsymbol{\mu}_\gamma$ to $P_1$ by sending their shares to $P_1$.

Unfortunately, the approach in [BBG+21] does not work in our setting. This is because the corruption threshold in our setting is already $t$.

– On one hand, if we keep using the same degree $d = t$ for the packed Shamir sharing, we can only pack $1 = d - t + 1$ secret in each sharing.
– On the other hand, if we choose to use a larger degree $d > t$, the resulting sharing would have degree $2d > 2t = n-1$, which cannot be reconstructed by all parties.

To evaluate multiplication gates, we rely on the technique of packed Beaver triples in [GPS22], which is a generalization of the technique of Beaver triples in [Bea92]. Informally, the idea is to compute the multiplication between two packed Shamir sharings of $[\![\boldsymbol{x}]\!]_d, [\![\boldsymbol{y}]\!]_d$ by using a packed Beaver triple $([\![\boldsymbol{a}]\!]_d, [\![\boldsymbol{b}]\!]_d, [\![\boldsymbol{c}]\!]_d)$ such that $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{F}^k$ are random vectors and $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$. Similarly to the technique of Beaver triple, all parties first reconstruct $\boldsymbol{x} + \boldsymbol{a}$ and $\boldsymbol{y} + \boldsymbol{b}$. Then, they can use $\boldsymbol{x} + \boldsymbol{a}, \boldsymbol{y} + \boldsymbol{b}$ and $([\![\boldsymbol{a}]\!]_d, [\![\boldsymbol{b}]\!]_d, [\![\boldsymbol{c}]\!]_d)$ to locally compute a packed Shamir sharing of $\boldsymbol{x} * \boldsymbol{y}$.

We adapt the technique of packed Beaver triples in [GPS22] to our setting as follows. We set $k = (n+3)/4$ and $d = t+k-1 = n-k$. Recall that in this way, the degree-$(n-k)$ packed Shamir secret sharing scheme is both multiplication-friendly and secure against $t$ corrupted parties. For a group of multiplication gates with input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$ and output wires $\boldsymbol{\gamma}$, observe that

$$\begin{aligned}\boldsymbol{\mu}_\gamma = \boldsymbol{v}_\alpha * \boldsymbol{v}_\beta - \boldsymbol{\lambda}_\gamma &= (\boldsymbol{\mu}_\alpha + \boldsymbol{\lambda}_\alpha) * (\boldsymbol{\mu}_\beta + \boldsymbol{\lambda}_\beta) - \boldsymbol{\lambda}_\gamma \\ &= \boldsymbol{\mu}_\alpha * \boldsymbol{\mu}_\beta + \boldsymbol{\mu}_\alpha * \boldsymbol{\lambda}_\beta + \boldsymbol{\mu}_\beta * \boldsymbol{\lambda}_\alpha + \boldsymbol{\lambda}_\alpha * \boldsymbol{\lambda}_\beta - \boldsymbol{\lambda}_\gamma.\end{aligned}$$

Recall that all parties hold $[\![\boldsymbol{\lambda}_\alpha]\!]_{n-k}, [\![\boldsymbol{\lambda}_\beta]\!]_{n-k}$ prepared in the circuit-dependent preprocessing phase. To compute the above equation, we require that

– In the circuit-dependent preprocessing phase, all parties also prepare a degree-$(n-1)$ packed Shamir sharing of $\boldsymbol{\Gamma}_\gamma = \boldsymbol{\lambda}_\alpha * \boldsymbol{\lambda}_\beta - \boldsymbol{\lambda}_\gamma$ as described later.
– In the online phase, the first party $P_1$ distributes $\boldsymbol{\mu}_\alpha, \boldsymbol{\mu}_\beta$ by using degree-$(k-1)$ packed Shamir sharings.

In this way, all parties can compute

$$\begin{aligned}[\![\boldsymbol{\mu}_\gamma]\!]_{n-1} = &[\![\boldsymbol{\mu}_\alpha]\!]_{k-1} * [\![\boldsymbol{\mu}_\beta]\!]_{k-1} + [\![\boldsymbol{\mu}_\alpha]\!]_{k-1} * [\![\boldsymbol{\lambda}_\beta]\!]_{n-k} \\ &+ [\![\boldsymbol{\mu}_\beta]\!]_{k-1} * [\![\boldsymbol{\lambda}_\alpha]\!]_{n-k} + [\![\boldsymbol{\Gamma}_\gamma]\!]_{n-1}.\end{aligned}$$

*Summary of the Online Protocol.* For each group of multiplication gates, $P_1$ needs to distribute two degree-$(k-1)$ packed Shamir sharings and all parties need to send their shares of $[\![\boldsymbol{\mu}_{\boldsymbol{\gamma}}]\!]_{n-1}$ to $P_1$. Thus, the total communication complexity is $3n$ field elements per group of $k$ multiplication gates. On average, the amortized communication complexity per multiplication gate is $3 \cdot n/k \approx 12$ elements.

### 2.2 Realizing Circuit-Dependent Preprocessing Phase

In the circuit-dependent preprocessing phase, our goal is to prepare the following two kinds of packed Shamir sharings: for each group of $k$ multiplication gates,

- For the input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$, all parties prepare $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}]\!]_{n-k}, [\![\boldsymbol{\lambda}_{\boldsymbol{\beta}}]\!]_{n-k}$.
- For the output wires $\boldsymbol{\gamma}$, all parties prepare $[\![\boldsymbol{\Gamma}_{\boldsymbol{\gamma}}]\!]_{n-1}$, where $\boldsymbol{\Gamma}_{\boldsymbol{\gamma}} = \boldsymbol{\lambda}_{\boldsymbol{\alpha}} * \boldsymbol{\lambda}_{\boldsymbol{\beta}} - \boldsymbol{\lambda}_{\boldsymbol{\gamma}}$.

Recall that the random value $\lambda_\alpha$ associated with each wire $\alpha$ satisfies that

- For each output wire $\alpha$ of input gates and multiplication gates, $\lambda_\alpha$ is uniformly random.
- For each addition gate with input wires $\alpha, \beta$ and output wire $\gamma$, $\lambda_\gamma = \lambda_\alpha + \lambda_\beta$.

Although we only need to prepare packed Shamir sharings for the input wires of multiplication gates, we need to first generate uniform values that are associated with the output wires of input gates and multiplication gates, and then compute the random values associated with the input wires of multiplication gates. We survey the potential solutions from [BBG$^+$21] and [GPS22].

*The Solution in [BBG$^+$21].* In [BBG$^+$21], the authors rely on pseudo-random secret sharings to prepare the random packed Shamir sharings for the input wires of multiplication gates. However, this approach requires to use pseudo-random generators, which means that they are NOT in the IT setting. And it only works when the number of corrupted parties is a constant since their construction is based on the replicated secret sharing scheme where the share size grows exponentially with the number of corrupted parties.

*The Solution in [GPS22].* We can potentially use the protocol in [GPS22] to prepare the random packed Shamir sharings for the input wires of multiplication gates. As we analysed above, it can achieve $O(n/k)$ elements of communication per secret in the circuit-dependent preprocessing phase with $O(n)$ elements of communication in the circuit-independent preprocessing phase. However, directly using the approach in [GPS22] has the following two drawbacks:

- It requires $O(\mathsf{Depth})$ rounds in the circuit-dependent preprocessing phase. This is because the protocol in [GPS22] also needs to interact for addition gates, and the computation of addition gates is done layer by layer.
- As discussed in the introduction, the constant factor hidden in the big-$O$ notation is very large.

*Our Solution.* Our idea is to first prepare a *single* packed Shamir sharing for *each wire*. Concretely, for each output wire $\alpha$ of input gates and multiplication gates, all parties prepare a random degree-$(n-k)$ packed Shamir sharing in the form $[\![\lambda_\alpha \cdot \mathbf{1}]\!]_{n-k}$, where $\mathbf{1} = (1, \dots, 1) \in \mathbb{F}^k$. In other words, the secrets of $[\![\lambda_\alpha \cdot \mathbf{1}]\!]_{n-k}$ are $k$ copies of the same value $\lambda_\alpha$.

Then all parties can *locally* compute $[\![\lambda_\alpha \cdot \mathbf{1}]\!]_{n-k}$ for each wire $\alpha$, which is the output of an addition gate. This is done by adding the two packed sharings associated with the input wires of an addition gate. Next, for each group of $k$ multiplication gates with input wires $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$, suppose $\boldsymbol{\alpha} = (\alpha_1, \dots, \alpha_k)$. All parties can locally compute a degree-$(n-1)$ packed Shamir sharing of $\boldsymbol{\lambda}_{\boldsymbol{\alpha}}$ by

$$[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}]\!]_{n-1} = \boldsymbol{e}_1 * [\![\lambda_{\alpha_1} \cdot \mathbf{1}]\!]_{n-k} + \dots + \boldsymbol{e}_k * [\![\lambda_{\alpha_k} \cdot \mathbf{1}]\!]_{n-k},$$

where $\boldsymbol{e}_i$ is the $i$-th unit vector in $\mathbb{F}^k$ satisfying that all entries of $\boldsymbol{e}_i$ are $0$ except the $i$-th entry is $1$. To see why this is true, note that the secrets of the RHS are equal to

$$\boldsymbol{e}_1 * (\lambda_{\alpha_1} \cdot \mathbf{1}) + \dots + \boldsymbol{e}_k * (\lambda_{\alpha_k} \cdot \mathbf{1}) = \lambda_{\alpha_1} \cdot \boldsymbol{e}_1 + \dots + \lambda_{\alpha_k} \cdot \boldsymbol{e}_k = \boldsymbol{\lambda}_{\boldsymbol{\alpha}}.$$

To obtain $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}]\!]_{n-k}$ from $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}]\!]_{n-1}$, all parties perform a degree-reduction step. As we will see later, the degree-reduction step is merged with the computation of $[\![\boldsymbol{\Gamma}_{\boldsymbol{\gamma}}]\!]_{n-1}$.

As a result, our approach avoids the expensive network routing and achieves constant rounds.

*Preparing Packed Shamir Sharings for $\{\boldsymbol{\Gamma}_\gamma\}_\gamma$.* For each group of multiplication gates, let $\boldsymbol{\alpha}, \boldsymbol{\beta}$ denote the input wires and $\gamma$ denote the output wires. Recall that in the last step, all parties have locally computed $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}]\!]_{n-1}, [\![\boldsymbol{\lambda}_{\boldsymbol{\beta}}]\!]_{n-1}$. Similarly, they can locally compute $[\![\boldsymbol{\lambda}_\gamma]\!]_{n-1}$. To compute $[\![\boldsymbol{\Gamma}_\gamma]\!]_{n-1}$, where $\boldsymbol{\Gamma}_\gamma = \boldsymbol{\lambda}_{\boldsymbol{\alpha}} * \boldsymbol{\lambda}_{\boldsymbol{\beta}} - \boldsymbol{\lambda}_\gamma$, the main task is to compute a packed Shamir sharing of the multiplication result $\boldsymbol{\lambda}_{\boldsymbol{\alpha}} * \boldsymbol{\lambda}_{\boldsymbol{\beta}}$.

We again rely on the technique of packed Shamir sharing in [GPS22]. Concretely, all parties first prepare a random packed Beaver triple $([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, [\![\boldsymbol{c}]\!]_{n-k})$, where $\boldsymbol{a}, \boldsymbol{b}$ are random vectors in $\mathbb{F}^k$ and $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$. Then all parties perform the following steps:

1. All parties locally compute $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a}]\!]_{n-1} = [\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}]\!]_{n-1}+[\![\boldsymbol{a}]\!]_{n-k}$ and $[\![\boldsymbol{\lambda}_{\boldsymbol{\beta}}+\boldsymbol{b}]\!]_{n-1} = [\![\boldsymbol{\lambda}_{\boldsymbol{\beta}}]\!]_{n-1}+[\![\boldsymbol{b}]\!]_{n-k}$[7].
2. The first party $P_1$ collects the whole sharings $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \boldsymbol{a}]\!]_{n-1}, [\![\boldsymbol{\lambda}_{\boldsymbol{\beta}} + \boldsymbol{b}]\!]_{n-1}$ and reconstructs the secrets $\boldsymbol{d}_1 = \boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \boldsymbol{a}, \boldsymbol{d}_2 = \boldsymbol{\lambda}_{\boldsymbol{\beta}} + \boldsymbol{b}$. Then $P_1$ distributes $[\![\boldsymbol{d}_1]\!]_{k-1}, [\![\boldsymbol{d}_2]\!]_{k-1}$ to all parties.
3. All parties locally compute
$$\begin{aligned}[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}} * \boldsymbol{\lambda}_{\boldsymbol{\beta}}]\!]_{n-1} &= [\![\boldsymbol{d}_1]\!]_{k-1} * [\![\boldsymbol{d}_2]\!]_{k-1} - [\![\boldsymbol{d}_1]\!]_{k-1} * [\![\boldsymbol{b}]\!]_{n-k} \\ &\quad - [\![\boldsymbol{d}_2]\!]_{k-1} * [\![\boldsymbol{a}]\!]_{n-k} + [\![\boldsymbol{c}]\!]_{n-1}.\end{aligned}$$
The correctness follows from the fact that $\boldsymbol{\lambda}_{\boldsymbol{\alpha}} = \boldsymbol{d}_1 - \boldsymbol{a}, \boldsymbol{\lambda}_{\boldsymbol{\beta}} = \boldsymbol{d}_2 - \boldsymbol{b}$, and
$$\begin{aligned}\boldsymbol{\lambda}_{\boldsymbol{\alpha}} * \boldsymbol{\lambda}_{\boldsymbol{\beta}} &= (\boldsymbol{d}_1 - \boldsymbol{a}) * (\boldsymbol{d}_2 - \boldsymbol{b}) \\ &= \boldsymbol{d}_1 * \boldsymbol{d}_2 - \boldsymbol{d}_1 * \boldsymbol{b} - \boldsymbol{d}_2 * \boldsymbol{a} + \boldsymbol{a} * \boldsymbol{b} \\ &= \boldsymbol{d}_1 * \boldsymbol{d}_2 - \boldsymbol{d}_1 * \boldsymbol{b} - \boldsymbol{d}_2 * \boldsymbol{a} + \boldsymbol{c}.\end{aligned}$$

Note that, all parties can also locally compute $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}]\!]_{n-k} = [\![\boldsymbol{d}_1]\!]_{k-1} - [\![\boldsymbol{a}]\!]_{n-k}$ and $[\![\boldsymbol{\lambda}_{\boldsymbol{\beta}}]\!]_{n-k} = [\![\boldsymbol{d}_2]\!]_{k-1} - [\![\boldsymbol{b}]\!]_{n-k}$. Thus, the degree-reduction steps have been implicit performed above.

*Summary of the Circuit-Dependent Preprocessing Phase.* In the circuit-dependent preprocessing phase, for each group of $k$ multiplication gates, all parties need to send their shares of $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \boldsymbol{a}]\!]_{n-1}, [\![\boldsymbol{\lambda}_{\boldsymbol{\beta}} + \boldsymbol{b}]\!]_{n-1}$ to $P_1$, and $P_1$ needs to distribute $[\![\boldsymbol{d}_1]\!]_{k-1}, [\![\boldsymbol{d}_2]\!]_{k-1}$ to all parties. Therefore, the communication complexity per multiplication gate is $4 \cdot n/k \approx 16$ elements.

Note that the random packed Shamir sharings in the form of $[\![\lambda_\alpha \cdot \mathbf{1}]\!]_{n-k}$ and the random packed Beaver triples in the form of $([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, [\![\boldsymbol{c}]\!]_{n-k})$ are prepared in the circuit-independent preprocessing phase as discussed below.

## 2.3 Realizing Circuit-Independent Preprocessing Phase

In the circuit-independent preprocessing phase, our goal is to prepare the following random sharings.

- For each output wire of input gates and multiplication gates, all parties prepare a random degree-$(n-k)$ packed Shamir sharing in the form of $[\![\lambda_\alpha \cdot \mathbf{1}]\!]_{n-k}$.
- For each group of multiplication gates, all parties prepare a random packed Beaver triples $([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, [\![\boldsymbol{c}]\!]_{n-k})$.

*Preparing Random Sharings for a Given Linear Secret Sharing Scheme.* Let $\Sigma$ be a linear secret sharing scheme in $\mathbb{F}$. To prepare random $\Sigma$-sharings, we follow a similar approach to that in [DN07]. At a high-level,

1. Each party $P_j$ first generates and distributes a random $\Sigma$ sharing, denoted by $\boldsymbol{S}^{(j)}$.
2. Let $\boldsymbol{M}^{\mathrm{T}}$ be a Vandermonde matrix of size $n \times (t+1)$ in $\mathbb{F}$. All parties use $\boldsymbol{M}$ as a random extractor to extract $n - t = t + 1$ random sharings. This is done by simply computing $(\boldsymbol{R}^{(1)}, \dots, \boldsymbol{R}^{(t+1)})^{\mathrm{T}} = \boldsymbol{M}(\boldsymbol{S}^{(1)}, \dots, \boldsymbol{S}^{(n)})^{\mathrm{T}}$.

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 $\Sigma$ is a linear secret sharing scheme. Thus, all parties will output valid $\Sigma$-sharings in the above approach. The security follows from the fact that any sub-matrix of size $(t+1) \cdot (t+1)$ of an $n \times (t+1)$ 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.

We can use the above approach to prepare random sharings in the form of $[\![\lambda_\alpha \cdot \mathbf{1}]\!]_{n-k}$. The communication complexity per sharing is $2n$ elements.

[7] In TurboPack, we need to use a random degree-$(n-1)$ packed Shamir sharing of $\mathbf{0} \in \mathbb{F}^k$ to protect the shares of honest parties. In the technical overview, we omit this issue for simplicity

*Preparing Packed Beaver Triples.* For $([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, [\![\boldsymbol{c}]\!]_{n-k})$, the first two sharings can be prepared by using the above approach. However, we do not know how to efficiently compute a packed Shamir sharing of the multiplication result $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$ from $[\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}$.

Our idea is to first prepare $k$ standard Beaver triples by using degree-$t$ Shamir sharings and then transform them to a packed Beaver triple. To simplify the transformation, we choose to use different secret slots in different degree-$t$ Shamir sharings.

**Shamir Sharings with Different Secret Slots**: With more details, recall that a degree-$t$ Shamir secret sharing scheme corresponds to a degree-$t$ polynomial $f$ such that $f(1), \ldots, f(n)$ are the shares and $f(0)$ is the secret. However, we do not need to always use the evaluation point $0$ to store the secret. Concretely, for all $i \in \{1, \ldots, k\}$, we use $[\![x|_i]\!]_t$ to denote a degree-$t$ Shamir sharing whose secret is stored at the evaluation point $-i+1$[8]. I.e., the corresponding polynomial $f$ satisfies that (1) $f$ has degree $t$, (2) $f(1), \ldots, f(n)$ are the shares, and (3) $f(-i+1)$ is the secret.

**Transforming to Packed Sharings**: Let $\boldsymbol{e}_i$ denote the $i$-th unit vector. Now suppose all parties hold $k$ degree-$t$ Shamir sharings $\{[\![x_i|_i]\!]_t\}_{i=1}^{n}$. We observe that $[\![x_i|_i]\!]_t$ can be viewed as a degree-$t$ packed Shamir sharing with the $i$-th secret to be $x_i$. Therefore, all parties can *locally* convert them to a degree-$(n-k)$ packed Shamir sharing by computing $[\![\boldsymbol{x}]\!]_{n-k} = \boldsymbol{e}_1 * [\![x_1|_1]\!]_t + \ldots + \boldsymbol{e}_k * [\![x_k|_k]\!]_t$.

**Preparing Standard Beaver Triples**: Thus, the problem is reduced to prepare $\{([\![a_i|_i]\!]_t, [\![b_i|_i]\!]_t, [\![c_i|_i]\!]_t)\}_{i=1}^{k}$. For $[\![a_i|_i]\!]_t, [\![b_i|_i]\!]_t$, we can use the above approach to prepare them, which costs $4n$ elements. To compute $[\![c_i|_i]\!]_t$, we rely on the state-of-the-art multiplication protocol [GLO+21] in the standard honest majority setting, which costs $4n$ elements. The communication complexity of preparing packed Beaver triples is $8n$ elements per multiplication gate.

*Summary of the Circuit-Independent Preprocessing Phase.* Beyond the above two kinds of random sharings, we also need to prepare 3 random degree-$(n-1)$ packed Shamir sharings of $\mathbf{0} \in \mathbb{F}^k$ for each group of multiplication gates. These random sharings of $\mathbf{0}$ are used to protect the shares of honest parties. By using the above approach to prepare them, the communication complexity per multiplication gate is $6 \cdot n/k \approx 24$ elements.

In summary, our circuit-independent preprocessing phase has communication complexity $2n + 8n + 24 = 10n + 24$ elements per multiplication gate.

## 2.4 An Optimization of TurboPack

We note that TurboPack uses the technique of packed Beaver triples [GPS22] two times. The first time is in the online phase where all parties need to compute a packed Shamir sharing of $\boldsymbol{v_\alpha} * \boldsymbol{v_\beta}$ for each group of multiplication gates. Here, all parties hold degree-$(n-k)$ packed Shamir sharings of $\boldsymbol{\lambda_\alpha}, \boldsymbol{\lambda_\beta}$ and the first party $P_1$ distributes two degree-$(k-1)$ packed Shamir sharings of $\boldsymbol{\mu_\alpha}, \boldsymbol{\mu_\beta}$. The second time is in the circuit-dependent preprocessing phase where all parties need to compute a packed Shamir sharing of $\boldsymbol{\lambda_\alpha} * \boldsymbol{\lambda_\beta}$ for each group of multiplication gates. Here, all parties hold degree-$(n-k)$ packed Shamir sharings of $\boldsymbol{a}, \boldsymbol{b}$ and the first party $P_1$ distributes two degree-$(k-1)$ packed Shamir sharings of $\boldsymbol{\lambda_\alpha} + \boldsymbol{a}$ and $\boldsymbol{\lambda_\beta} + \boldsymbol{b}$.

We observe that we can directly use the packed Beaver triple $([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, [\![\boldsymbol{c}]\!]_{n-k})$ in the online phase. This requires $P_1$ to distribute degree-$(k-1)$ packed Shamir sharings of $\boldsymbol{v_\alpha} + \boldsymbol{a}, \boldsymbol{v_\beta} + \boldsymbol{b}$. For $\boldsymbol{v_\alpha} + \boldsymbol{a}$, note that $P_1$ learns $\boldsymbol{\mu_\alpha}$ in the online phase and $\boldsymbol{\lambda_\alpha} + \boldsymbol{a}$ in the circuit-dependent preprocessing phase, and $\boldsymbol{v_\alpha} + \boldsymbol{a} = \boldsymbol{\mu_\alpha} + \boldsymbol{\lambda_\alpha} + \boldsymbol{a}$. Thus, instead of asking $P_1$ to distribute a degree-$(k-1)$ packed Shamir sharing of $\boldsymbol{\mu_\alpha}$ in the online phase and a degree-$(k-1)$ packed Shamir sharing of $\boldsymbol{\lambda_\alpha} + \boldsymbol{a}$ in the circuit-dependent preprocessing phase, we can let $P_1$ only distribute a degree-$(k-1)$ packed Shamir sharing of $\boldsymbol{\mu_\alpha} + \boldsymbol{\lambda_\alpha} + \boldsymbol{a} = \boldsymbol{v_\alpha} + \boldsymbol{b}$ in the online phase. In this way, we can save the cost in the circuit-dependent preprocessing phase by a factor of 2.

We refer the readers to Section C for more details.

## 2.5 Towards Malicious Security

In this part, we discuss how to achieve the malicious security without affecting the concrete efficiency. The main difficulty comes from the fact that degree-$(n-k)$ packed Shamir sharing is not robust: corrupted parties can change the secrets of a degree-$(n-k)$ packed Shamir sharing by locally changing their own shares. This is different from IT MPC protocols that are based on degree-$t$

[8] Here we assume $\{-k+1, \ldots, n\}$ are $n+k$ distinct field elements in $\mathbb{F}$. For a general field, they can be replaced by any $n+k$ distinct field elements.

(packed) Shamir sharings, where the whole sharing is fully determined by the shares of honest parties. It also means that the verification protocols in the recent IT MPC protocols with honest majority [GS20, BGIN20, GLO+21] do not work.

To add robustness, the work [GPS22] relies on IT MACs. However, the use of IT MACs would increase the communication complexity by a factor of 2 and require a large enough finite field.

Recall that in the circuit-independent preprocessing phase, all parties prepare degree-$(n-k)$ packed Shamir sharings $[\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}$ for each group of multiplication gates. In the online phase, all parties receive from $P_1$ two degree-$(k-1)$ packed Shamir sharings $[\![\boldsymbol{v_\alpha}+\boldsymbol{a}]\!]_{k-1}, [\![\boldsymbol{v_\beta}+\boldsymbol{b}]\!]_{k-1}$. We observe that

- For $[\![\boldsymbol{a}]\!]_{n-k}$, all parties first prepare $k$ individual degree-$t$ Shamir sharings $\{[\![a_i|_i]\!]_t\}_{i=1}^k$ and then transform them to a degree-$(n-k)$ packed Shamir sharing.
- For $[\![\boldsymbol{v_\alpha}+\boldsymbol{a}]\!]_{k-1}$, we can view it as a degree-$(k-1)$ Shamir sharing of $v_{\alpha_i}+a_i$ stored at the $i$-th secret slot, i.e., $[\![v_{\alpha_i}+a_i|_i]\!]_{k-1}$.

Thus, if we keep the individual degree-$t$ Shamir sharings $\{[\![a_i|_i]\!]_t\}_{i=1}^k$, all parties can locally compute

$$[\![\boldsymbol{v_\alpha}+\boldsymbol{a}]\!]_{k-1} - [\![a_i|_i]\!]_t = [\![v_{\alpha_i}+a_i|_i]\!]_{k-1} - [\![a_i|_i]\!]_t = [\![v_{\alpha_i}|_i]\!]_t.$$

In this way, all parties can compute an individual degree-$t$ Shamir sharing for each input wire of multiplication gates.

To check the correctness of the computation, note that each input of multiplication gates is equal to some fixed linear combination of the outputs of multiplication gates in previous layers. Also note that the output of a multiplication gate can be written as the product of its two inputs. Thus, for each input of multiplication gates, what we want to verify is an inner-product. At this stage, we still cannot use the verification protocols in [GS20, BGIN20, GLO+21] since the secrets of these degree-$t$ Shamir sharings do not use the same secret slot.

*Verification Protocol in [BBG+21].* Recall that the work [BBG+21] focuses on the sub-optimal corruption threshold and uses a degree-$t$ packed Shamir sharing to store $k' = t - t' + 1$ secrets, where $t'$ is the corruption threshold in [BBG+21]. In their verification protocol, however, the authors view each degree-$t$ packed Shamir sharing $[\![\boldsymbol{x}]\!]_t$ as a degree-$t$ Shamir sharing for each secret $x_i$, i.e., $[\![x_i|_i]\!]_t$. Thus, although our setting is different from that in [BBG+21], we can potentially use the verification protocol in [BBG+21].

The verification protocol in [BBG+21] first transforms the check of $|C|$ inner-products into one check of a single inner-product. Then they adapt the technique in [BBCG+19] to verify the single inner-product and achieves sub-linear communication complexity in the circuit size. However, the verification protocol in [BBG+21] does not use the technique in [BBCG+19] in a black box way. It has computation complexity $O(|C| \cdot \sqrt{|C|})$ due to the use of the technique in [BBCG+19] which can be a bottleneck for the concrete efficiency.

*Our Solution.* Our verification protocol is also based on [BBCG+19] but we manage to use the technique in [BBCG+19] in a black-box way. It allows us to directly use other variants of the techniques in [BBCG+19] in a black box way, for example, the verification protocol in [GS20], which naturally offers a trade-off between the round complexity and the computation complexity. Concretely, for all $d < \sqrt{|C|}$, the verification protocol in [GS20] can achieve $O(|C| \cdot d)$ computation complexity at the cost of $\log_d |C|$ rounds. This trade-off is also explored in the work [BGIN19] for 3-party setting and [BGIN20] for $n$-party setting.

Another issue that is not noticed in [BBG+21] is that directly transforming the check of $|C|$ inner-products into one check of a single inner-product may cost $O(|C|^2)$ local computation. This is because an input of a multiplication gate can be a linear combination of $O(|C|)$ outputs of multiplication gates in the previous layers. Merging $|C|$ inner-products, where each has size $O(|C|)$, into one inner-product would cost $O(|C|^2)$ local computation in the worst case. We show how to efficiently compute the single inner-product that all parties need to check with $O(|C|)$ computation complexity.

As a result, our verification protocol also achieves sub-linear communication complexity in the circuit size and is computationally efficient. We refer the readers to Section D for more details.

## 3 Preliminaries

### 3.1 The Model

We consider a set of parties $\{P_1, P_2, ..., P_n\}$ where each party can provide inputs, receive outputs, and participate in the computation. For every pair of parties, there exists a secure (private and authentic) synchronous channel so that they can directly send messages to each other. The communication complexity is measured by the number of bits $X$ via private channels.

We focus on functions which can be represented as arithmetic circuits over a finite field $\mathbb{F}$ with input, addition, multiplication, and output gates[9]. We use $\kappa$ to denote the security parameter and let $\mathbb{K}$ be an extension field of $\mathbb{F}$ (with $|\mathbb{K}| \geq 2^\kappa$). For simplicity, we use $\kappa$ to denote the size of an element in $\mathbb{K}$. In this work, we assume that the number of parties $n$ and the circuit size $|C|$ are bounded by polynomials of the security parameter $\kappa$.

In this work, we focus on the honest majority setting, where the number of corrupted parties $t = (n-1)/2$. We refer the readers to Section A for the security definition.

*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. 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. Note that it does not hold in the standard MPC model. We refer the readers to Section A for more details.

### 3.2 Packed Shamir Secret Sharing Scheme

In our work, we are interested in the packed Shamir secret sharing scheme. We use the packed secret-sharing technique introduced by Franklin and Yung [FY92]. This is a generalization of the standard Shamir secret sharing scheme [Sha79]. Let $\mathbb{F}$ be a finite field of size $|\mathbb{F}| \geq 2n$. Let $n$ be the number of parties and $k$ be the number of secrets that are packed in one sharing. A *degree-$d$* ($d \geq k-1$) packed Shamir sharing of $\boldsymbol{x} = (x_1, \ldots, x_k) \in \mathbb{F}^k$ is a vector $(w_1, \ldots, w_n)$ 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 $i \in \{1, 2, \ldots, k\}$, and $f(i) = w_i$ for all $i \in \{1, 2, \ldots, n\}$. 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 $\boldsymbol{x}$.

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. We recall two properties of the packed Shamir sharing scheme:

- Linear Homomorphism: For all $d \geq k-1$ and $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}^k$, $[\![\boldsymbol{x}+\boldsymbol{y}]\!]_d = [\![\boldsymbol{x}]\!]_d + [\![\boldsymbol{y}]\!]_d$.
- Multiplicative: 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 $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}^k$, $[\![\boldsymbol{x} * \boldsymbol{y}]\!]_{d_1+d_2} = [\![\boldsymbol{x}]\!]_{d_1} \cdot [\![\boldsymbol{y}]\!]_{d_2}$.

These two properties directly follow from the computation of the underlying polynomials.

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 $[\![\boldsymbol{c} * \boldsymbol{x}]\!]_{d+k-1} = [\![\boldsymbol{c}]\!]_{k-1} \cdot [\![\boldsymbol{x}]\!]_d$. This property is referred to as multiplication-friendliness in [GPS22].

Recall that $t$ is the number of corrupted parties. Also recall that a degree-$d$ packed Shamir secret sharing scheme is secure against $t-k+1$ corrupted parties. When setting $k = (n-t+1)/2 = (n+3)/4$, a degree-$(n-k)$ packed Shamir sharing is both secure against $t$ corrupted parties and multiplication-friendly.

[9] 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.

*Shamir Secret Sharing Schemes with Different Secret Slots.* 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 $i \in \{1, \ldots, k\}$, 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 $f(-i + 1) = x$.

## 4 Efficient MPC via Packing with Semi-honest Security

Recall that, we use $\mathsf{c}$ to denote the number of clients and $n$ to denote the number of parties. Also recall the corruption threshold $t = (n-1)/2$ and the packing parameter $k = (n-t+1)/2 = (n+3)/4$.

### 4.1 Ideal Functionality for Circuit-Dependent Preprocessing

We first give the ideal functionality $\mathcal{F}_{\mathsf{Prep}}$, given as Functionality 1 below, that prepares correlated randomness for the online phase. We consider the *circuit-dependent* preprocessing phase. I.e., the functionality will take as input the circuit $C$ without the real inputs. We will explain the reason of generating these random sharings when introducing the online protocol in the next part.

**Functionality 1: $\mathcal{F}_{\mathsf{Prep}}$**

1. **Assign Random Values to Wires in** $C$: $\mathcal{F}_{\mathsf{Prep}}$ receives the circuit $C$ from all parties.
   (a) For each output wire $\alpha$ of an input gate or a multiplication gate, $\mathcal{F}_{\mathsf{Prep}}$ samples a uniform value $\lambda_\alpha$ and associates it with the wire $\alpha$.
   (b) Starting from the first layer of $C$ to the last layer, for each addition gate with input wires $\alpha, \beta$ and output wire $\gamma$, $\mathcal{F}_{\mathsf{Prep}}$ sets $\lambda_\gamma = \lambda_\alpha + \lambda_\beta$.
2. **Preparing Degree-**$(n-k)$ **Packed Shamir Sharings**: $\mathcal{F}_{\mathsf{Prep}}$ receives the set of corrupted parties, denoted by $\mathcal{C}orr$. For each intermediate layer in $C$, all multiplication gates are divided into groups of size $k$. For each group of multiplication gates with input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$:
   (a) $\mathcal{F}_{\mathsf{Prep}}$ receives from the adversary a set of shares $\{u_j^{(1)}, u_j^{(2)}\}_{j \in \mathcal{C}orr}$. $\mathcal{F}_{\mathsf{Prep}}$ computes degree-$(n-k)$ packed Shamir sharings $[\![\boldsymbol{\lambda_\alpha}]\!]_{n-k}, [\![\boldsymbol{\lambda_\beta}]\!]_{n-k}$ such that for all $P_j \in \mathcal{C}orr$, the $j$-th share of $([\![\boldsymbol{\lambda_\alpha}]\!]_{n-k}, [\![\boldsymbol{\lambda_\beta}]\!]_{n-k})$ is $(u_j^{(1)}, u_j^{(2)})$.
   (b) $\mathcal{F}_{\mathsf{Prep}}$ distributes the shares of $[\![\boldsymbol{\lambda_\alpha}]\!]_{n-k}, [\![\boldsymbol{\lambda_\beta}]\!]_{n-k}$ to honest parties.
3. **Preparing Degree-**$(n-1)$ **Packed Shamir Sharings**: For the input layer, all input gates are divided into groups of size $k$ such that the input gates of each group belong to the same client. For each group of input gates with output wires $\boldsymbol{\alpha}$:
   (a) $\mathcal{F}_{\mathsf{Prep}}$ receives from the adversary a set of shares $\{u_j\}_{j \in \mathcal{C}orr}$. $\mathcal{F}_{\mathsf{Prep}}$ samples a random degree-$(n-1)$ packed Shamir sharing $[\![\boldsymbol{\lambda_\alpha}]\!]_{n-1}$ such that for all $P_j \in \mathcal{C}orr$, the $j$-th share of $[\![\boldsymbol{\lambda_\alpha}]\!]_{n-1}$ is $u_j$.
   (b) $\mathcal{F}_{\mathsf{Prep}}$ distributes the shares of $[\![\boldsymbol{\lambda_\alpha}]\!]_{n-1}$ to honest parties.
   Similarly, for the output layer in $C$, all output gates are divided into groups of size $k$ such that the output gates of each group belong to the same client. For each group of output gates with input wires $\boldsymbol{\alpha}$, $\mathcal{F}_{\mathsf{Prep}}$ prepares and distributes $[\![\boldsymbol{\lambda_\alpha}]\!]_{n-1}$ in the same way as above.
4. **Preparing Packed Beaver Triples**: For each group of multiplication gates with input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$ and output wires $\boldsymbol{\gamma}$:
   (a) $\mathcal{F}_{\mathsf{Prep}}$ receives from the adversary a set of shares $\{u_j\}_{j \in \mathcal{C}orr}$. $\mathcal{F}_{\mathsf{Prep}}$ samples a random degree-$(n-1)$ packed Shamir sharing $[\![\boldsymbol{\Gamma_\gamma}]\!]_{n-1}$ such that for all $P_j \in \mathcal{C}orr$, the $j$-th share of $[\![\boldsymbol{\Gamma_\gamma}]\!]_{n-1}$ is $u_j$.
   (b) $\mathcal{F}_{\mathsf{Prep}}$ distributes the shares of $[\![\boldsymbol{\Gamma_\gamma} = \boldsymbol{\lambda_\alpha} * \boldsymbol{\lambda_\beta} - \boldsymbol{\lambda_\gamma}]\!]_{n-1}$ to honest parties.

### 4.2 Online Protocol via Packing

In the online phase, we want to maintain the invariant that for each wire $\alpha$, the first party $P_1$ learns the difference $\mu_\alpha = v_\alpha - \lambda_\alpha$, where $v_\alpha$ is the real values associated with the wire $\alpha$. Then at the end of the protocol, for each group of output gates that belong to some `Client`, all parties will send their shares of $[\![\boldsymbol{\lambda_\alpha}]\!]_{n-1}$ to `Client`, where $\boldsymbol{\alpha}$ are the input wires associated with these output gates, and $P_1$ will send $\boldsymbol{\mu_\alpha}$ to `Client`. In this way, `Client` can reconstruct his outputs $\boldsymbol{v_\alpha} = \boldsymbol{\mu_\alpha} + \boldsymbol{\lambda_\alpha}$.

We will discuss how this invariant can be achieved as follows.

*Input Phase.* Recall that in the preprocessing phase, for each group of input gates that belong to some `Client`, $\mathcal{F}_{\mathsf{Prep}}$ distributes a degree-$(n-1)$ packed Shamir sharing $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}]\!]_{n-1}$ to all parties, where $\boldsymbol{\alpha}$ are the output wires associated with these input gates. To allow $P_1$ to learn $\boldsymbol{\mu}_{\boldsymbol{\alpha}}$, `Client` first collects the whole sharing $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}]\!]_{n-1}$ from all parties, then reconstructs the secret $\boldsymbol{\lambda}_{\boldsymbol{\alpha}}$, and finally computes and sends $\boldsymbol{\mu}_{\boldsymbol{\alpha}} = \boldsymbol{v}_{\boldsymbol{\alpha}} - \boldsymbol{\lambda}_{\boldsymbol{\alpha}}$ to $P_1$. Note that here $\boldsymbol{v}_{\boldsymbol{\alpha}}$ are the inputs of `Client`. The description of the protocol $\Pi_{\mathsf{Input}}$ appears in Protocol 1. The communication complexity per batch of $k$ input gates is $n+k$ elements.

**Protocol 1:** $\Pi_{\mathsf{Input}}$

1. For each group of input gates that belong to `Client`, let $\boldsymbol{\alpha}$ denote the batch of output wires of these input gates. All parties receive the sharing $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}]\!]_{n-1}$ from $\mathcal{F}_{\mathsf{Prep}}$ and `Client` holds inputs $\boldsymbol{v}_{\boldsymbol{\alpha}}$.
2. All parties send to `Client` their shares of $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}]\!]_{n-1}$.
3. `Client` reconstructs the secret $\boldsymbol{\lambda}_{\boldsymbol{\alpha}}$ and computes $\boldsymbol{\mu}_{\boldsymbol{\alpha}} = \boldsymbol{v}_{\boldsymbol{\alpha}} - \boldsymbol{\lambda}_{\boldsymbol{\alpha}}$.
4. `Client` sends $\boldsymbol{\mu}_{\boldsymbol{\alpha}}$ to $P_1$.

*Computation Phase.* Now we discuss how $P_1$ can learn $\mu_\alpha$ for every wire $\alpha$ in the circuit $C$. This follows the idea in [BBG+21] with the change that we use the technique of packed Beaver triples introduced in [GPS22] for multiplications.

The circuit is evaluated layer by layer. Note that the invariant is achieved in the first layer (the input layer). Now assume the invariant is maintained in previous layers. I.e., $P_1$ learns $\mu_\alpha$ for every input wire $\alpha$ of the current layer since $\alpha$ serves as an output wire in previous layers. For an addition gate with input wires $\alpha, \beta$ and output wire $\gamma$, we have $v_\gamma = v_\alpha + v_\beta$. Recall that in $\mathcal{F}_{\mathsf{Prep}}$, we have $\lambda_\gamma = \lambda_\alpha + \lambda_\beta$. Therefore $P_1$ can locally compute

$$\mu_\gamma = v_\gamma - \lambda_\gamma = (v_\alpha + v_\beta) - (\lambda_\alpha + \lambda_\beta) = (v_\alpha - \lambda_\alpha) + (v_\beta - \lambda_\beta) = \mu_\alpha + \mu_\beta.$$

For multiplication gates, we follow the technique of packed Beaver triples in [GPS22]. The description of the protocol $\Pi_{\mathsf{Mult}}$ appears in Protocol 2. The communication complexity per batch of $k$ multiplication gates is $3n$ elements.

**Protocol 2:** $\Pi_{\mathsf{Mult}}$

1. For each group of multiplication gates with input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$ and output wires $\boldsymbol{\gamma}$, $P_1$ learns $\boldsymbol{\mu}_{\boldsymbol{\alpha}}, \boldsymbol{\mu}_{\boldsymbol{\beta}}$ and all parties receive three packed Shamir sharings $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}]\!]_{n-k}, [\![\boldsymbol{\lambda}_{\boldsymbol{\beta}}]\!]_{n-k}$ and $[\![\boldsymbol{\Gamma}_{\boldsymbol{\gamma}}]\!]_{n-1}$ from $\mathcal{F}_{\mathsf{Prep}}$, where $\boldsymbol{\Gamma}_{\boldsymbol{\gamma}} = \boldsymbol{\lambda}_{\boldsymbol{\alpha}} * \boldsymbol{\lambda}_{\boldsymbol{\beta}} - \boldsymbol{\lambda}_{\boldsymbol{\gamma}}$.
2. $P_1$ computes $[\![\boldsymbol{\mu}_{\boldsymbol{\alpha}}]\!]_{k-1}$ and $[\![\boldsymbol{\mu}_{\boldsymbol{\beta}}]\!]_{k-1}$ and distributes the shares to all parties.
3. All parties locally compute
$$\begin{aligned}[\![\boldsymbol{\mu}_{\boldsymbol{\gamma}}]\!]_{n-1} = {} & [\![\boldsymbol{\mu}_{\boldsymbol{\alpha}}]\!]_{k-1} * [\![\boldsymbol{\mu}_{\boldsymbol{\beta}}]\!]_{k-1} + [\![\boldsymbol{\mu}_{\boldsymbol{\alpha}}]\!]_{k-1} * [\![\boldsymbol{\lambda}_{\boldsymbol{\beta}}]\!]_{n-k} \\ & + [\![\boldsymbol{\mu}_{\boldsymbol{\beta}}]\!]_{k-1} * [\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}]\!]_{n-k} + [\![\boldsymbol{\Gamma}_{\boldsymbol{\gamma}}]\!]_{n-1}.\end{aligned}$$
4. $P_1$ collects the whole sharing $[\![\boldsymbol{\mu}_{\boldsymbol{\gamma}}]\!]_{n-1}$ from all parties and reconstructs $\boldsymbol{\mu}_{\boldsymbol{\gamma}}$.

**Functionality 2:** $\mathcal{F}_{\mathsf{PrepInd}}$

1. **Preparing Random Packed Sharings**: $\mathcal{F}_{\mathsf{PrepInd}}$ receives the set of corrupted parties, denoted by $\mathcal{C}orr$. For each output wire $\alpha$ of input gates and multiplication gates:
   (a) $\mathcal{F}_{\mathsf{PrepInd}}$ receives from the adversary a set of shares $\{u_j\}_{j\in\mathcal{C}orr}$. $\mathcal{F}_{\mathsf{PrepInd}}$ samples a random value $\lambda_\alpha$ and computes a degree-$(n-k)$ packed Shamir sharing $[\![\lambda_\alpha \cdot \mathbf{1}]\!]_{n-k}$ such that for all $P_j \in \mathcal{C}orr$, the $j$-th share of $[\![\lambda_\alpha \cdot \mathbf{1}]\!]_{n-k}$ is $u_j$.
   (b) $\mathcal{F}_{\mathsf{PrepInd}}$ distributes the shares of $[\![\lambda_\alpha \cdot \mathbf{1}]\!]_{n-k}$ to honest parties.
2. **Preparing Packed Beaver Triples**: For each group of $k$ multiplication gates:
   (a) $\mathcal{F}_{\mathsf{PrepInd}}$ receives from the adversary a set of shares $\{(u_j^{(1)}, u_j^{(2)}, u_j^{(3)})\}_{j\in\mathcal{C}orr}$. $\mathcal{F}_{\mathsf{PrepInd}}$ samples two random vectors $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{F}^k$ and computes $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$. Then $\mathcal{F}_{\mathsf{PrepInd}}$ computes three degree-$(n-k)$ packed Shamir sharings $[\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, [\![\boldsymbol{c}]\!]_{n-k}$ such that for all $P_j \in \mathcal{C}orr$, the $j$-th share of $([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, [\![\boldsymbol{c}]\!]_{n-k})$ is $(u_j^{(1)}, u_j^{(2)}, u_j^{(3)})$.
   (b) $\mathcal{F}_{\mathsf{PrepInd}}$ distributes the shares of $([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, [\![\boldsymbol{c}]\!]_{n-k})$ to honest parties.
3. **Preparing Random Masked Sharings for Multiplication Gates**: For each group of $k$ multiplication gates:

(a) $\mathcal{F}_{\mathsf{PrepInd}}$ receives from the adversary a set of shares $\{(u_j^{(1)}, u_j^{(2)}, u_j^{(3)})\}_{j \in \mathcal{C}orr}$. $\mathcal{F}_{\mathsf{PrepInd}}$ sets $\boldsymbol{o}^{(1)} = \boldsymbol{o}^{(2)} = \boldsymbol{o}^{(3)} = \mathbf{0} \in \mathbb{F}^k$. Then $\mathcal{F}_{\mathsf{PrepInd}}$ samples three random degree-$(n-1)$ packed Shamir sharings $[\![\boldsymbol{o}^{(1)}]\!]_{n-1}, [\![\boldsymbol{o}^{(2)}]\!]_{n-1}, [\![\boldsymbol{o}^{(3)}]\!]_{n-1}$ such that for all $P_j \in \mathcal{C}orr$, the $j$-th share of $([\![\boldsymbol{o}^{(1)}]\!]_{n-1}, [\![\boldsymbol{o}^{(2)}]\!]_{n-1}, [\![\boldsymbol{o}^{(3)}]\!]_{n-1})$ is $(u_j^{(1)}, u_j^{(2)}, u_j^{(3)})$.

(b) $\mathcal{F}_{\mathsf{PrepInd}}$ distributes the shares of $([\![\boldsymbol{o}^{(1)}]\!]_{n-1}, [\![\boldsymbol{o}^{(2)}]\!]_{n-1}, [\![\boldsymbol{o}^{(3)}]\!]_{n-1})$ to honest parties.

4. **Preparing Random Masked Sharings for Input and Output Gates**: For each group of $k$ input gates or output gates, $\mathcal{F}_{\mathsf{PrepInd}}$ 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 above.

*Output Phase.* In the output layer, for each group of $k$ output gates that belong to some `Client`, let $\boldsymbol{\alpha}$ denote the input wires of these output gates. Recall that all parties receive a degree-$(n-1)$ packed Shamir sharing $[\![\boldsymbol{\lambda_\alpha}]\!]_{n-1}$ from $\mathcal{F}_{\mathsf{Prep}}$ in the preprocessing phase. By the invariant, $P_1$ learns $\boldsymbol{\mu_\alpha} = \boldsymbol{v_\alpha} - \boldsymbol{\lambda_\alpha}$. Note that $\boldsymbol{v_\alpha}$ are the output values of `Client`. Therefore, all parties send their shares of $[\![\boldsymbol{\lambda_\alpha}]\!]_{n-1}$ to `Client` and $P_1$ sends $\boldsymbol{\mu_\alpha}$ to `Client`. In this way, `Client` can reconstruct the result $\boldsymbol{v_\alpha}$. The communication complexity per batch of $k$ output gates is $n+k$ elements.

*Online Protocol.* Now we are ready to present the online protocol. The description of the protocol $\Pi_{\mathsf{Online}}$ appears in Protocol 3.

**Protocol 3: $\Pi_{\mathsf{Online}}$**

1. **Preprocessing Phase**: All parties invoke $\mathcal{F}_{\mathsf{Prep}}$ to receive correlated randomness that will be used in the online phase.
2. **Input Phase**: In the input layer, for each group of $k$ input gates that belong to some `Client`, let $\boldsymbol{\alpha}$ denote the output wires of these input gates. All parties and `Client` invoke $\Pi_{\mathsf{Input}}$. At the end of the protocol, $P_1$ learns $\boldsymbol{\mu_\alpha} = \boldsymbol{v_\alpha} - \boldsymbol{\lambda_\alpha}$, where $\boldsymbol{v_\alpha}$ are the input values of `Client`, and $\boldsymbol{\lambda_\alpha}$ are the random values associated with the batch of wires $\boldsymbol{\alpha}$ generated by $\mathcal{F}_{\mathsf{Prep}}$.
3. **Computation Phase**: All parties maintain the invariant that for each wire $\alpha$, $P_1$ learns $\mu_\alpha = v_\alpha - \lambda_\alpha$, where $v_\alpha$ is the real value associated with the wire $\alpha$, and $\lambda_\alpha$ is a random value associated with $\alpha$ generated by $\mathcal{F}_{\mathsf{Prep}}$. The circuit is evaluated layer by layer. Assume that the invariant holds for wires in previous layers. Consider gates in the current layer.
   For each addition gate with input wires $\alpha, \beta$ and output wire $\gamma$, $P_1$ locally compute $\mu_\gamma = \mu_\alpha + \mu_\beta$.
   For each group of $k$ multiplication gates with input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$ and output wires $\boldsymbol{\gamma}$, all parties invoke $\Pi_{\mathsf{Mult}}$. At the end of the protocol, $P_1$ learns $\boldsymbol{\mu_\gamma}$.
4. **Output Phase**: For each group of $k$ output gates that belong to some `Client`, let $\boldsymbol{\alpha}$ denote the input wires of these output gates. Recall that all parties receive $[\![\boldsymbol{\lambda_\alpha}]\!]_{n-1}$ from $\mathcal{F}_{\mathsf{Prep}}$, and by the invariant, $P_1$ learns $\boldsymbol{\mu_\alpha} = \boldsymbol{v_\alpha} - \boldsymbol{\lambda_\alpha}$. All parties send their shares of $[\![\boldsymbol{\lambda_\alpha}]\!]_{n-1}$ to `Client`, and $P_1$ sends $\boldsymbol{\mu_\alpha}$ to `Client`. Then `Client` reconstructs $\boldsymbol{\lambda_\alpha}$ and computes $\boldsymbol{v_\alpha} = \boldsymbol{\lambda_\alpha} + \boldsymbol{\mu_\alpha}$.

**Functionality 3: $\mathcal{F}_{\mathsf{Main}}$**

1. $\mathcal{F}_{\mathsf{Main}}$ receives the input from all clients. Let $x$ denote the input and $C$ denote the circuit.
2. $\mathcal{F}_{\mathsf{Main}}$ computes $C(x)$ and distributes the output to all clients.

The online communication complexity of $\Pi_{\mathsf{Online}}$ is $3|C| \cdot n/k + O(\mathsf{Depth} \cdot n)$ field elements, where Depth is the circuit depth. The term $O(\mathsf{Depth} \cdot n)$ is because all parties need to communicate at least $3n$ elements in each layer even if there is only a single multiplication gate. Recall that $k = (n+3)/4$. The online communication complexity of TurboPack is 12 elements per gate among all parties.

The ideal functionality $\mathcal{F}_{\mathsf{Main}}$ appears in Functionality 3. We have the following lemma.

**Lemma 1.** *Protocol $\Pi_{\mathsf{Online}}$ securely computes $\mathcal{F}_{\mathsf{Main}}$ in the $\mathcal{F}_{\mathsf{Prep}}$-hybrid model against a semi-honest adversary who controls $t$ out of $n = 2t + 1$ parties and corrupts up to $\mathsf{c}$ of the clients.*

The proof of Lemma 1 can be found in Section B.1.

## 4.3 Instantiating Circuit-Dependent Preprocessing

In this part, we show how to realize $\mathcal{F}_{\mathsf{Prep}}$. Recall that in $\mathcal{F}_{\mathsf{Prep}}$, we need to prepare degree-$(n-k)$ packed Shamir sharings for the random values $\{\lambda_\alpha\}_\alpha$ associated with the wires in the circuit. We also need to prepare packed Beaver triples, which are the degree-$(n-1)$ packed Shamir sharings for $\{\boldsymbol{\Gamma}_\gamma\}_\gamma$. We refer the readers to Section 2 for an overview of our construction.

*Functionality for the Circuit-Independent Preprocessing Phase.* We first give the ideal functionality $\mathcal{F}_{\mathsf{PrepInd}}$ that prepares correlated randomness for the circuit-dependent preprocessing phase. The functionality will take as input the number of gates in the circuit $C$ without the structure of $C$.

**Protocol 4: $\Pi_{\mathsf{Prep}}$**

1. **Circuit-Independent Preprocessing Phase**: All parties invoke $\mathcal{F}_{\mathsf{PrepInd}}$ to receive correlated randomness.
2. **Computing a Random Sharing for Each Wire**: For each output wire $\alpha$ of input gates and multiplication gates, all parties receive $[\![\lambda_\alpha \cdot \mathbf{1}]\!]_{n-k}$ from $\mathcal{F}_{\mathsf{PrepInd}}$. All parties follow Step 1 of $\mathcal{F}_{\mathsf{Prep}}$ and compute $[\![\lambda_\alpha \cdot \mathbf{1}]\!]_{n-k}$ for each wire $\alpha$ in the circuit $C$.
3. **Preparing Degree-$(n-k)$ Packed Shamir Sharings**: For each group of multiplication gates with input wires $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_k)$, $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_k)$, recall that all parties have computed $\{[\![\lambda_{\alpha_i} \cdot \mathbf{1}]\!]\}_{i=1}^k$ and $\{[\![\lambda_{\beta_i} \cdot \mathbf{1}]\!]\}_{i=1}^k$ in the last step. Let $\boldsymbol{e}_i \in \mathbb{F}^k$ be the $i$-th unit vector, i.e., all entries of $\boldsymbol{e}_i$ are $0$ except the $i$-th entry is $1$. All parties locally compute $[\![\boldsymbol{\lambda_\alpha}]\!]_{n-1} = \sum_{i=1}^k \boldsymbol{e}_i * [\![\lambda_{\alpha_i} \cdot \mathbf{1}]\!]_{n-k}$ and $[\![\boldsymbol{\lambda_\beta}]\!]_{n-1} = \sum_{i=1}^k \boldsymbol{e}_i * [\![\lambda_{\beta_i} \cdot \mathbf{1}]\!]_{n-k}$. All parties use $([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{o}^{(1)}]\!]_{n-1})$ to reduce the degree of $[\![\boldsymbol{\lambda_\alpha}]\!]_{n-1}$. Here $[\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{o}^{(1)}]\!]_{n-1}$ are prepared in $\mathcal{F}_{\mathsf{PrepInd}}$.
   (a) All parties locally compute $[\![\boldsymbol{\lambda_\alpha} + \boldsymbol{a}]\!]_{n-1} = [\![\boldsymbol{\lambda_\alpha}]\!]_{n-1} + [\![\boldsymbol{a}]\!]_{n-k} + [\![\boldsymbol{o}^{(1)}]\!]_{n-1}$.
   (b) $P_1$ collects the whole sharing $[\![\boldsymbol{\lambda_\alpha} + \boldsymbol{a}]\!]_{n-1}$ and reconstructs the secret $\boldsymbol{d} = \boldsymbol{\lambda_\alpha} + \boldsymbol{a}$. Then $P_1$ computes the degree-$(k-1)$ packed Shamir sharing $[\![\boldsymbol{d}]\!]_{k-1}$ and distributes the shares to other parties.
   (c) All parties locally compute $[\![\boldsymbol{\lambda_\alpha}]\!]_{n-k} = [\![\boldsymbol{d}]\!]_{k-1} - [\![\boldsymbol{a}]\!]_{n-k}$.
   Similarly, all parties use $([\![\boldsymbol{b}]\!]_{n-k}, [\![\boldsymbol{o}^{(2)}]\!]_{n-1})$ to reduce the degree of $[\![\boldsymbol{\lambda_\beta}]\!]_{n-1}$. Here $[\![\boldsymbol{b}]\!]_{n-k}, [\![\boldsymbol{o}^{(2)}]\!]_{n-1}$ are prepared in $\mathcal{F}_{\mathsf{PrepInd}}$.
4. **Preparing Degree-$(n-1)$ Packed Shamir Sharings**: For each group of input gates, let $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_k)$ be the output wires of these gates. Recall that all parties have computed $\{[\![\lambda_{\alpha_i} \cdot \mathbf{1}]\!]\}_{i=1}^k$. Let $[\![\boldsymbol{o}]\!]_{n-1}$ be the random degree-$(n-1)$ packed Shamir sharing of $\mathbf{0} \in \mathbb{F}^k$ prepared in $\mathcal{F}_{\mathsf{PrepInd}}$. All parties locally compute $[\![\boldsymbol{\lambda_\alpha}]\!]_{n-1} := \sum_{i=1}^k \boldsymbol{e}_i * [\![\lambda_{\alpha_i} \cdot \mathbf{1}]\!]_{n-k} + [\![\boldsymbol{o}]\!]_{n-1}$.
   The same step is done for the input wires of each group of output gates.
5. **Preparing Packed Shamir Sharings for $\boldsymbol{\Gamma}_\gamma$**: For a group of $k$ multiplication gates with input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$ and output wires $\boldsymbol{\gamma}$, all parties have computed $[\![\boldsymbol{\lambda_\alpha}]\!]_{n-k}$ and $[\![\boldsymbol{\lambda_\beta}]\!]_{n-k}$ in the last step, which are in the forms $[\![\boldsymbol{\lambda_\alpha}]\!]_{n-k} = [\![\boldsymbol{d}_1]\!]_{k-1} - [\![\boldsymbol{a}]\!]_{n-k}$ and $[\![\boldsymbol{\lambda_\beta}]\!]_{n-k} = [\![\boldsymbol{d}_2]\!]_{k-1} - [\![\boldsymbol{b}]\!]_{n-k}$. Recall that all parties have computed $\{[\![\lambda_{\gamma_i} \cdot \mathbf{1}]\!]_{n-k}\}_{i=1}^k$. Also recall that all parties receive $[\![\boldsymbol{o}^{(3)}]\!]_{n-1}$ from $\mathcal{F}_{\mathsf{PrepInd}}$. All parties locally compute

$$[\![\boldsymbol{\Gamma}_\gamma]\!]_{n-1} = [\![\boldsymbol{d}_1]\!]_{k-1} * [\![\boldsymbol{d}_2]\!]_{k-1} - [\![\boldsymbol{d}_1]\!]_{k-1} * [\![\boldsymbol{b}]\!]_{n-k} - [\![\boldsymbol{d}_2]\!]_{k-1} * [\![\boldsymbol{a}]\!]_{n-k}$$
$$+ [\![\boldsymbol{c}]\!]_{n-k} - (\textstyle\sum_{i=1}^k \boldsymbol{e}_i * [\![\lambda_{\gamma_i} \cdot \mathbf{1}]\!]_{n-k}) + [\![\boldsymbol{o}^{(3)}]\!]_{n-1}.$$

*Protocol for the Circuit-Dependent Preprocessing Phase.* The description of the protocol $\Pi_{\mathsf{Prep}}$ appears in Protocol 4. The communication complexity of $\Pi_{\mathsf{Prep}}$ is $4|C| \cdot n/k$ field elements. Recall that $k = (n+3)/4$. The communication complexity of $\Pi_{\mathsf{Prep}}$ is 16 elements per gate among all parties.

**Lemma 2.** *Protocol $\Pi_{\mathsf{Prep}}$ securely computes $\mathcal{F}_{\mathsf{Prep}}$ in the $\mathcal{F}_{\mathsf{PrepInd}}$-hybrid model against a semi-honest adversary who controls $t$ out of $n = 2t+1$ parties.*

The proof of Lemma 2 can be found in Section B.2.

### 4.4 Instantiating Circuit-Independent Preprocessing

In this part, we discuss how to realize $\mathcal{F}_{\mathsf{PrepInd}}$. The task can be divided into two parts: (1) preparing random sharings and (2) computing multiplications.

To prepare random sharings, we follow the technique in [DN07] as described in Procedure 5.

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

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

3. All parties locally compute $(\boldsymbol{R}^{(1)}, \dots, \boldsymbol{R}^{(t+1)})^{\mathrm{T}} = \boldsymbol{M}(\boldsymbol{S}^{(1)}, \dots, \boldsymbol{S}^{(n)})^{\mathrm{T}}$. and output $(\boldsymbol{R}^{(1)}, \dots, \boldsymbol{R}^{(t+1)})$.

To prepare a packed Beaver triple, we first prepare $k$ Beaver triples by using Shamir secret sharing schemes. These $k$ Beaver triples are then transformed to a single packed Beaver triple. We refer the readers to Section 2 for an overview of our construction.

*Protocol for* $\mathcal{F}_{\mathsf{PrepInd}}$. TURBOPACK uses the ideal functionality $\mathcal{F}_{\mathsf{SingleMult}}$ described in Functionality 4 below. The protocol $\Pi_{\mathsf{PrepInd}}$ is described in Protocol 6.

**Functionality 4:** $\mathcal{F}_{\mathsf{SingleMult}}$

1. $\mathcal{F}_{\mathsf{SingleMult}}$ receives the secret position $i$ from all parties. Let $[\![x|_i]\!]_t, [\![y|_i]\!]_t$ denote the input sharings. $\mathcal{F}_{\mathsf{SingleMult}}$ receives from honest parties their shares of $[\![x|_i]\!]_t, [\![y|_i]\!]_t$. Then $\mathcal{F}_{\mathsf{SingleMult}}$ reconstructs the secrets $x, y$. $\mathcal{F}_{\mathsf{SingleMult}}$ further computes the shares of $[\![x|_i]\!]_t, [\![y|_i]\!]_t$ held by corrupted parties, and sends these shares to the adversary.
2. $\mathcal{F}_{\mathsf{SingleMult}}$ receives from the adversary a set of shares $\{z_i\}_{i \in \mathcal{C}orr}$.
3. $\mathcal{F}_{\mathsf{SingleMult}}$ computes $x \cdot y$. Based on the secret $z := x \cdot y$ and the $t$ shares $\{z_i\}_{i \in \mathcal{C}orr}$, $\mathcal{F}_{\mathsf{SingleMult}}$ reconstructs the whole sharing $[\![z|_i]\!]_t$ and distributes the shares of $[\![z|_i]\!]_t$ to honest parties.

**Protocol 6:** $\Pi_{\mathsf{PrepInd}}$

1. **Preparing Random Packed Sharings**: Let $N_1$ be the number of input gates and output gates. Let $\Sigma_1$ be the secret sharing scheme corresponding to $[\![r \cdot \mathbf{1}]\!]_{n-k}$. All parties invoke $N_1/(t+1)$ times of $\pi_{\mathsf{RandSh}}(\Sigma_1)$ to prepare $N_1$ random sharings in the form of $[\![r \cdot \mathbf{1}]\!]_{n-k}$.
2. **Preparing Packed Beaver Triples**: Let $N_2$ denote the number of groups of multiplication gates. For all $i \in \{1, 2, \dots, k\}$, let $\Sigma_{2,i}$ be the secret sharing scheme corresponding to $[\![r|_i]\!]_t$. All parties invoke $2N_2/(t+1)$ times of $\pi_{\mathsf{RandSh}}(\Sigma_{2,i})$ to prepare $2N_2$ random sharings in the form of $[\![r|_i]\!]_t$.
   (a) For each group of multiplication gates, let $\{[\![a_i|_i]\!]_t, [\![b_i|_i]\!]_t\}_{i=1}^k$ be the unused random sharings.
   (b) For all $i \in \{1, 2, \dots, k\}$, all parties invoke $\mathcal{F}_{\mathsf{SingleMult}}$ on $(i, [\![a_i|_i]\!]_t, [\![b_i|_i]\!]_t)$ and receive $[\![c_i|_i]\!]_t$.
   (c) Let $\boldsymbol{e}_i \in \mathbb{F}^k$ be the $i$-th unit vector, i.e., all entries of $\boldsymbol{e}_i$ are $0$ except the $i$-th entry is $1$. All parties locally transform $\boldsymbol{e}_i$ to the degree-$(k-1)$ packed Shamir sharing $[\![\boldsymbol{e}_i]\!]_{k-1}$. Then, all parties locally compute

$$[\![\boldsymbol{a}]\!]_{n-k} = \sum_{i=1}^{k} [\![\boldsymbol{e}_i]\!]_{k-1} * [\![a_i|_i]\!]_t,$$

$$[\![\boldsymbol{b}]\!]_{n-k} = \sum_{i=1}^{k} [\![\boldsymbol{e}_i]\!]_{k-1} * [\![b_i|_i]\!]_t,$$

$$[\![\boldsymbol{c}]\!]_{n-k} = \sum_{i=1}^{k} [\![\boldsymbol{e}_i]\!]_{k-1} * [\![c_i|_i]\!]_t.$$

3. **Preparing Random Masked Sharings for Multiplication Gates**: Let $\Sigma_3$ be the secret sharing scheme corresponding to $[\![\mathbf{0}]\!]_{n-1}$, where $\mathbf{0} = (0, \dots, 0) \in \mathbb{F}^k$. All parties invoke $3N_2/(t+1)$ times of $\pi_{\mathsf{RandSh}}(\Sigma_3)$ to prepare $3N_2$ random sharings in the form of $[\![\mathbf{0}]\!]_{n-1}$.
4. **Preparing Random Masked Sharings for Input and Output Gates**: Let $N_3$ be the number of groups of input gates and output gates. All parties invoke $N_3/(t+1)$ times of $\pi_{\mathsf{RandSh}}(\Sigma_3)$ to prepare $N_3$ random sharings in the form of $[\![\mathbf{0}]\!]_{n-1}$.

We analyse the communication complexity of $\Pi_{\mathsf{PrepInd}}$:

– Step 1, Step 3, Step 4 require to prepare different kinds of random sharings. The procedure $\pi_{\mathsf{RandSh}}(\Sigma)$ outputs $t+1$ random $\Sigma$-sharings at the cost of communicating $n$ $\Sigma$-sharings. Thus, Step 1 requires to communicate $2N_1 \cdot n$ elements. Step 3 requires to communicate $6N_2 \cdot n$ elements. Step 4 requires to communicate $2N_3 \cdot n$ elements.
– Step 2 requires to first prepare random degree-$t$ Shamir sharings, which is $4N_2 \cdot k \cdot n$ elements. Then for each group of $k$ multiplication gates, all parties need to invoke $\mathcal{F}_{\mathsf{SingleMult}}$ $k$ times. When using [GLO+21] to instantiate $\mathcal{F}_{\mathsf{SingleMult}}$, which requires $4$ elements of communication, the total cost of $\mathcal{F}_{\mathsf{SingleMult}}$ is $4N_2 \cdot k \cdot n$ elements. Thus, Step 2 requires to communicate $8N_2 \cdot k \cdot n$ elements.

Note that $N_1$ is of size $|C|$, $N_2$ is of size $|C|/k$, and $N_3$ is small compared with the circuit size. Thus, the communication complexity of $\Pi_{\mathsf{PrepInd}}$ is $10|C| \cdot n + 24|C|$ elements among all parties. The amortized communication complexity per gate is $10n + 24$ elements.

**Lemma 3.** *Protocol $\Pi_{\mathsf{PrepInd}}$ securely computes $\mathcal{F}_{\mathsf{PrepInd}}$ in the $\mathcal{F}_{\mathsf{SingleMult}}$-hybrid model against a semi-honest adversary who controls $t$ out of $n = 2t + 1$ parties.*

The proof of Lemma 3 can be found in Section B.3.

Combining our protocols $\Pi_{\mathsf{PrepInd}}$, $\Pi_{\mathsf{Prep}}$, and $\Pi_{\mathsf{Online}}$ and instantiating $\mathcal{F}_{\mathsf{SingleMult}}$ by [GLO+21], we obtain the following theorem.

**Theorem 1.** *In the client-server model, let $\mathsf{c}$ denote the number of clients, $n$ denote the number of parties (servers), and $t = (n-1)/2$ denote the number of corrupted parties (servers). Let $\mathbb{F}$ be a finite field of size $|\mathbb{F}| \geq 2n$. For an arithmetic circuit $C$ over $\mathbb{F}$, there exists an information-theoretic MPC protocol which securely computes the arithmetic circuit $C$ in the presence of a semi-honest adversary controlling up to $\mathsf{c}$ clients and $t$ parties. The splitting communication complexity per gate is (1) $10n+24$ elements per gate in the circuit-independent preprocessing phase, (2) $16$ elements per gate in the circuit-dependent preprocessing phase, and (3) $12$ elements per gate in the online phase. (Terms that are independent of or sub-linear in the circuit size are omitted as they only add cost $o(1)$ per gate.)*

In Section C, we show an optimization of TurboPack which allows us to further reduce the communication complexity by a factor of 2 in the circuit-dependent preprocessing phase.

## 5 Performance Study

| Width | Prep. | Number of parties | | | | | |
|---|---|---|---|---|---|---|---|
| | | 5 | | 13 | | 21 | |
| | | TP (s) | Factor (×) | TP (s) | Factor (×) | TP (s) | Factor (×) |
| 100 | CD | 0.16 / 0.45 | **8.70 / 1.09** | 0.24 / 0.40 | **3.54 / 0.88** | 0.51 / 0.61 | **4.23 / 1.14** |
| | CI | 0.07 / 0.53 | **4.94 / 1.28** | 0.16 / 0.48 | **3.33 / 1.01** | 0.38 / 0.74 | **3.41 / 1.36** |
| 1k | CD | 0.40 / 0.35 | **5.75 / 0.72** | 1.24 / 0.74 | **4.72 / 0.99** | 3.20 / 0.74 | **5.25 / 0.52** |
| | CI | 0.29 / 0.46 | **5.03 / 0.93** | 0.96 / 1.01 | **4.08 / 1.31** | 2.60 / 1.35 | **4.64 / 0.91** |
| 10k | CD | 2.97 / 0.94 | **5.13 / 1.08** | 11.39 / 1.68 | **5.24 / 0.61** | 30.88 / 3.54 | **5.68 / 0.42** |
| | CI | 2.30 / 1.62 | **5.13 / 1.61** | 9.14 / 3.93 | **4.90 / 1.29** | 25.36 / 9.06 | **5.06 / 1.03** |
| 100k | CD | 33.51 / 4.81 | **6.07 / 0.97** | 113.39 / 13.28 | **5.40 / 0.52** | 306.50 / 30.85 | **5.78 / 0.38** |
| | CI | 26.45 / 11.87 | **6.04 / 1.94** | 90.76 / 35.91 | **4.99 / 1.27** | 252.05 / 85.30 | **5.17 / 1.00** |

| Width | Prep. | Number of parties | | | | | |
|---|---|---|---|---|---|---|---|
| | | 29 | | 37 | | 45 | |
| | | TP (s) | Factor (×) | TP (s) | Factor (×) | TP (s) | Factor (×) |
| 100 | CD | 0.93 / 0.57 | **4.56 / 0.84** | 1.38 / 0.75 | **4.64 / 0.94** | 2.34 / 0.65 | **4.91 / 0.63** |
| | CI | 0.73 / 0.77 | **3.89 / 1.11** | 1.16 / 0.97 | **4.18 / 1.18** | 2.01 / 0.98 | **4.37 / 0.93** |
| 1k | CD | 6.66 / 1.15 | **5.87 / 0.51** | 11.73 / 1.56 | **6.51 / 0.39** | 19.74 / 1.81 | **7.16 / 0.27** |
| | CI | 5.49 / 2.31 | **5.13 / 1.01** | 9.91 / 3.39 | **5.82 / 0.83** | 16.96 / 4.59 | **6.36 / 0.66** |
| 10k | CD | 65.18 / 6.41 | **6.42 / 0.33** | 119.13 / 10.26 | **7.05 / 0.27** | 198.91 / 15.09 | **7.64 / 0.23** |
| | CI | 54.46 / 17.13 | **5.69 / 0.86** | 101.28 / 28.11 | **6.26 / 0.74** | 171.79 / 42.21 | **6.83 / 0.63** |
| 100k | CD | 645.41 / 59.45 | **6.25 / 0.31** | 1183.21 / 97.88 | **6.99 / 0.26** | 1990.68 / 147.61 | **7.62 / 0.22** |
| | CI | 539.02 / 165.84 | **5.51 / 0.85** | 1007.70 / 273.39 | **6.22 / 0.72** | 1719.31 / 418.98 | **6.80 / 0.63** |

Table 1: Running times and comparison of TurboPack with DN07, in a LAN setting with 1ms latency and 1Gbps bandwidth, for a circuit of depth 10 and varying width and number of parties. The TP columns refer to the running time of TurboPack in seconds. The “factor” columns refer to the ratio between the running time of TurboPack and DN07. The format of the timings and ratios is “Offline / Online”. In the CD. Prep case our offline and online phases are ①+② and ③, while in the CI. Prep scenario these are ① and ②+③.

In this section we study the performance of TurboPack and compare it to existing work in the context of maximal adversary honest majority MPC, where $n = 2t + 1$. As a baseline for comparison, we choose an optimized version of DN07 [DN07], using ideas from [GSZ20] that reduces online communication by setting some shares to be zero, together with the observations that the messages sent by some of the parties are known already in a circuit-dependent offline phase, and hence the online phase can be made lighter. The details of this protocol can be found in Section E in the Appendix. This is the protocol with the most optimal *online* communication complexity, as it only uses $1$ element per party per multiplication gate in the online phase.[10][11] We have fully implemented the *passive* version of TurboPack, and in the same framework we implemented the optimized DN07 for a fair comparison. In this section we present and discuss the experimental results we have obtained.

### 5.1 Communication Complexity

| Type of prep. | Phase | Ours | DN07 | Ours/DN07 |
|---|---|---|---|---|
| CD prep. model | Offline (①+②) | $10n + 32$ | $4.5n$ | $2.23 + 7.12/n$ |
| | Online (③) | $12$ | $1n$ | $12/n$ |
| CI prep. model | Offline (①) | $10n + 24$ | $4n$ | $2.5 + 6/n$ |
| | Online (②+③) | $20$ | $1.5n$ | $13.34/n$ |
| Total (①+②+③) | | $10n + 44$ | $5.5n$ | $1.82 + 8/n$ |

Table 2: Communication complexity per multiplication gate compared to the optimized DN07 protocol. CD prep. refers to the setting when the offline phase is allowed to depend on the function, while CI prep. is when the offline phase is both input and function-independent. Either case the offline phase of DN07 remains the same.

Table 2 summarizes the communication complexity per multiplication gate of TurboPack[12] and compares it with that of the optimized version of DN07 from Section E in the Appendix.[13] For our protocol we use the optimized version from Section C in the Appendix. The complexities can be found in Theorem 2. For the purpose of evaluating offline and online communication separately, we consider two models as discussed above: circuit-dependent (CD) and circuit-independent (CI) preprocessing. Part of TurboPack (phase ②) can be run while knowing the circuit but not the inputs, so our online phase in the CD prep. model is better than in the CI prep. model. Such optimization is not possible in DN07.

We observe that the online phase of TurboPack, regardless of whether we are in the CD or CI model, outperforms that of DN07 asymptotically (in $n$) since the communication complexity of our online phase is independent of the number of parties, while that of DN07 grows linearly with $n$. Furthermore, concrete constants are small enough so that improvements can be seen for small values of $n$: in the CD case our online phase is better than that of DN07 for $n \geq 12$, and in the CI case this happens for $n \geq 19$, with the gap widening as $n$ grows. For example, for $n = 48$ our online phase requires $4\times$ less communication than that of DN07, and for $n = 60$ this improves to $5\times$, in the CD prep. model, Regarding total communication complexity, TurboPack performs around $1.8\times$ worse than DN07, asymptotically. This is not a large factor, considering the gains in the online phase.

### 5.2 Implementation setup

TurboPack is end-to-end implemented from scratch in C++, without any dependencies besides the C++ standard library. The source code is open and can be found in https://github.com/

[10] The protocol with the best *total* communication complexity is ATLAS [GLO+21], but since our goal is to optimize the online phase, we compare to the protocol with the most optimal online phase. Asymptotically, the best online phase is in [GPS22], but as we have argued in the introduction in practice it is DN07 [DN07].

[11] We remark that one of the protocols in [DE21b] achieves the same online communication complexity, but as we argue in Section E in the Appendix, the optimized DN07 protocol is simpler and more efficient.

[12] We do not consider the optimization from Section C, which improves phase ② by a factor of two.

[13] We remark that TurboPack performs better than that of DN07 for input and output gates, but we assume the number of such gates to be much smaller than the number of multiplication gates, and hence we ignore this.

deescuderoo/turbopack.git. For an effective comparison, we also implement the optimized DN07 protocol [DN07], and use it as the baseline. The implementation of TurboPack does not include the optimization discussed in Section C, and for the instantiation of $\mathcal{F}_{\mathsf{SingleMult}}$ we use DN07, not the optimal ATLAS. As a result, the communication complexity of phase ① is $11.5n + 24$, that of ② is $16$, and that of ③ is $12$, where, as described in Section 1, phase ① is circuit and input-independent, phase ② is input-independent but circuit dependent, and phase ③ is circuit and input-dependent. Recall that the communication complexity of (optimized) DN07 is $4n$ in the circuit-independent offline phase, $0.5n$ in the circuit-dependent offline phase, and $1n$ elements in the online phase.

For the experiments we use the 61-bit Mersenne field for computation, and we deploy TurboPack in a single machine. Each party is its own process, and we use interprocess communication for emulating actual communication. To simulate real distributed environments, we make use of the linux `tc` command from the network emulation package `netem`[14] to modify bandwidth and latency. We use a bandwidth of 1Gbps, and we use 1ms of latency, which aims at emulating a LAN network. Other networking settings are considered in Section F in the Appendix. We use an AWS `c5.metal` instance. Since each party runs as a single process, we chose to use a machine with a good amount of cores (96) to support the amount of parties we consider without adding too much overhead due to context switching and similar OS-related issues. This creates an experimental setup that is easier to replicate for future works. All of our experiments report the average of five runs.

We acknowledge that running parties locally as a process in a single machine has the drawback of decreasing the computational power per party when the number of parties increases, since these processes will compete for the resources in the machine and there is a non-negligible overhead in context switching (which is particularly relevant when the number of parties exceed the number of available cores). As a result, for large number of parties our results may not reflect the exact running times that one would get with TurboPack in an actual distributed scenario. However, we argue that, for the purpose of determining the improvement factor of TurboPack with respect to our baseline DN07, this approach should be sufficient. Indeed, since both protocols are run in the exact same setting, so both protocols get the same per-party computational slowdown, and hence the improvement ratio of TurboPack with respect to DN07—which is ultimately what we want to measure—should remain faithful.

### 5.3 Performance comparison with respect to DN07

Here we study what the improvement of TurboPack with respect to DN07 is. To this end, we report the running time of TurboPack and the improvement factor relative to DN07[15] in several settings, considering multiple circuits with different characteristics, and also varying the number of parties. For our experiments we fix the depth to be $10$, and focus on increasing the width of the circuit (i.e. the amount of multiplications per layer), and we consider a LAN setting with 1ms latency and 1Gbps bandwidth. We do this in both the **CD** model, meaning that the preprocessing is allowed to depend on the circuit, and also in the **CI** model, where the preprocessing does not depend on the circuit. The results are presented in Table 1. For the number of parties up to $29$ the results are averaged over five iterations. For $37$ and $45$, only one iteration is used due to long running times.

We first begin by analyzing the effect of the width in our improvement factor. Since our techniques enable the parties to pack $k = (n + 3)/4$ multiplications across the same layer into one, the improvements of TurboPack can only be seen when the width is above certain threshold. We see this in Table 1: for a small width of $100$ the online phase of TurboPack generally offers little-to-none improvement with respect to that of DN07. However, as the width increases, we start seeing noticeable improvements.

Now, the number of parties also plays an important role in how much better our online phase is with respect to that of DN07, since the number of parties dictate how many multiplication gates can be packed. In the CD prep. model, the improvement factor of our online phase with respect to that of DN07 is $12/1n = 12/n$, and in the CI prep. model it is $28/1.5n = 18.6/n$ (recall we are not considering the optimizations to TurboPack discussed in Section C in the Appendix). This means that, for large values of $n$ our online phase will outperform that of DN07, and indeed, we observe this behavior experimentally.

[14] https://wiki.linuxfoundation.org/networking/netem

[15] We do not include the running time of DN07 since it can be derived from our running time and the improvement factor.

For example, for $n = 5$ we see generally little improvement, but when $n$ grows to values like 29 or even 37 and 45, the online phase of TurboPack outperforms that of DN07 by factors that range from $3\times$ to $5\times$, depending on the width, in the CD prep. model. In the CI prep. model, naturally, the improvement factor of our online phase is smaller, but we still see improvements for large values of $n$ nevertheless. Furthermore, naturally, when even larger number of parties are considered are work will offer even better improvement factors. One thing to note is that in many cases the experimental improvement factor does not match exactly the expected communication improvement factor. For example, for $n = 21$, in the CD prep. model the communication ratio is $12/21 \approx 0.57$, which is closely attained for a width of 1k, but then for a width of 10k this experimental factor drops to 0.42, and it drops even more for width 100k We believe this may be attributed to communication playing a much larger role when the number of parties grow. For $n = 45$ on the other hand, the communication ratio of the online phase in the CD prep. model is $12/45 \approx 0.27$, which matches the factor found in Table 1 for this setting, when the width is 1k, but then again it drops as the width increases.

The offline phase in our implementation takes $11.5n + 40$ elements per multiplication gate in the CD prep. model, and $11.5n + 24$ in the CI prep. model. In contrast, DN07 takes $4.5n$ and $4n$ elements, respectively. However, experimentally we find that the improvement factor in runtimes of TurboPack with respect to DN07 in the offline phase is slightly larger than expected. For example, for $n = 21$, in the CD prep. model the communication factor of the offline phase is 2.98. However, experimentally, these factors range from 4.23 up to 5.78. For the CI prep. model the communication factor when again $n = 21$ is 3.16, but once again we get larger runtime factors that go from 3.41 up to 5.17. When $n$ is larger, say $n = 45$, the factor in the CI and CD models are 2.75 and 3, respectively, but in our experiments we find these range from 4.37 to 6.80, and from 4.91 to 7.62, respectively. We believe that, even though communication plays a major role, this might be caused by the overhead in terms of computation that our techniques impose: the offline phase requires the parties to store much more data, sampling many more shares, $P_1$ has to pack and unpacked shared values, which are operations not needed in DN07.

In Section F in the Appendix we present and discuss more experimental results ran in other networking settings. One interesting thing we observe there is that, as the latency goes up, the improvement factor of TurboPack with respect to DN07 gets better, suggesting that indeed that the mild computation overhead of TurboPack may be a cause of slowdown, and higher latency give enough time for computation.

Finally, we also remark that our implementation for packed-secret sharing is very rudimentary: polynomial interpolation and evaluation are achieved by simple non-optimized matrix multiplications. We do not make use of any libraries for polynomial computation or linear algebra with the aim of maintaining a portable and self-contained implementation. We believe that using more efficient methods for manipulating polynomials (as discussed *e.g.* in `https://github.com/becgabri/packed-ss-template`), which is an operation needed extensively in TurboPack, may help improve the computation overhead of our protocol and hence achieve better performance ratios with respect to existing work.

## Acknowledgments

V. Goyal, Y. Song—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. Y. Song was also supported by a Cylab Presidential Fellowship.

This paper was prepared in part for information purposes by the Artificial Intelligence Research group 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. 2022 JP Morgan Chase & Co. All rights reserved.

## References

ACE$^{+}$21. Mark Abspoel, Ronald Cramer, Daniel Escudero, Ivan Damgård, and Chaoping Xing. Improved single-round secure multiplication using regenerating codes. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 222–244. Springer, 2021.

BBCG$^{+}$19. Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Zero-knowledge proofs on secret-shared data via fully linear pcps. In *Advances in Cryptology – CRYPTO 2019*, pages 67–97, Cham, 2019. Springer International Publishing.

BBG$^{+}$21. Fabrice Benhamouda, Elette Boyle, Niv Gilboa, Shai Halevi, Yuval Ishai, and Ariel Nof. Generalized pseudorandom secret sharing and efficient straggler-resilient secure computation. In Kobbi Nissim and Brent Waters, editors, *Theory of Cryptography*, pages 129–161, Cham, 2021. Springer International Publishing.

BDOZ11. Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 169–188. Springer, 2011.

Bea92. Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *Advances in Cryptology — CRYPTO '91*, pages 420–432, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

BENO19. Aner Ben-Efraim, Michael Nielsen, and Eran Omri. Turbospeedz: Double your online spdz! improving spdz using function dependent preprocessing. In *International Conference on Applied Cryptography and Network Security*, pages 530–549. Springer, 2019.

BGIN19. Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. Practical fully secure three-party computation via sublinear distributed zero-knowledge proofs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS 19, page 869?886, New York, NY, USA, 2019. Association for Computing Machinery.

BGIN20. Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. Efficient fully secure computation via distributed zero-knowledge proofs. In *Advances in Cryptology – ASIACRYPT 2020*, pages 244–276, Cham, 2020. Springer International Publishing.

BGJK21a. Gabrielle Beck, Aarushi Goel, Abhishek Jain, and Gabriel Kaptchuk. Order-c secure multiparty computation for highly repetitive circuits. In *Advances in Cryptology – EUROCRYPT 2021*, pages 663–693, Cham, 2021. Springer International Publishing.

BGJK21b. Gabrielle Beck, Aarushi Goel, Abhishek Jain, and Gabriel Kaptchuk. Order-c secure multiparty computation for highly repetitive circuits. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 663–693. Springer, 2021.

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

CGH$^{+}$18. Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority mpc for malicious adversaries. In *Annual International Cryptology Conference*, pages 34–64. Springer, 2018.

DE21a. Anders Dalskov and Daniel Escudero. Honest majority mpc with abort with minimal online communication. In *International Conference on Cryptology and Information Security in Latin America*, pages 453–472. Springer, 2021.

DE21b. Anders Dalskov and Daniel Escudero. Honest majority mpc with abort with minimal online communication. In *International Conference on Cryptology and Information Security in Latin America*, pages 453–472. Springer, 2021.

DIK10. Ivan Damgård, Yuval Ishai, and Mikkel Krøigaard. Perfectly secure multiparty computation and the computational overhead of cryptography. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 445–465. Springer, 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. In *European Symposium on Research in Computer Security*, pages 1–18. Springer, 2013.

DLN19. Ivan Damgård, Kasper Green Larsen, and Jesper Buus Nielsen. Communication lower bounds for statistically secure mpc, with or without preprocessing. In *Annual International Cryptology Conference*, pages 61–84. Springer, 2019.

DN07. Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In *Annual International Cryptology Conference*, pages 572–590. Springer, 2007.

DNPR16. Ivan Damgård, Jesper Buus Nielsen, Antigoni Polychroniadou, and Michael Raskin. On the communication required for unconditionally secure multiplication. In *Annual International Cryptology Conference*, pages 459–488. Springer, 2016.

DPSZ12a. Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Annual Cryptology Conference*, pages 643–662. Springer, 2012.

DPSZ12b. Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Advances in Cryptology–CRYPTO 2012*, pages 643–662. Springer, 2012.

ESV21. Daniel Escudero and Eduardo Soria-Vazquez. Efficient information-theoretic multi-party computation over non-commutative rings. In *Annual International Cryptology Conference*, pages 335–364. Springer, 2021.

FY92. Matthew Franklin and Moti Yung. Communication Complexity of Secure Computation (Extended Abstract). In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing*, STOC '92, page 699–710, New York, NY, USA, 1992. Association for Computing Machinery.

GIOZ17. Juan Garay, Yuval Ishai, Rafail Ostrovsky, and Vassilis Zikas. The price of low communication in secure multi-party computation. In *Advances in Cryptology – CRYPTO 2017*, pages 420–446, Cham, 2017. Springer International Publishing.

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. In *Annual Cryptology Conference*, pages 721–741. Springer, 2015.

GLO$^+$21. Vipul Goyal, Hanjun Li, Rafail Ostrovsky, Antigoni Polychroniadou, and Yifan Song. Atlas: Efficient and scalable mpc in the honest majority setting. In *Advances in Cryptology – CRYPTO 2021*, pages 244–274, Cham, 2021. Springer International Publishing.

GPS21. Vipul Goyal, Antigoni Polychroniadou, and Yifan Song. Unconditional communication-efficient mpc via hall's marriage theorem. In *Annual International Cryptology Conference*, pages 275–304. Springer, 2021.

GPS22. Vipul Goyal, Antigoni Polychroniadou, and Yifan Song. Sharing transformation and dishonest majority mpc with packed secret sharing. *Annual International Cryptology Conference*, 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. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 694–723. Springer, 2021.

GSZ20. Vipul Goyal, Yifan Song, and Chenzhi Zhu. Guaranteed output delivery comes free in honest majority mpc. In *Advances in Cryptology – CRYPTO 2020*, pages 618–646, Cham, 2020. Springer International Publishing.

LPSY19. Yehuda Lindell, Benny Pinkas, Nigel P. Smart, and Avishay Yanai. Efficient constant-round multi-party computation combining bmr and spdz. *Journal of Cryptology*, 32(3):1026–1069, 2019.

NST17. Jesper Buus Nielsen, Thomas Schneider, and Roberto Trifiletti. Constant round maliciously secure 2pc with function-independent preprocessing using lego. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

NV18. Peter Sebastian Nordholt and Meilof Veeningen. Minimising communication in honest-majority mpc by batchwise multiplication verification. In *Applied Cryptography and Network Security*, pages 321–339, Cham, 2018. Springer International Publishing.

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

WRK17a. Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 21–37, New York, NY, USA, 2017. Association for Computing Machinery.

WRK17b. Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 39–56, New York, NY, USA, 2017. Association for Computing Machinery.

## A The Model

### A.1 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 $\pi$ 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 $\pi$ 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.

### A.2 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.

### A.3 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 $\mathsf{c}$ 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 $\mathsf{c}$ 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.

## B Security Proofs of Our Semi-honest Protocol

### B.1 Proof of Lemma 1

**Lemma 1.** *Protocol $\Pi_{\mathsf{Online}}$ securely computes $\mathcal{F}_{\mathsf{Main}}$ in the $\mathcal{F}_{\mathsf{Prep}}$-hybrid model against a semi-honest adversary who controls $t$ out of $n = 2t + 1$ parties and corrupts up to $\mathsf{c}$ of the clients.*

*Proof.* Let $\mathcal{A}$ denote the adversary. 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 correctness of $\Pi_{\mathsf{Online}}$ follows from the description.

We now describe the construction of the simulator $\mathcal{S}$.

1. In the preprocessing phase, $\mathcal{S}$ emulates the ideal functionality $\mathcal{F}_{\mathsf{Prep}}$ and receives the shares of corrupted parties for each packed Shamir sharing. Note that $\mathcal{F}_{\mathsf{Prep}}$ does not need to send any message to corrupted parties.
2. In the input phase, for each group of $k$ input gates that belong to some `Client`, let $\boldsymbol{\alpha}$ denote the batch of output wires of these input gates.
   - If `Client` is honest, after receiving the shares of $[\![\boldsymbol{\lambda_\alpha}]\!]_{n-1}$ from all parties, $\mathcal{S}$ samples random values as $\boldsymbol{\mu_\alpha}$ and sends them to $P_1$.
   - If `Client` is corrupted, $\mathcal{S}$ samples random values as $\boldsymbol{\lambda_\alpha}$. Then based on the secrets $\boldsymbol{\lambda_\alpha}$ and the shares of corrupted parties, $\mathcal{S}$ randomly samples the whole sharing $[\![\boldsymbol{\lambda_\alpha}]\!]_{n-1}$ and sends the shares of $[\![\boldsymbol{\lambda_\alpha}]\!]_{n-1}$ held by honest parties to `Client`. From the inputs $\boldsymbol{v_\alpha}$ of `Client`, $\mathcal{S}$ computes $\boldsymbol{\mu_\alpha} = \boldsymbol{v_\alpha} - \boldsymbol{\lambda_\alpha}$.
3. In the computation phase, we will maintain the invariant that $\mu_\alpha$ for each wire $\alpha$ is known to $\mathcal{S}$. Note that this is true for wires in the first layer (the input layer).
   For each addition gate with input wires $\alpha, \beta$ and output wire $\gamma$, $\mathcal{S}$ honestly compute $\mu_\gamma = \mu_\alpha + \mu_\beta$.
   For each group of $k$ multiplication gates with input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$ and output wires $\boldsymbol{\gamma}$, $\mathcal{S}$ simulates $\Pi_{\mathsf{Mult}}$ as follows.
   - If $P_1$ is honest, $\mathcal{S}$ computes degree-$(k-1)$ packed Shamir sharings $[\![\boldsymbol{\mu_\alpha}]\!]_{k-1}$ and $[\![\boldsymbol{\mu_\beta}]\!]_{k-1}$ based on $\boldsymbol{\mu_\alpha}$ and $\boldsymbol{\mu_\beta}$, which are known to $\mathcal{S}$ according to the invariant. Then, $\mathcal{S}$ computes the shares of $[\![\boldsymbol{\mu_\gamma}]\!]_{n-1}$ of corrupted parties. $\mathcal{S}$ sets the shares of $[\![\boldsymbol{\mu_\gamma}]\!]_{n-1}$ of honest parties to be uniform elements. Finally, $\mathcal{S}$ reconstructs $\boldsymbol{\mu_\gamma}$ (which is a vector of $k$ random values).
   - If $P_1$ is corrupted, $\mathcal{S}$ receives from $P_1$ the shares of $[\![\boldsymbol{\mu_\alpha}]\!]_{k-1}$ and $[\![\boldsymbol{\mu_\beta}]\!]_{k-1}$ of honest parties. Then $\mathcal{S}$ recovers the whole sharings $[\![\boldsymbol{\mu_\alpha}]\!]_{k-1}$ and $[\![\boldsymbol{\mu_\beta}]\!]_{k-1}$, and learns the shares of corrupted parties. Now $\mathcal{S}$ can compute the shares of $[\![\boldsymbol{\mu_\gamma}]\!]_{n-1}$ of corrupted parties.
     $\mathcal{S}$ samples random elements as the shares of $[\![\boldsymbol{\mu_\gamma}]\!]_{n-1}$ of honest parties and sends them to $P_1$. Finally, $\mathcal{S}$ reconstructs the secret $\boldsymbol{\mu_\gamma}$ (which is a vector of $k$ random values).
4. In the output phase, for each group of $k$ output gates that belong to some `Client`, let $\boldsymbol{\alpha}$ denote the batch of input wires of these output gates.
   - If `Client` is honest, $\mathcal{S}$ does nothing.
   - If `Client` is corrupted, $\mathcal{S}$ sends the inputs of corrupted clients to $\mathcal{F}_{\mathsf{Main}}$ (since $\mathcal{S}$ can access to the inputs and random tapes of corrupted clients and corrupted parties). Then $\mathcal{S}$ receives the outputs $\boldsymbol{v_\alpha}$ of `Client` from $\mathcal{F}_{\mathsf{Main}}$. Recall that $\mathcal{S}$ knows $\boldsymbol{\mu_\alpha}$. $\mathcal{S}$ computes $\boldsymbol{\lambda_\alpha} = \boldsymbol{v_\alpha} - \boldsymbol{\mu_\alpha}$. Based on the secrets $\boldsymbol{\lambda_\alpha}$ and the shares of corrupted parties, $\mathcal{S}$ randomly samples the whole sharing $[\![\boldsymbol{\lambda_\alpha}]\!]_{n-1}$. Finally, $\mathcal{S}$ sends to `Client` the shares of $[\![\boldsymbol{\lambda_\alpha}]\!]_{n-1}$ of honest parties. If $P_1$ is honest, $\mathcal{S}$ also sends to `Client` $\boldsymbol{\mu_\alpha}$.

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

We show that $\mathcal{S}$ perfectly simulates the behaviors of honest parties. It is sufficient to focus on the places where honest parties and clients need to communicate with corrupted parties and clients:

- In the input phase, for each group of $k$ input gates that belong to some `Client`, let $\boldsymbol{\alpha}$ denote the batch of output wires of these input gates. If `Client` is honest, then $\mathcal{S}$ needs to simulate the values $\boldsymbol{\mu_\alpha}$ sent from `Client` to $P_1$. In the ideal world, $\mathcal{S}$ simply samples random elements as $\boldsymbol{\mu_\alpha}$. Since $\boldsymbol{\mu_\alpha} = \boldsymbol{v_\alpha} - \boldsymbol{\lambda_\alpha}$ and $\boldsymbol{\lambda_\alpha}$ are uniformly random, the values $\boldsymbol{\mu_\alpha}$ are uniformly random. Therefore the distribution of $\boldsymbol{\mu_\alpha}$ simulated by $\mathcal{S}$ has the same distribution as that in the real world.
  If `Client` is corrupted, then $\mathcal{S}$ needs to simulate the shares of $[\![\boldsymbol{\lambda_\alpha}]\!]_{n-1}$ of honest parties. Since $[\![\boldsymbol{\lambda_\alpha}]\!]_{n-1}$ is a random degree-$(n-1)$ packed Shamir sharing given the shares of corrupted parties, by the property of the packed Shamir secret sharing scheme, the secrets $\boldsymbol{\lambda_\alpha}$ are uniformly

random given the shares of corrupted parties. In the ideal world, $\mathcal{S}$ randomly samples $\boldsymbol{\lambda_\alpha}$ and then randomly samples the shares of honest parties based on the secrets $\boldsymbol{\lambda_\alpha}$ and the shares of corrupted parties. Therefore, the distribution of the shares of $[\![\boldsymbol{\lambda_\alpha}]\!]_{n-1}$ of honest parties is identical to that in the real world. Note that from the inputs of `Client`, $\mathcal{S}$ can also compute $\boldsymbol{\mu_\alpha}$. Thus, $\mathcal{S}$ perfectly simulates the behaviors of honest parties and clients in the input phase.

– In the computation phase, we will show that $\mathcal{S}$ can always learn $\mu_\alpha$ for each wire $\alpha$ in the circuit. Furthermore, $\{\mu_\alpha\}_\alpha$ has the same distribution as that in the real world. Note that this is true for the first layer (the input layer).
For an addition gate with input wires $\alpha, \beta$ and output wire $\gamma$, $\mathcal{S}$ can compute $\mu_\gamma$ from $\mu_\alpha$ and $\mu_\beta$. The above statement holds.
For each group of $k$ multiplication gates with input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$ and output wires $\boldsymbol{\gamma}$, there are two cases.
   - If $P_1$ is honest, $\mathcal{S}$ honestly computes and distributes the sharings $[\![\boldsymbol{\mu_\alpha}]\!]_{k-1}$ and $[\![\boldsymbol{\mu_\beta}]\!]_{k-1}$. Since the distribution of $\boldsymbol{\mu_\alpha}, \boldsymbol{\mu_\beta}$ is the same in both worlds, the distribution of the sharings $[\![\boldsymbol{\mu_\alpha}]\!]_{k-1}, [\![\boldsymbol{\mu_\beta}]\!]_{k-1}$ is also the same. As for $[\![\boldsymbol{\mu_\gamma}]\!]_{n-1}$, recall that
$$\begin{aligned}[\![\boldsymbol{\mu_\gamma}]\!]_{n-1} = [\![\boldsymbol{\mu_\alpha}]\!]_{k-1} * [\![\boldsymbol{\mu_\beta}]\!]_{k-1} + [\![\boldsymbol{\mu_\alpha}]\!]_{k-1} * [\![\boldsymbol{\lambda_\beta}]\!]_{n-k} \\ + [\![\boldsymbol{\mu_\beta}]\!]_{k-1} * [\![\boldsymbol{\lambda_\alpha}]\!]_{n-k} + [\![\boldsymbol{\Gamma_\gamma}]\!]_{n-1}.\end{aligned}$$
Also recall that $\boldsymbol{\Gamma_\gamma} = \boldsymbol{\lambda_\alpha} * \boldsymbol{\lambda_\beta} - \boldsymbol{\lambda_\gamma}$. In $\mathcal{F}_{\mathsf{Prep}}$, $\boldsymbol{\lambda_\gamma}$ are uniformly random. Therefore, $\boldsymbol{\Gamma_\gamma}$ are also uniformly random. Thus, $[\![\boldsymbol{\Gamma_\gamma}]\!]_{n-1}$ is a random degree-$(n-1)$ packed Shamir sharing given the shares of corrupted parties. It satisfies that the shares of honest parties are uniformly random. Thus, the shares of $[\![\boldsymbol{\mu_\gamma}]\!]_{n-1}$ of honest parties are uniformly random. In the ideal world, $\mathcal{S}$ samples random elements as the shares of $[\![\boldsymbol{\mu_\gamma}]\!]_{n-1}$ and then computes $\boldsymbol{\mu_\gamma}$. Thus, the values $\boldsymbol{\mu_\gamma}$ have the same distribution as those in the real world.
   - If $P_1$ is corrupted, $\mathcal{S}$ receives the shares of $[\![\boldsymbol{\mu_\alpha}]\!]_{k-1}, [\![\boldsymbol{\mu_\beta}]\!]_{k-1}$ of honest parties. Then $\mathcal{S}$ can compute the shares of honest parties. Then $\mathcal{S}$ can compute the shares of $[\![\boldsymbol{\mu_\gamma}]\!]_{n-1}$ of corrupted parties. With the same argument as above, the shares of $[\![\boldsymbol{\mu_\gamma}]\!]_{n-1}$ of honest parties are uniformly random. In the ideal world, $\mathcal{S}$ samples random elements as the shares of $[\![\boldsymbol{\mu_\gamma}]\!]_{n-1}$ of honest parties and sends them to $P_1$. Therefore, the distribution of the shares of $[\![\boldsymbol{\mu_\gamma}]\!]_{n-1}$ of honest parties and the values $\boldsymbol{\mu_\gamma}$ is identical in both worlds.

In either case, $\mathcal{S}$ perfectly simulates the behaviors of honest parties and the values $\boldsymbol{\mu_\gamma}$ have the same distribution as those in the real world.

– Finally, in the output phase, for each group of $k$ output gates that belong to some `Client`, let $\boldsymbol{\alpha}$ denote the batch of input wires of these output gates. If `Client` is honest, honest parties and clients do not need to send any messages to corrupted parties and clients. If `Client` is corrupted, $\mathcal{S}$ can learn the outputs of `Client` from $\mathcal{F}_{\mathsf{Main}}$. Since $\mathcal{S}$ learns $\boldsymbol{\mu_\alpha}$, $\mathcal{S}$ can compute $\boldsymbol{\lambda_\alpha}$. In both worlds, $[\![\boldsymbol{\lambda_\alpha}]\!]_{n-1}$ is a random degree-$(n-1)$ packed Shamir sharing given the secrets $\boldsymbol{\lambda_\alpha}$ and the shares of corrupted parties. Thus, the shares of honest parties generated by $\mathcal{S}$ have the same distribution as that in the real world.

## B.2 Proof of Lemma 2

**Lemma 2.** *Protocol $\Pi_{\mathsf{Prep}}$ securely computes $\mathcal{F}_{\mathsf{Prep}}$ in the $\mathcal{F}_{\mathsf{PrepInd}}$-hybrid model against a semi-honest adversary who controls $t$ out of $n = 2t+1$ parties.*

*Proof.* It can be verified that the secrets of the output sharings of $\Pi_{\mathsf{Prep}}$ have the same distribution of those of $\mathcal{F}_{\mathsf{Prep}}$.

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.

1. In Step 1, $\mathcal{S}$ emulates the ideal functionality $\mathcal{F}_{\mathsf{PrepInd}}$ and learns the shares of corrupted parties.
2. In Step 2, $\mathcal{S}$ follows the protocol to compute the shares of corrupted parties for each $[\![\lambda_\alpha \cdot \mathbf{1}]\!]_{n-k}$.
3. In Step 3, for each group of multiplication gates, $\mathcal{S}$ follows the protocol to compute the shares of $[\![\boldsymbol{\lambda_\alpha}]\!]_{n-1}$ and $[\![\boldsymbol{\lambda_\beta}]\!]_{n-1}$ of corrupted parties. Then $\mathcal{S}$ simulates the process of degree reduction for $[\![\boldsymbol{\lambda_\alpha}]\!]_{n-1}$ as follows:
   (a) In Step 3.(a), $\mathcal{S}$ computes the shares of $[\![\boldsymbol{\lambda_\alpha} + \boldsymbol{a}]\!]_{n-1}$ of corrupted parties and sets the shares of honest parties to be uniform values.

(b) In Step 3.(b), if $P_1$ is honest, $\mathcal{S}$ honestly follows the protocol. Otherwise, $\mathcal{S}$ sends the shares of $[\![\boldsymbol{\lambda_\alpha}+\boldsymbol{a}]\!]_{n-1}$ of honest parties to $P_1$ and receives the shares of $[\![\boldsymbol{d}]\!]_{k-1}$ of honest parties. $\mathcal{S}$ recovers the whole sharing $[\![\boldsymbol{d}]\!]_{k-1}$ and learns the shares of corrupted parties.

(c) In Step 3.(c), $\mathcal{S}$ follows the protocol to compute the shares of $[\![\boldsymbol{\lambda_\alpha}]\!]_{n-k}$ of corrupted parties and sends them to $\mathcal{F}_{\mathsf{Prep}}$.

$\mathcal{S}$ simulates the degree reduction for $[\![\boldsymbol{\lambda_\beta}]\!]_{n-1}$ similarly.

4. In Step 4, for each group of input gates, $\mathcal{S}$ follows the protocol to compute the shares of $[\![\boldsymbol{\lambda_\alpha}]\!]_{n-1}$ of corrupted parties. Then, $\mathcal{S}$ computes the shares of $[\![\boldsymbol{\lambda_\alpha}]\!]_{n-1} := [\![\boldsymbol{\lambda_\alpha}]\!]_{n-1} + [\![\boldsymbol{o}]\!]_{n-1}$ held by corrupted parties and sends them to $\mathcal{F}_{\mathsf{Prep}}$.
   $\mathcal{S}$ does the same for the input wires of each group of output gates.
5. In Step 5, $\mathcal{S}$ follows the protocol and computes the shares of $[\![\boldsymbol{\Gamma_\gamma}]\!]_{n-1}$ of corrupted parties and sends them to $\mathcal{F}_{\mathsf{Prep}}$.

This completes the description of $\mathcal{S}$. Now, we show that $\mathcal{S}$ perfectly simulate the behaviors of honest parties. We note that the only step where honest parties need to send messages to corrupted parties is Step 3.(b). Observe that $[\![\boldsymbol{a}]\!]_{n-k}$ is a random degree-$(n-k)$ packed Shamir sharing and $[\![\boldsymbol{o}^{(1)}]\!]_{n-1}$ is a random degree-$(n-1)$ packed Shamir sharing of $\mathbf{0} \in \mathbb{F}^k$. Therefore, $[\![\boldsymbol{a}]\!]_{n-k} + [\![\boldsymbol{o}^{(1)}]\!]_{n-1}$ is a random degree-$(n-1)$ packed Shamir sharing. Recall that $[\![\boldsymbol{\lambda_\alpha}+\boldsymbol{a}]\!]_{n-1} = [\![\boldsymbol{\lambda_\alpha}]\!]_{n-1} + [\![\boldsymbol{a}]\!]_{n-k} + [\![\boldsymbol{o}^{(1)}]\!]_{n-1}$. Thus, $[\![\boldsymbol{\lambda_\alpha}+\boldsymbol{a}]\!]_{n-1}$ is a random degree-$(n-1)$ packed Shamir sharing, which satisfies that the shares of honest parties are uniformly random given the shares of corrupted parties. Thus, the shares of honest parties generated by $\mathcal{S}$ have the same distribution in both worlds. After generating the shares of $[\![\boldsymbol{\lambda_\alpha}+\boldsymbol{a}]\!]_{n-1}$ of honest parties, $\mathcal{S}$ honestly follows Step 3.(b). Therefore, $\mathcal{S}$ perfectly simulates the behaviors of honest parties.

Then, we analyse the output of $\Pi_{\mathsf{Prep}}$. For each degree-$(n-k)$ packed Shamir sharing of $\boldsymbol{\lambda_\alpha}$ prepared in Step 3, $\mathcal{S}$ can compute the shares of corrupted parties as described above. Since the secrets $\boldsymbol{\lambda_\alpha}$ in the real world have the same distribution as those computed by $\mathcal{F}_{\mathsf{Prep}}$, the sharing $[\![\boldsymbol{\lambda_\alpha}]\!]_{n-k}$ has the same distribution in both worlds.

For each degree-$(n-1)$ packed Shamir sharing of $\boldsymbol{\lambda_\alpha}$ prepared in Step 4, $\mathcal{S}$ can compute the shares of corrupted parties as described above. In the real world, since $[\![\boldsymbol{o}]\!]_{n-1}$ is a random degree-$(n-1)$ packed Shamir sharing of $\mathbf{0} \in \mathbb{F}^k$, $[\![\boldsymbol{\lambda_\alpha}]\!]_{n-1}$ is a random degree-$(n-1)$ packed Shamir sharing given the secrets $\boldsymbol{\lambda_\alpha}$ and the shares of corrupted parties. In the ideal world, $\mathcal{F}_{\mathsf{Prep}}$ generates a random degree-$(n-1)$ packed Shamir sharing of $\boldsymbol{\lambda_\alpha}$ given the shares of corrupted parties. Therefore, the sharing $[\![\boldsymbol{\lambda_\alpha}]\!]_{n-1}$ has the same distribution in both worlds.

Similarly, for each packed Shamir sharing of $\boldsymbol{\Gamma_\gamma}$, recall that

$$\begin{aligned}[\![\boldsymbol{\Gamma_\gamma}]\!]_{n-1} = {} & [\![\boldsymbol{d}_1]\!]_{k-1} * [\![\boldsymbol{d}_2]\!]_{k-1} - [\![\boldsymbol{d}_1]\!]_{k-1} * [\![\boldsymbol{b}]\!]_{n-k} \\ & - [\![\boldsymbol{d}_2]\!]_{k-1} * [\![\boldsymbol{a}]\!]_{n-k} + [\![\boldsymbol{c}]\!]_{n-k} \\ & - (\sum_{i=1}^{k} \boldsymbol{e}_i * [\![\lambda_{\gamma_i} \cdot \mathbf{1}]\!]_{n-k}) + [\![\boldsymbol{o}^{(3)}]\!]_{n-1}.\end{aligned}$$

$\mathcal{S}$ can compute the shares of corrupted parties. In the real world, since $[\![\boldsymbol{o}^{(3)}]\!]_{n-1}$ is a random degree-$(n-1)$ packed Shamir sharing of $\mathbf{0} \in \mathbb{F}^k$, $[\![\boldsymbol{\Gamma_\gamma}]\!]_{n-1}$ is a random degree-$(n-1)$ packed Shamir sharing of $\boldsymbol{\Gamma_\gamma}$ given the shares of corrupted parties. In the ideal world, $\mathcal{F}_{\mathsf{Prep}}$ generates a random degree-$(n-1)$ packed Shamir sharing of $\boldsymbol{\Gamma_\gamma}$ given the shares of corrupted parties. Therefore, the distribution of $[\![\boldsymbol{\Gamma_\gamma}]\!]_{n-1}$ is identical in both worlds.

We conclude that Protocol $\Pi_{\mathsf{Prep}}$ securely computes $\mathcal{F}_{\mathsf{Prep}}$ in the $\mathcal{F}_{\mathsf{PrepInd}}$-hybrid model against a semi-honest adversary who controls $t$ parties.

### B.3 Proof of Lemma 3

**Lemma 3.** *Protocol $\Pi_{\mathsf{PrepInd}}$ securely computes $\mathcal{F}_{\mathsf{PrepInd}}$ in the $\mathcal{F}_{\mathsf{SingleMult}}$-hybrid model against a semi-honest adversary who controls $t$ out of $n = 2t+1$ parties.*

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

*Simulating* $\pi_{\mathsf{RandSh}}$.

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 $P_i$, $\mathcal{S}$ generates a random $\Sigma$-sharing and sends the shares to corrupted parties. For each corrupted party $P_i$, $\mathcal{S}$ learns the sharing generated by $P_i$ (since $\mathcal{S}$ has access to the random tape of the corrupted party $P_i$).
3. In Step 3, $\mathcal{S}$ computes the shares of corrupted parties for each $\Sigma$-sharing $\boldsymbol{R}^{(i)}$.

*Simulating the Main Protocol.*

1. In Step 1, $\mathcal{S}$ simulates $\pi_{\mathsf{RandSh}}(\Sigma_1)$ as described above. Then $\mathcal{S}$ sends the shares of corrupted parties to $\mathcal{F}_{\mathsf{PrepInd}}$.
2. In Step 2, $\mathcal{S}$ simulates $\pi_{\mathsf{RandSh}}(\Sigma_{2,i})$ for all $i \in \{1, 2, \ldots, k\}$ as described above. Then, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{SingleMult}}$ and receives the shares of corrupted parties. Finally, for each group of multiplication gates, $\mathcal{S}$ computes the shares of $([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, [\![\boldsymbol{c}]\!]_{n-k})$ of corrupted parties, and sends them to $\mathcal{F}_{\mathsf{PrepInd}}$.
3. In Step 3, $\mathcal{S}$ simulates $\pi_{\mathsf{RandSh}}(\Sigma_3)$ as described above. Then $\mathcal{S}$ sends the shares of corrupted parties to $\mathcal{F}_{\mathsf{PrepInd}}$.
4. In Step 4, $\mathcal{S}$ simulates $\pi_{\mathsf{RandSh}}(\Sigma_3)$ as described above. Then $\mathcal{S}$ sends the shares of corrupted parties to $\mathcal{F}_{\mathsf{PrepInd}}$.

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

Now, we show that $\mathcal{S}$ perfectly simulate the behaviors of honest parties. We note that the only place where honest parties need to send messages to corrupted parties is in Step 2 of $\pi_{\mathsf{RandSh}}$. The simulator $\mathcal{S}$ honestly generates a random $\Sigma$-sharing as that in the real world. Thus, $\mathcal{S}$ perfectly simulates the behaviors of honest parties.

Then, we show that the output of honest parties in both worlds have the same distribution. For $\pi_{\mathsf{RandSh}}$, recall that the matrix $\boldsymbol{M}^{\mathrm{T}}$ is a Vandermonde matrix of size $n \times (t+1)$, which satisfies that any $(t+1) \times (t+1)$ sub-matrix of $\boldsymbol{M}^{\mathrm{T}}$ is invertible. Therefore, given the $\Sigma$-sharings prepared by corrupted parties and the shares of corrupted parties, there is a one-to-one map between $\{\boldsymbol{S}^{(i)}\}_{i\in\mathcal{H}}$ and $\{\boldsymbol{R}^{(i)}\}_{i=1}^{t+1}$. Since $\{\boldsymbol{S}^{(i)}\}_{i\in\mathcal{H}}$ are $n-t=t+1$ random $\Sigma$-sharings given the shares of corrupted parties, $\{\boldsymbol{R}^{(i)}\}_{i=1}^{t+1}$ are also $t+1$ random $\Sigma$-sharings given the shares of corrupted parties. Thus, the random sharings generated in Step 1, Step 3, and Step 4 have the same distribution in both worlds.

For Step 2 in the main protocol, following from the same argument as above, all parties hold random degree-$t$ Shamir sharings from $\pi_{\mathsf{RandSh}}(\Sigma_{2,i})$ for all $i \in \{1, 2, \ldots, k\}$. For each group of multiplication gates, the sharings $\{[\![a_i|_i]\!]_t, [\![b_i|_i]\!]_t\}_{i=1}^t$ satisfy that $a_i, b_i$ are uniform values in the real world. Then, all parties receive $[\![c_i|_i]\!]_t$ from $\mathcal{F}_{\mathsf{SingleMult}}$ such that $c_i = a_i \cdot b_i$. Finally, all parties locally compute

$$[\![\boldsymbol{a}]\!]_{n-k} = \sum_{i=1}^{k} [\![\boldsymbol{e}_i]\!]_{k-1} * [\![a_i|_i]\!]_t \qquad [\![\boldsymbol{b}]\!]_{n-k} = \sum_{i=1}^{k} [\![\boldsymbol{e}_i]\!]_{k-1} * [\![b_i|_i]\!]_t$$

$$[\![\boldsymbol{c}]\!]_{n-k} = \sum_{i=1}^{k} [\![\boldsymbol{e}_i]\!]_{k-1} * [\![c_i|_i]\!]_t$$

Observe that $[\![\boldsymbol{a}]\!]_{n-k}$ is determined by the secrets $\boldsymbol{a}$ and the shares of corrupted parties. In the ideal world, $\boldsymbol{a}$ is a uniform vector sampled by $\mathcal{F}_{\mathsf{PrepInd}}$ and the shares are provided by the simulator $\mathcal{S}$. Recall that $\mathcal{S}$ simply follows the protocol to compute the shares of corrupted parties. Thus, $[\![\boldsymbol{a}]\!]_{n-k}$ has the same distribution in both worlds. The same argument works for $[\![\boldsymbol{b}]\!]_{n-k}$ and $[\![\boldsymbol{c}]\!]_{n-k}$.

We conclude that Protocol $\Pi_{\mathsf{PrepInd}}$ securely computes $\mathcal{F}_{\mathsf{PrepInd}}$ in the $\mathcal{F}_{\mathsf{SingleMult}}$-hybrid model against a semi-honest adversary who controls $t$ parties.

## C Optimization of TurboPack

In this part, we show an optimization of TurboPack which allows us to further reduce the communication complexity by a factor of 2 in the circuit-dependent preprocessing phase.

Recall that in the online phase, we want to maintain the invariant that $P_1$ learns $\mu_\alpha = v_\alpha - \lambda_\alpha$ for all wires in the circuit, where $v_\alpha$ is the real value and $\lambda_\alpha$ is a random value prepared in the preprocessing phase. To this end, for each group of multiplication gates with input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$ and output wires $\boldsymbol{\gamma}$, we use the technique of packed Beaver triples to compute $\boldsymbol{\mu}_{\boldsymbol{\gamma}}$, which requires all parties to prepare the following random sharings: $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}]\!]_{n-k}, [\![\boldsymbol{\lambda}_{\boldsymbol{\beta}}]\!]_{n-k}$ and $[\![\boldsymbol{\Gamma}_{\boldsymbol{\gamma}}]\!]_{n-1}$, where $\boldsymbol{\Gamma}_{\boldsymbol{\gamma}} = \boldsymbol{\lambda}_{\boldsymbol{\alpha}} * \boldsymbol{\lambda}_{\boldsymbol{\beta}} - \boldsymbol{\lambda}_{\boldsymbol{\gamma}}$.

In the circuit-dependent preprocessing phase, to prepare $[\![\boldsymbol{\Gamma}_\gamma]\!]_{n-1}$, we again use the technique of packed Beaver triples, which requires all parties to prepare a packed Beaver triple $([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, [\![\boldsymbol{c}]\!]_{n-k})$. Our optimization is to directly use a packed Beaver triple to compute $\boldsymbol{\mu}_\gamma$ in the online phase. That is, we skip the step of preparing $[\![\boldsymbol{\Gamma}_\gamma]\!]_{n-1}$ first.

Recall that $\boldsymbol{\mu}_\gamma = \boldsymbol{v}_\gamma - \boldsymbol{\lambda}_\gamma = \boldsymbol{v}_{\boldsymbol{\alpha}} * \boldsymbol{v}_{\boldsymbol{\beta}} - \boldsymbol{\lambda}_\gamma$. The main task is to compute $\boldsymbol{v}_{\boldsymbol{\alpha}} * \boldsymbol{v}_{\boldsymbol{\beta}}$. The technique of packed Shamir sharing requires all parties to hold degree-$(k-1)$ packed sharings $[\![\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{a}]\!]_{k-1}$ and $[\![\boldsymbol{v}_{\boldsymbol{\beta}} + \boldsymbol{b}]\!]_{k-1}$. In the online phase, the invariant ensures that $P_1$ learns $\boldsymbol{\mu}_{\boldsymbol{\alpha}}$ and $\boldsymbol{\mu}_{\boldsymbol{\beta}}$. If $P_1$ also learns $(\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{a}) - \boldsymbol{\mu}_{\boldsymbol{\alpha}}$ and $(\boldsymbol{v}_{\boldsymbol{\beta}} + \boldsymbol{b}) - \boldsymbol{\mu}_{\boldsymbol{\beta}}$, $P_1$ can distribute degree-$(k-1)$ packed sharings $[\![\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{a}]\!]_{k-1}$ and $[\![\boldsymbol{v}_{\boldsymbol{\beta}} + \boldsymbol{b}]\!]_{k-1}$ to all parties.

Note that $(\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{a}) - \boldsymbol{\mu}_{\boldsymbol{\alpha}} = \boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \boldsymbol{a}$, which are indeed learnt by $P_1$ in the circuit-dependent preprocessing phase. Recall that in the circuit-dependent preprocessing phase, all parties first compute a degree-$(n-1)$ packed Shamir sharing $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}]\!]_{n-1}$. Then they do degree reduction by using $([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{o}^{(1)}]\!]_{n-1})$, where $[\![\boldsymbol{o}^{(1)}]\!]_{n-1}$ is a random degree-$(n-1)$ packed Shamir sharing of $\boldsymbol{0} \in \mathbb{F}^k$ prepared in $\mathcal{F}_{\mathsf{PrepInd}}$, and the resulting sharing has the form

$$[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}]\!]_{n-k} = [\![\boldsymbol{d}_1]\!]_{k-1} - [\![\boldsymbol{a}]\!]_{n-k}.$$

During this process, $P_1$ reconstructs the sharing $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \boldsymbol{a}]\!]_{n-1} = [\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}]\!]_{n-1} + [\![\boldsymbol{a}]\!]_{n-k} + [\![\boldsymbol{o}^{(1)}]\!]_{n-1}$ and learns $\boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \boldsymbol{a}$.

Therefore, we modify the online protocol by letting $P_1$ distribute $[\![\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{a}]\!]_{k-1}$ and $[\![\boldsymbol{v}_{\boldsymbol{\beta}} + \boldsymbol{b}]\!]_{k-1}$. Then all parties can use the technique of packed Beaver triples to compute $\boldsymbol{\mu}_\gamma$. For the circuit-dependent preprocessing phase, we no longer needs to compute $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}]\!]_{n-k}$, $[\![\boldsymbol{\lambda}_{\boldsymbol{\beta}}]\!]_{n-k}$ and $[\![\boldsymbol{\Gamma}_\gamma]\!]_{n-1}$. Therefore, in Step 3.(b) of $\Pi_{\mathsf{Prep}}$, $P_1$ no longer needs to distribute $[\![\boldsymbol{d}]\!]_{k-1}$ to all parties. As a result, the communication complexity of the circuit-dependent preprocessing phase is reduced to $2|C| \cdot n/k \approx 8|C|$ elements.

We elaborate the modifications as follows.

## C.1 Modification of the Circuit-Dependent Preprocessing Phase

We modify the protocol $\Pi_{\mathsf{Prep}}$ as follows.

- In Step 3.(b), $P_1$ collects the whole sharing $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \boldsymbol{a}]\!]_{n-1}$ from all parties and reconstructs the secrets $\boldsymbol{d} = \boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \boldsymbol{a}$. $P_1$ does NOT distribute the degree-$(k-1)$ packed Shamir sharing $[\![\boldsymbol{d}]\!]_{k-1}$ to all parties.
- Step 3.(c) is removed.
- At the end of Step 3, all parties compute $[\![\boldsymbol{\lambda}_{\boldsymbol{\beta}} + \boldsymbol{b}]\!]_{n-1}$ and reconstruct the secrets to $P_1$ in a similar way.
- In Step 4, the same step is done also for the output wires of each group of multiplication gates by using $[\![\boldsymbol{o}^{(3)}]\!]_{n-1}$. That is, all parties will locally compute a random degree-$(n-1)$ packed Shamir sharing $[\![\boldsymbol{\lambda}_\gamma]\!]_{n-1}$ for the output wires $\boldsymbol{\gamma}$ of each group of multiplication gates.
- Step 5 is replaced as follows: For each group of multiplication gates with input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$ and output wires $\boldsymbol{\gamma}$, all parties take as output the packed Beaver triple $([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, [\![\boldsymbol{c}]\!]_{n-k})$. $P_1$ also takes $\boldsymbol{d}_1, \boldsymbol{d}_2$ as output. Here $\boldsymbol{d}_1 = \boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \boldsymbol{a}$ and $\boldsymbol{d}_2 = \boldsymbol{\lambda}_{\boldsymbol{\beta}} + \boldsymbol{b}$.

For completeness, we give the functionality corresponding to the improved circuit-dependent preprocessing phase.

**Functionality 5: $\mathcal{F}_{\mathsf{PrepImproved}}$**

1. **Assign Random Values to Wires in** $C$: $\mathcal{F}_{\mathsf{PrepImproved}}$ receives the circuit $C$ from all parties. Then $\mathcal{F}_{\mathsf{PrepImproved}}$ assigns random values to wires in $C$ as follows.
   (a) For each output wire $\alpha$ of an input gate or a multiplication gate, $\mathcal{F}_{\mathsf{PrepImproved}}$ samples a uniform value $\lambda_\alpha$ and associates it with the wire $\alpha$.
   (b) Starting from the first layer of $C$ to the last layer, for each addition gate with input wires $\alpha, \beta$ and output wire $\gamma$, $\mathcal{F}_{\mathsf{PrepImproved}}$ sets $\lambda_\gamma = \lambda_\alpha + \lambda_\beta$.
2. **Preparing Packed Beaver Triples**: For each intermediate layer in $C$, all multiplication gates are divided into groups of size $k$. For each group of $k$ multiplication gates, $\mathcal{F}_{\mathsf{PrepImproved}}$ prepares a packed Beaver triple $([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, [\![\boldsymbol{c}]\!]_{n-k})$, which satisfies that $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$. This is done by the following steps.
   (a) $\mathcal{F}_{\mathsf{PrepImproved}}$ receives the set of corrupted parties, denoted by $\mathcal{C}orr$. $\mathcal{F}_{\mathsf{PrepImproved}}$ receives from the adversary a set of shares $\{(u_j^{(1)}, u_j^{(2)}, u_j^{(3)})\}_{j \in \mathcal{C}orr}$. $\mathcal{F}_{\mathsf{PrepImproved}}$ samples two random vec-

tors $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{F}^k$ and computes $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$. Then $\mathcal{F}_{\mathsf{PrepImproved}}$ computes three degree-$(n-k)$ packed Shamir sharings $[\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, [\![\boldsymbol{c}]\!]_{n-k}$ such that for all $P_j \in \mathcal{C}orr$, the $j$-th share of $([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, [\![\boldsymbol{c}]\!]_{n-k})$ is $(u_j^{(1)}, u_j^{(2)}, u_j^{(3)})$.

(b) $\mathcal{F}_{\mathsf{PrepImproved}}$ distributes the shares of $([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, [\![\boldsymbol{c}]\!]_{n-k})$ to honest parties.

3. **Distributing $\boldsymbol{\lambda_\alpha} + \boldsymbol{a}$ 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 $([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, [\![\boldsymbol{c}]\!]_{n-k})$ be the packed Beaver triple associated with these gates. $\mathcal{F}_{\mathsf{PrepImproved}}$ computes $\boldsymbol{\lambda_\alpha} + \boldsymbol{a}$ and $\boldsymbol{\lambda_\beta} + \boldsymbol{b}$, 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 $\boldsymbol{\beta}$.
4. **Preparing Degree-$(n-1)$ Packed Shamir Sharings**: $\mathcal{F}_{\mathsf{PrepImproved}}$ will prepare degree-$(n-1)$ packed Shamir sharings for the following batches of wires:
   - For the input layer, all input gates are divided into groups of size $k$ such that the input gates of each group belong to the same client. For each group of input gates with output wires $\boldsymbol{\alpha}$, $\mathcal{F}_{\mathsf{PrepImproved}}$ will prepare a degree-$(n-1)$ packed Shamir sharing of $\boldsymbol{\lambda_\alpha}$.
   - For the output layer in $C$, all output gates are divided into groups of size $k$ such that the output gates of each group belong to the same client. For each group of output gates with input wires $\boldsymbol{\alpha}$, $\mathcal{F}_{\mathsf{PrepImproved}}$ will prepare a degree-$(n-1)$ packed Shamir sharing of $\boldsymbol{\lambda_\alpha}$.
   - For each group of multiplication gates with output wires $\boldsymbol{\gamma}$, $\mathcal{F}_{\mathsf{PrepImproved}}$ will prepare a degree-$(n-1)$ packed Shamir sharing of $\boldsymbol{\lambda_\gamma}$.

   For each batch of wires $\boldsymbol{\alpha}$ in the above scenarios, $\mathcal{F}_{\mathsf{PrepImproved}}$ does the following.

   (a) $\mathcal{F}_{\mathsf{PrepImproved}}$ receives from the adversary a set of shares $\{u_j\}_{j \in \mathcal{C}orr}$. $\mathcal{F}_{\mathsf{PrepImproved}}$ samples a random degree-$(n-1)$ packed Shamir sharing $[\![\boldsymbol{\lambda_\alpha}]\!]_{n-1}$ such that for all $P_j \in \mathcal{C}orr$, the $j$-th share of $[\![\boldsymbol{\lambda_\alpha}]\!]_{n-1}$ is $u_j$.

   (b) $\mathcal{F}_{\mathsf{PrepImproved}}$ distributes the shares of $[\![\boldsymbol{\lambda_\alpha}]\!]_{n-1}$ to honest parties.

## C.2 Modification of the Online Phase

We modify the protocol $\Pi_{\mathsf{Mult}}$ as follows.

**Protocol 7: $\Pi_{\mathsf{MultImproved}}$**

1. For each group of multiplication gates with input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$ and output wires $\boldsymbol{\gamma}$, all parties receive from $\mathcal{F}_{\mathsf{PrepImproved}}$
   - A packed Beaver triple $([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, [\![\boldsymbol{c}]\!]_{n-k})$,
   - A random degree-$(n-1)$ packed Shamir sharing $[\![\boldsymbol{\lambda_\gamma}]\!]_{n-1}$.

   $P_1$ receives from $\mathcal{F}_{\mathsf{PrepImproved}}$ two vectors $\boldsymbol{d}_1 = \boldsymbol{\lambda_\alpha} + \boldsymbol{a}, \boldsymbol{d}_2 = \boldsymbol{\lambda_\beta} + \boldsymbol{b}$. $P_1$ also learns $\boldsymbol{\mu_\alpha}, \boldsymbol{\mu_\beta}$ during the online phase.
2. $P_1$ locally computes $\boldsymbol{v_\alpha} + \boldsymbol{a} = \boldsymbol{\mu_\alpha} + \boldsymbol{d}_1$ and $\boldsymbol{v_\beta} + \boldsymbol{b} = \boldsymbol{\mu_\beta} + \boldsymbol{d}_2$. Then, $P_1$ computes $[\![\boldsymbol{v_\alpha} + \boldsymbol{a}]\!]_{k-1}$ and $[\![\boldsymbol{v_\beta} + \boldsymbol{b}]\!]_{k-1}$ and distributes the shares to all parties.
3. All parties locally compute

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

4. $P_1$ collects the whole sharing $[\![\boldsymbol{\mu_\gamma}]\!]_{n-1}$ from all parties and reconstructs $\boldsymbol{\mu_\gamma}$.

## C.3 Theorem for the Improved Protocol

As for the security of our improved protocol, we note that

- In the original protocol, for each group of multiplication gates with input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$, $P_1$ will distribute $[\![\boldsymbol{\lambda_\alpha} + \boldsymbol{a}]\!]_{k-1}, [\![\boldsymbol{\lambda_\beta} + \boldsymbol{b}]\!]_{k-1}$ in the circuit-dependent preprocessing phase and distribute $[\![\boldsymbol{\mu_\alpha}]\!]_{k-1}, [\![\boldsymbol{\mu_\beta}]\!]_{k-1}$ in the online phase.
- In the improved protocol, for each group of multiplication gates with input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$, $P_1$ will distribute $[\![\boldsymbol{v_\alpha} + \boldsymbol{a}]\!]_{k-1}, [\![\boldsymbol{v_\beta} + \boldsymbol{b}]\!]_{k-1}$ in the online phase.

Observe that $[\![\boldsymbol{v_\alpha} + \boldsymbol{a}]\!]_{k-1} = [\![\boldsymbol{\lambda_\alpha} + \boldsymbol{a}]\!]_{k-1} + [\![\boldsymbol{\mu_\alpha}]\!]_{k-1}$ and $[\![\boldsymbol{v_\beta} + \boldsymbol{b}]\!]_{k-1} = [\![\boldsymbol{\lambda_\beta} + \boldsymbol{b}]\!]_{k-1} + [\![\boldsymbol{\mu_\beta}]\!]_{k-1}$. Therefore, the messages exchanged in the improved protocol can be derived from the messages exchanged in the original protocol. It implies that any attack made by a semi-honest adversary towards the improved protocol also works for the original protocol. Thus, the improved protocol achieves at least the same level of security as the original protocol. We have the following theorem.

**Theorem 2.** *In the client-server model, let $\mathsf{c}$ denote the number of clients, $n$ denote the number of parties (servers), and $t = (n-1)/2$ denote the number of corrupted parties (servers). Let $\mathbb{F}$ be a finite field of size $|\mathbb{F}| \geq 2n$. For an arithmetic circuit $C$ over $\mathbb{F}$, there exists an information-theoretic MPC protocol which securely computes the arithmetic circuit $C$ in the presence of a semi-honest adversary controlling up to $\mathsf{c}$ clients and $t$ parties. The splitting communication complexity per gate is (1) $10n+24$ elements per gate in the circuit-independent preprocessing phase, (2) $8$ elements per gate in the circuit-dependent preprocessing phase, and (3) $12$ elements per gate in the online phase. (Terms that are independent of or sub-linear in the circuit size are omitted as they only add cost $o(1)$ per gate.)*

## D Malicious Security

In this section, we discuss how to compile TurboPack to achieve malicious security. We observe that the main difficulty comes from the fact that degree-$(n-k)$ packed Shamir sharing is not robust: corrupted parties can change the secrets of a degree-$(n-k)$ packed Shamir sharing by locally changing their own shares. Concretely, for a degree-$(n-k)$ packed Shamir sharing $[\![\boldsymbol{x}]\!]_{n-k}$, corrupted parties can locally compute a degree-$(n-k)$ packed Shamir sharing $[\![\varDelta(\boldsymbol{x})]\!]_{n-k}$ where

- The shares of $[\![\varDelta(\boldsymbol{x})]\!]_{n-k}$ of honest parties are $0$.
- The first $k-1$ values of $\varDelta(\boldsymbol{x})$ can be arbitrary values.

Recall that $k = (n-t+1)/2$, we have $n-k = t+k-1$. It means that a degree-$(n-k)$ packed Shamir sharing can be determined by $n-k+1 = t+k$ values. Note that the above have fixed $t+k$ values. Corrupted parties can locally compute the last value of $\varDelta(\boldsymbol{x})$ and their shares of $[\![\varDelta(\boldsymbol{x})]\!]_{n-k}$. Since the shares of $[\![\varDelta(\boldsymbol{x})]\!]_{n-k}$ held by honest parties are $0$, corrupted parties can locally compute $[\![\boldsymbol{x}+\varDelta(\boldsymbol{x})]\!]_{n-k}$, causing a change of the secrets by $\varDelta(\boldsymbol{x})$ without being noticed.

The previous work [GPS22] follows from [DPSZ12b] and uses information-theoretic MACs to detect the above attack. However, this approach increases the communication complexity by at least a factor of 2 since it requires to compute each multiplication gate 2 times in [GPS22]. Another drawback is that the information-theoretic MAC requires to use a large enough finite field.

Our idea is to try to compute a degree-$t$ Shamir sharing for each wire value as the state-of-the-art MPC protocols [DN07, GIP+14, CGH+18, NV18, GSZ20, BGIN20, GLO+21] in the honest majority setting. However, our approach differs from previous techniques in the sense that the degree-$t$ Shamir sharings may use different evaluation points to store the secrets. We observe that, in our improved semi-honest protocol in Section C, for each group of multiplication gates,

- All parties hold a packed Beaver triple $([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, [\![\boldsymbol{c}]\!]_{n-k})$. In particular, all parties also hold $\{([\![a_i|_i]\!]_t, [\![b_i|_i]\!]_t, [\![c_i|_i]\!]_t)\}_{i=1}^k$ in the circuit-independent preprocessing phase.
- In the online phase, $P_1$ will distribute $[\![\boldsymbol{v_\alpha}+\boldsymbol{a}]\!]_{k-1}, [\![\boldsymbol{v_\beta}+\boldsymbol{b}]\!]_{k-1}$ to all parties. In particular, $[\![\boldsymbol{v_\alpha}+\boldsymbol{a}]\!]_{k-1}$ can be viewed as a degree-$(k-1)$ Shamir sharing $[\![v_{\alpha_i}+a_i|_i]\!]_{k-1}$ for all $i \in \{1,2,\ldots,k\}$.

Thus, all parties can compute a degree-$t$ Shamir sharing $[\![v_{\alpha_i}|_i]\!]_t = [\![v_{\alpha_i}+a_i|_i]\!]_{k-1} - [\![a_i|_i]\!]_t$. In particular, the secret is stored at position $i$.

Therefore, our semi-honest protocol has already allowed all parties to compute a degree-$t$ Shamir sharing for input wires of each multiplication gate. We will show that this is sufficient to verify the correctness of the computation. In the following, we will introduce TurboPack phase by phase. Our principle is to try to use the same semi-honest protocol so that we can achieve the same concrete efficiency as the semi-honest version. We will highlight our changes compared with the semi-honest protocol and explain the reasons.

### D.1 Circuit-Independent Preprocessing Phase

In the circuit-independent preprocessing phase, we make the following two changes:

1. For each packed Beaver triple $([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, [\![\boldsymbol{c}]\!]_{n-k})$, recall that they are computed from $\{([\![a_i|_i]\!]_t, [\![b_i|_i]\!]_t, [\![c_i|_i]\!]_t)\}_{i=1}^k$. All parties will also take $\{([\![a_i|_i]\!]_t, [\![b_i|_i]\!]_t, [\![c_i|_i]\!]_t)\}_{i=1}^k$ as output.
2. For each group of input gates or output gates, all parties will prepare a random degree-$t$ Shamir sharing $[\![r_i|_i]\!]_t$ for all $i \in \{1,2,\ldots,k\}$. In the online phase, these degree-$t$ Shamir sharings allow clients to detect attacks launched by corrupted parties.

We first describe the functionality for the circuit-independent preprocessing phase with malicious security. We allow an adversary to launch two kinds of additive attacks: (1) for each degree-$(n-k)$ packed Shamir sharing $[\![\lambda_\alpha \cdot \mathbf{1}]\!]_{n-k}$, an adversary is allowed to add a constant error (chosen by himself) to the first secret; (2) for each multiplication triple $([\![a_i|_i]\!]_t, [\![b_i|_i]\!]_t, [\![c_i|_i]\!]_t)$, an adversary is allowed to add a constant error (chosen by himself) to the secret $c_i$. The description of the functionality appears in $\mathcal{F}_{\mathsf{PrepIndMal}}$.

**Functionality 6:** $\mathcal{F}_{\mathsf{PrepIndMal}}$

1. **Preparing Random Packed Sharings**: $\mathcal{F}_{\mathsf{PrepIndMal}}$ receives the set of corrupted parties, denoted by $\mathcal{C}orr$. $\mathcal{F}_{\mathsf{PrepIndMal}}$ prepares a random degree-$(n-k)$ packed Shamir sharing in the form of $[\![\lambda_\alpha \cdot \mathbf{1}]\!]_{n-k}$ for each output wire $\alpha$ of an input gate a multiplication gate in $C$ as follows.
   (a) $\mathcal{F}_{\mathsf{PrepIndMal}}$ receives from the adversary a set of shares $\{u_j\}_{j\in\mathcal{C}orr}$ and an additive error $\delta_\alpha$. Let $\boldsymbol{e}_1 = (1, 0, \ldots, 0) \in \mathbb{F}^k$. $\mathcal{F}_{\mathsf{PrepIndMal}}$ samples a random value $\lambda_\alpha$ and computes a degree-$(n-k)$ packed Shamir sharing $[\![\lambda_\alpha \cdot \mathbf{1} + \delta_\alpha \cdot \boldsymbol{e}_1]\!]_{n-k}$ such that for all $P_j \in \mathcal{C}orr$, the $j$-th share of $[\![\lambda_\alpha \cdot \mathbf{1}]\!]_{n-k}$ is $u_j$.
   (b) $\mathcal{F}_{\mathsf{PrepIndMal}}$ distributes the shares of $[\![\lambda_\alpha \cdot \mathbf{1} + \delta_\alpha \cdot \boldsymbol{e}_1]\!]_{n-k}$ to honest parties.
2. **Preparing Packed Beaver Triples**: For each group of $k$ multiplication gates, $\mathcal{F}_{\mathsf{PrepIndMal}}$ prepares a set of Beaver triples $\{([\![a_i|_i]\!]_t, [\![b_i|_i]\!]_t, [\![c_i|_i]\!]_t)\}_{i=1}^k$, which satisfy that $a_i, b_i$ are random values and $c_i = a_i \cdot b_i$. This is done by the following steps.
   (a) For all $i \in \{1, 2, \ldots, k\}$, $\mathcal{F}_{\mathsf{PrepIndMal}}$ receives from the adversary a set of shares $\{(u_{i,j}^{(1)}, u_{i,j}^{(2)}, u_{i,j}^{(3)})\}_{j\in\mathcal{C}orr}$ and an additive error $\eta_i$. $\mathcal{F}_{\mathsf{PrepIndMal}}$ samples two random values $a_i, b_i \in \mathbb{F}$ and computes $c_i = a_i \cdot b_i + \eta_i$. Then $\mathcal{F}_{\mathsf{PrepIndMal}}$ computes three degree-$t$ Shamir sharings $([\![a_i|_i]\!]_t, [\![b_i|_i]\!]_t, [\![c_i|_i]\!]_t)$ such that for all $P_j \in \mathcal{C}orr$, the $j$-th share of $([\![a_i|_i]\!]_t, [\![b_i|_i]\!]_t, [\![c_i|_i]\!]_t)$ is $(u_{i,j}^{(1)}, u_{i,j}^{(2)}, u_{i,j}^{(3)})$.
   (b) For all $i \in \{1, 2, \ldots, k\}$, $\mathcal{F}_{\mathsf{PrepIndMal}}$ distributes the shares of $([\![a_i|_i]\!]_t, [\![b_i|_i]\!]_t, [\![c_i|_i]\!]_t)$ to honest parties.
3. **Preparing Random Masked Sharings for Multiplication Gates**: For each group of $k$ multiplication gates, $\mathcal{F}_{\mathsf{PrepIndMal}}$ prepares three degree-$(n-1)$ packed Shamir sharings of $\mathbf{0} \in \mathbb{F}^k$ as follows.
   (a) $\mathcal{F}_{\mathsf{PrepIndMal}}$ receives from the adversary a set of shares $\{(u_j^{(1)}, u_j^{(2)}, u_j^{(3)})\}_{j\in\mathcal{C}orr}$. $\mathcal{F}_{\mathsf{PrepIndMal}}$ sets $\boldsymbol{o}^{(1)} = \boldsymbol{o}^{(2)} = \boldsymbol{o}^{(3)} = \mathbf{0} \in \mathbb{F}^k$. Then $\mathcal{F}_{\mathsf{PrepIndMal}}$ samples three random degree-$(n-1)$ packed Shamir sharings $[\![\boldsymbol{o}^{(1)}]\!]_{n-1}, [\![\boldsymbol{o}^{(2)}]\!]_{n-1}, [\![\boldsymbol{o}^{(3)}]\!]_{n-1}$ such that for all $P_j \in \mathcal{C}orr$, the $j$-th share of $([\![\boldsymbol{o}^{(1)}]\!]_{n-1}, [\![\boldsymbol{o}^{(2)}]\!]_{n-1}, [\![\boldsymbol{o}^{(3)}]\!]_{n-1})$ is $(u_j^{(1)}, u_j^{(2)}, u_j^{(3)})$.
   (b) $\mathcal{F}_{\mathsf{PrepIndMal}}$ distributes the shares of $([\![\boldsymbol{o}^{(1)}]\!]_{n-1}, [\![\boldsymbol{o}^{(2)}]\!]_{n-1}, [\![\boldsymbol{o}^{(3)}]\!]_{n-1})$ to honest parties.
4. **Preparing Random Sharings for Input and Output Gates**: For each group of $k$ input gates or output gates, $\mathcal{F}_{\mathsf{PrepIndMal}}$ 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 above. $\mathcal{F}_{\mathsf{PrepIndMal}}$ also prepares $k$ random degree-$t$ Shamir sharings $\{[\![r_i|_i]\!]_t\}_{i=1}^k$ in the same way as that for $\{[\![a_i|_i]\!]_t\}_{i=1}^k$.

Now we describe the protocol $\Pi_{\mathsf{PrepIndMal}}$ that realizes $\mathcal{F}_{\mathsf{PrepIndMal}}$. It follows the semi-honest version $\Pi_{\mathsf{PrepInd}}$ and uses the ideal functionality $\mathcal{F}_{\mathsf{SingleMultMal}}$. Here by using a weaker functionality $\mathcal{F}_{\mathsf{SingleMultMal}}$, which allows an additive attack towards the multiplication result, we can instantiate it by the same semi-honest multiplication protocol in [GLO+21]. We refer the readers to [GLO+21] for more details.

Regarding the communication complexity of $\Pi_{\mathsf{PrepIndMal}}$, the only difference compared with the semi-honest protocol $\Pi_{\mathsf{PrepInd}}$ is Step 4, where all parties need to prepare $N_3 \cdot k$ random degree-$t$ Shamir sharings. Here $N_3 \cdot k$ is the number of input and output gates. Since $N_3 \cdot k$ is small compared with the circuit size, the concrete efficiency of $\Pi_{\mathsf{PrepIndMal}}$ remains $10|C| \cdot n + 24|C|$ elements among all parties.

**Functionality 7:** $\mathcal{F}_{\mathsf{SingleMultMal}}$

1. $\mathcal{F}_{\mathsf{SingleMultMal}}$ receives the secret position $i$ from all parties. Let $[\![x|_i]\!]_t, [\![y|_i]\!]_t$ denote the input sharings. $\mathcal{F}_{\mathsf{SingleMultMal}}$ receives from honest parties their shares of $[\![x|_i]\!]_t, [\![y|_i]\!]_t$. Then $\mathcal{F}_{\mathsf{SingleMultMal}}$ reconstructs the secrets $x, y$. $\mathcal{F}_{\mathsf{SingleMultMal}}$ further computes the shares of $[\![x|_i]\!]_t, [\![y|_i]\!]_t$ held by corrupted parties, and sends these shares to the adversary.
2. $\mathcal{F}_{\mathsf{SingleMultMal}}$ receives from the adversary a set of shares $\{z_i\}_{i\in\mathcal{C}orr}$ and an additive error $\eta$.
3. $\mathcal{F}_{\mathsf{SingleMultMal}}$ computes $x \cdot y + \eta$. Based on the secret $z := x \cdot y + \eta$ and the $t$ shares $\{z_i\}_{i\in\mathcal{C}orr}$, $\mathcal{F}_{\mathsf{SingleMultMal}}$ reconstructs the whole sharing $[\![z|_i]\!]_t$ and distributes the shares of $[\![z|_i]\!]_t$ to honest parties.

**Protocol 8:** $\Pi_{\mathsf{PrepIndMal}}$

1. **Preparing Random Packed Sharings**: Let $N_1$ be the number of input gates and output gates. Let $\Sigma_1$ be the secret sharing scheme corresponding to $[\![r \cdot \mathbf{1}]\!]_{n-k}$. All parties invoke $N_1/(t+1)$ times of $\pi_{\mathsf{RandSh}}(\Sigma_1)$ to prepare $N_1$ random sharings in the form of $[\![r \cdot \mathbf{1}]\!]_{n-k}$. For each output wire of an input gate or a multiplication gate, the first unused random sharing is associated with this wire.
2. **Preparing Packed Beaver Triples**: Let $N_2$ denote the number of groups of multiplication gates. For all $i \in \{1, 2, \dots, k\}$, let $\Sigma_{2,i}$ be the secret sharing scheme corresponding to $[\![r|_i]\!]_t$. All parties invoke $2N_2/(t+1)$ times of $\pi_{\mathsf{RandSh}}(\Sigma_{2,i})$ to prepare $2N_2$ random sharings in the form of $[\![r|_i]\!]_t$.
   (a) For each group of multiplication gates, let $\{[\![a_i|_i]\!]_t, [\![b_i|_i]\!]_t\}_{i=1}^k$ be the unused random sharings.
   (b) For all $i \in \{1, 2, \dots, k\}$, all parties invoke $\mathcal{F}_{\mathsf{SingleMultMal}}$ on $(i, [\![a_i|_i]\!]_t, [\![b_i|_i]\!]_t)$ and receive $[\![c_i|_i]\!]_t$.
3. **Preparing Random Masked Sharings for Multiplication Gates**: Let $\Sigma_3$ be the secret sharing scheme corresponding to $[\![\mathbf{0}]\!]_{n-1}$, where $\mathbf{0} = (0, \dots, 0) \in \mathbb{F}^k$. All parties invoke $3N_2/(t+1)$ times of $\pi_{\mathsf{RandSh}}(\Sigma_3)$ to prepare $3N_2$ random sharings in the form of $[\![\mathbf{0}]\!]_{n-1}$. For each group of multiplication gates, the first 3 unused random sharings are associated with these group of multiplication gates.
4. **Preparing Random Sharings for Input and Output Gates**: Let $N_3$ be the number of groups of input gates and output gates. All parties invoke $N_3/(t+1)$ times of $\pi_{\mathsf{RandSh}}(\Sigma_3)$ to prepare $N_3$ random sharings in the form of $[\![\mathbf{0}]\!]_{n-1}$. For each group of input gates or output gates, the first unused random sharing is associated with this group of gates.
   For all $i \in \{1, 2, \dots, k\}$, all parties also invoke $N_3/(t+1)$ times of $\pi_{\mathsf{RandSh}}(\Sigma_{2,i})$ to prepare $N_3$ random sharings in the form of $[\![r|_i]\!]_t$. For each group of input gates or output gates, the first unused random sharing is associated with this group of gates.

**Lemma 4.** *Protocol $\Pi_{\mathsf{PrepIndMal}}$ securely computes $\mathcal{F}_{\mathsf{PrepIndMal}}$ in the $\mathcal{F}_{\mathsf{SingleMultMal}}$-hybrid model against a fully malicious adversary who controls $t$ 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.

*Simulating $\pi_{\mathsf{RandSh}}$.*

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 $P_i$, $\mathcal{S}$ generates a random $\Sigma$-sharing $\boldsymbol{S}^{(i)}$ and sends the shares to corrupted parties. For each corrupted party $P_i$, $\mathcal{S}$ receives the shares of $\boldsymbol{S}^{(i)}$ honest parties. Depending on the sharing scheme used in $\pi_{\mathsf{RandSh}}$, $\mathcal{S}$ sets the sharing $\boldsymbol{S}^{(i)}$ distributed by $P_i$ as follows:
   – For $\Sigma_1$, each $\Sigma_1$-sharing is a degree-$(n-k)$ packed Shamir sharing in the form of $[\![r \cdot \mathbf{1}]\!]_{n-k}$. It requires $n-k+1 = t+k$ shares or secrets to reconstruct the whole sharing. $\mathcal{S}$ has received the shares of honest parties from $P_i$. $\mathcal{S}$ sets the last $k-1$ secrets to be $0$ and reconstructs the whole sharing as $\boldsymbol{S}^{(0)}$, denoted by $[\![\delta^{(i)} \cdot \boldsymbol{e}_1]\!]_{n-k}$. Here $\boldsymbol{e}_1 = (1, 0, \dots, 0) \in \mathbb{F}^k$. Notice that the last $k-1$ secrets have been set to be $0$. The value $\delta^{(i)}$ represents the first secret which can be non-zero. It is also viewed as the additive error of $\boldsymbol{S}^{(i)}$.
   – For $\Sigma_{2,i}$, each $\Sigma_{2,i}$-sharing is a degree-$t$ Shamir sharing in the form of $[\![r|_i]\!]_t$. It requires $t+1$ shares to reconstruct the whole sharing. $\mathcal{S}$ uses the shares of honest parties to reconstruct the whole sharing.
   – For $\Sigma_3$, each $\Sigma_3$-sharing is a degree-$(n-1)$ packed Shamir sharing of $\mathbf{0} \in \mathbb{F}^k$. It requires $n$ shares or secrets to reconstruct the whole sharing. $\mathcal{S}$ has received the shares of honest parties from $P_i$. $\mathcal{S}$ sets the secrets to be $0$ and also sets the shares of the first $n-(t+k+1)$ corrupted parties to be $0$. Then $\mathcal{S}$ reconstructs the whole sharing as $\boldsymbol{S}^{(0)}$.
3. In Step 3, $\mathcal{S}$ computes the shares of corrupted parties for each $\Sigma$-sharing $\boldsymbol{R}^{(i)}$. For $\Sigma_1$, $\mathcal{S}$ also computes the additive error $\delta_i$ for each $\boldsymbol{R}^{(i)}$. Note that $\delta_i$ is a linear combination of the additive errors $\{\delta^{(i)}\}_{i=1}^n$ of $\{\boldsymbol{S}^{(i)}\}_{i=1}^n$. In particular, for each honest party $P_i$, $\delta^{(i)} = 0$, and for each corrupted party $P_i$, $\delta^{(i)}$ is the additive error computed in the last step.
   $\mathcal{S}$ sends the shares of corrupted parties to $\mathcal{F}_{\mathsf{PrepIndMal}}$. For each $\Sigma_1$-sharing, $\mathcal{S}$ also sends the corresponding additive error to $\mathcal{F}_{\mathsf{PrepIndMal}}$.

*Simulating the Main Protocol.*

1. In Step 1, $\mathcal{S}$ simulates $\pi_{\mathsf{RandSh}}(\Sigma_1)$ as described above.

2. In Step 2, $\mathcal{S}$ simulates $\pi_{\mathsf{RandSh}}(\Sigma_{2,i})$ for all $i \in \{1, 2, \ldots, k\}$ as described above. Then, for each pair of $([\![a_i|_i]\!]_t, [\![b_i|_i]\!]_t)$, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{SingleMultMal}}$ and receives from the adversary the shares of $[\![c_i|_i]\!]_t$ of corrupted parties and the additive error $\eta_i$. Finally, for each pair of $([\![a_i|_i]\!]_t, [\![b_i|_i]\!]_t)$, $\mathcal{S}$ sends the shares of $[\![c_i|_i]\!]_t$ of corrupted parties and the additive error $\eta_i$ to $\mathcal{F}_{\mathsf{PrepIndMal}}$.
3. In Step 3, $\mathcal{S}$ simulates $\pi_{\mathsf{RandSh}}(\Sigma_3)$ as described above.
4. In Step 4, $\mathcal{S}$ simulates $\pi_{\mathsf{RandSh}}(\Sigma_{2,i})$ for all $i \in \{1, 2, \ldots, k\}$ and $\pi_{\mathsf{RandSh}}(\Sigma_3)$ as described above.

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

Now, we show that $\mathcal{S}$ perfectly simulate the behaviors of honest parties. We note that the only place where honest parties need to send messages to corrupted parties is in Step 2 of $\pi_{\mathsf{RandSh}}$. The simulator $\mathcal{S}$ honestly generates a random $\Sigma$-sharing as that in the real world. Thus, $\mathcal{S}$ perfectly simulates the behaviors of honest parties.

Then, we show that the output of honest parties in both worlds have the same distribution. For $\pi_{\mathsf{RandSh}}$, we observe that, when the shares of $\{\boldsymbol{S}^{(i)}\}_{i\in\mathcal{H}}$ of corrupted parties are fixed and the shares of $\{\boldsymbol{S}^{(i)}\}_{i\in\mathcal{C}orr}$ of honest parties are also fixed, the shares of $\{\boldsymbol{R}^{(i)}\}_{i=1}^{t+1}$ held by honest parties are *independent* of the shares of $\{\boldsymbol{S}^{(i)}\}_{i\in\mathcal{C}orr}$ of corrupted parties. This is because honest parties can compute their shares of $\{\boldsymbol{R}^{(i)}\}_{i=1}^{t+1}$ by using their shares of $\{\boldsymbol{S}^{(i)}\}_{i=1}^{n}$. Once corrupted parties send out the shares of $\{\boldsymbol{S}^{(i)}\}_{i\in\mathcal{C}orr}$ of honest parties, they can no longer change the shares of $\{\boldsymbol{R}^{(i)}\}_{i=1}^{t+1}$ of honest parties. It suggests that we may think the adversary first chooses the shares of corrupted parties in the same way as $\mathcal{S}$ described above and later on changes the shares of corrupted parties arbitrarily.

Recall that the matrix $\boldsymbol{M}^{\mathrm{T}}$ is a Vandermonde matrix of size $n \times (t+1)$, which satisfies that any $(t+1) \times (t+1)$ sub-matrix of $\boldsymbol{M}^{\mathrm{T}}$ is invertible. Therefore, given the $\Sigma$-sharings prepared by corrupted parties and the shares of corrupted parties, there is a one-to-one map between $\{\boldsymbol{S}^{(i)}\}_{i\in\mathcal{H}}$ and $\{\boldsymbol{R}^{(i)}\}_{i=1}^{t+1}$. Note that for $\{\boldsymbol{S}^{(i)}\}_{i\in\mathcal{H}}$, they are generated by honest parties. Therefore, $\{\boldsymbol{S}^{(i)}\}_{i\in\mathcal{H}}$ are $n - t = t + 1$ random $\Sigma$-sharings given the shares of corrupted parties.

– When $\Sigma = \Sigma_1$, $\{\boldsymbol{R}^{(i)}\}_{i=1}^{t+1}$ are random $\Sigma_1$-sharings with additive errors $\{\delta_i\}_{i=1}^{t+1}$ given the shares of corrupted parties. In the ideal world, $\mathcal{S}$ sends the shares of corrupted parties and the additive errors $\{\delta_i\}_{i=1}^{t+1}$ to $\mathcal{F}_{\mathsf{PrepIndMal}}$. Therefore, the random $\Sigma_1$-sharings have the same distribution in both worlds.
– When $\Sigma = \Sigma_{2,i}$, $\{\boldsymbol{R}^{(i)}\}_{i=1}^{t+1}$ are random $\Sigma_{2,i}$-sharings given the shares of corrupted parties. In the ideal world, $\mathcal{S}$ sends the shares of corrupted parties to $\mathcal{F}_{\mathsf{PrepIndMal}}$. Therefore, the random $\Sigma_{2,i}$-sharings have the same distribution in both worlds.
– When $\Sigma = \Sigma_3$, $\{\boldsymbol{R}^{(i)}\}_{i=1}^{t+1}$ are random $\Sigma_3$-sharings given the shares of corrupted parties. In the ideal world, $\mathcal{S}$ sends the shares of corrupted parties to $\mathcal{F}_{\mathsf{PrepIndMal}}$. Therefore, the random $\Sigma_3$-sharings have the same distribution in both worlds.

In Step 2, we have shown that $([\![a_i|_i]\!]_t, [\![b_i|_i]\!]_t)$ has the same distribution in both worlds. It is sufficient to show that $[\![c_i|_i]\!]_t$ is also identically distributed in both worlds. Observe that a degree-$t$ Shamir sharing is determined by its secret and the shares of corrupted parties. In the real world, the secret $c_i = a_i \cdot b_i + \eta_i$, where $\eta_i$ is an additive error provided by the adversary in $\mathcal{F}_{\mathsf{SingleMultMal}}$. The shares of $[\![c_i|_i]\!]_t$ of corrupted parties are also chosen by the adversary in $\mathcal{F}_{\mathsf{SingleMultMal}}$. In the ideal world, $\mathcal{S}$ receives both $\eta_i$ and the shares of corrupted parties when emulating $\mathcal{F}_{\mathsf{SingleMultMal}}$. Then $\mathcal{S}$ sends these values to $\mathcal{F}_{\mathsf{PrepIndMal}}$. Thus, $[\![c_i|_i]\!]_t$ is identically distributed in both worlds.

We conclude that Protocol $\Pi_{\mathsf{PrepIndMal}}$ securely computes $\mathcal{F}_{\mathsf{PrepIndMal}}$ in the $\mathcal{F}_{\mathsf{SingleMultMal}}$-hybrid model against a semi-honest adversary who controls $t$ parties.

## D.2 Circuit-Dependent Preprocessing Phase

In the circuit-dependent preprocessing phase, we follow the improved version of $\Pi_{\mathsf{Prep}}$ described in Section C with the following change:

– For each group of input gates or output gates, all parties will also output $\{[\![r_i|_i]\!]_t\}_{i=1}^{k}$ prepared in $\mathcal{F}_{\mathsf{PrepIndMal}}$.
– For each group of multiplication gates, all parties will output $\{([\![a_i|_i]\!]_t, [\![b_i|_i]\!]_t, [\![c_i|_i]\!]_t)\}_{i=1}^{t}$ instead of $([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, [\![\boldsymbol{c}]\!]_{n-k})$.
– For each group of output gates with input wires $\boldsymbol{\alpha}$, all parties will reconstruct the vector $\boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \boldsymbol{r}$ to $P_1$. Here $\boldsymbol{r} = (r_1, \ldots, r_k)$ and $r_1, \ldots, r_k$ are the secrets of $\{[\![r_i|_i]\!]_t\}_{i=1}^{k}$ prepared for these output gates.

We first describe the functionality for the circuit-dependent preprocessing phase with malicious security. We allow an adversary to launch two kinds of additive attacks: (1) for each degree-$(n-1)$ packed Shamir sharing $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}]\!]_{n-1}$, an adversary is allowed to add a vector of additive errors $\Delta(\boldsymbol{\lambda}_{\boldsymbol{\alpha}})$ (chosen by himself) to the secrets; (2) for each multiplication triple $([\![a_i|_i]\!]_t, [\![b_i|_i]\!]_t, [\![c_i|_i]\!]_t)$, an adversary is allowed to add a constant error (chosen by himself) to the secret $c_i$. We note that each wire $\alpha$ in the circuit connects two gates, and it acts as an output wire of the first gate and acts as an input wire of the second gate.

**Functionality 8: $\mathcal{F}_{\mathsf{PrepMal}}$**

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 $\alpha$ of an input gate or a multiplication gate, $\mathcal{F}_{\mathsf{PrepMal}}$ samples a uniform value $\lambda_\alpha$ and associates it with the wire $\alpha$.
   (b) Starting from the first layer of $C$ to the last layer, for each addition gate with input wires $\alpha, \beta$ and output wire $\gamma$, $\mathcal{F}_{\mathsf{PrepMal}}$ sets $\lambda_\gamma = \lambda_\alpha + \lambda_\beta$.
2. **Preparing Beaver Triples**: $\mathcal{F}_{\mathsf{PrepMal}}$ receives the set of corrupted parties, denoted by $\mathcal{C}orr$. For each group of $k$ multiplication gates, $\mathcal{F}_{\mathsf{PrepMal}}$ prepares a set of Beaver triples $\{([\![a_i|_i]\!]_t, [\![b_i|_i]\!]_t, [\![c_i|_i]\!]_t)\}_{i=1}^k$, which satisfy that $a_i, b_i$ are random values and $c_i = a_i \cdot b_i$. This is done by the following steps.
   (a) For all $i \in \{1, 2, \ldots, k\}$, $\mathcal{F}_{\mathsf{PrepMal}}$ receives from the adversary a set of shares $\{(u_{i,j}^{(1)}, u_{i,j}^{(2)}, u_{i,j}^{(3)})\}_{j \in \mathcal{C}orr}$ and an additive error $\eta_i$. $\mathcal{F}_{\mathsf{PrepMal}}$ samples two random values $a_i, b_i \in \mathbb{F}$ and computes $c_i = a_i \cdot b_i + \eta_i$. Then $\mathcal{F}_{\mathsf{PrepMal}}$ computes three degree-$t$ Shamir sharings $([\![a_i|_i]\!]_t, [\![b_i|_i]\!]_t, [\![c_i|_i]\!]_t)$ such that for all $P_j \in \mathcal{C}orr$, the $j$-th share of $([\![a_i|_i]\!]_t, [\![b_i|_i]\!]_t, [\![c_i|_i]\!]_t)$ is $(u_{i,j}^{(1)}, u_{i,j}^{(2)}, u_{i,j}^{(3)})$.
   (b) For all $i \in \{1, 2, \ldots, k\}$, $\mathcal{F}_{\mathsf{PrepMal}}$ distributes the shares of $([\![a_i|_i]\!]_t, [\![b_i|_i]\!]_t, [\![c_i|_i]\!]_t)$ to honest parties.
3. **Preparing Degree-$t$ Shamir Sharings**: For each group of $k$ input gates or output gates, $\mathcal{F}_{\mathsf{PrepMal}}$ prepares a set of random degree-$t$ Shamir sharings $\{[\![r_i|_i]\!]_t\}_{i=1}^k$ as follows.
   (a) For all $i \in \{1, 2, \ldots, k\}$, $\mathcal{F}_{\mathsf{PrepMal}}$ receives from the adversary a set of shares $\{u_{i,j}\}_{j \in \mathcal{C}orr}$. $\mathcal{F}_{\mathsf{PrepMal}}$ samples a random value $r_i \in \mathbb{F}$. Then $\mathcal{F}_{\mathsf{PrepMal}}$ computes a degree-$t$ Shamir sharings $[\![r_i|_i]\!]_t$ such that for all $P_j \in \mathcal{C}orr$, the $j$-th share of $[\![r_i|_i]\!]_t$ is $u_{i,j}$.
   (b) For all $i \in \{1, 2, \ldots, k\}$, $\mathcal{F}_{\mathsf{PrepMal}}$ distributes the shares of $[\![r_i|_i]\!]_t$ to honest parties.
4. **Distributing $\boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \boldsymbol{a}$ 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 $\boldsymbol{a} = (a_1, a_2, \ldots, a_k)$ and $\boldsymbol{b} = (b_1, b_2, \ldots, b_k)$. $\mathcal{F}_{\mathsf{PrepMal}}$ receives from the adversary two vectors $\Delta(\boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \boldsymbol{a}), \Delta(\boldsymbol{\lambda}_{\boldsymbol{\beta}} + \boldsymbol{b}) \in \mathbb{F}^k$. Then, $\mathcal{F}_{\mathsf{PrepMal}}$ computes $\boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \boldsymbol{a} + \Delta(\boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \boldsymbol{a})$ and $\boldsymbol{\lambda}_{\boldsymbol{\beta}} + \boldsymbol{b} + \Delta(\boldsymbol{\lambda}_{\boldsymbol{\beta}} + \boldsymbol{b})$, and sends them to $P_1$. Here $\boldsymbol{\lambda}_{\boldsymbol{\alpha}}$ and $\boldsymbol{\lambda}_{\boldsymbol{\beta}}$ are the random values associated with the wires $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$.
   For each group of output gates, let $\boldsymbol{\alpha}$ denote the batch of input wires of these gates. Recall that $\mathcal{F}_{\mathsf{PrepMal}}$ has prepared $\{[\![r_i|_i]\!]_t\}_{i=1}^k$. Let $\boldsymbol{r} = (r_1, \ldots, r_k)$. $\mathcal{F}_{\mathsf{PrepMal}}$ receives from the adversary a vector $\Delta(\boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \boldsymbol{r}) \in \mathbb{F}^k$. Then, $\mathcal{F}_{\mathsf{PrepMal}}$ computes $\boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \boldsymbol{r} + \Delta(\boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \boldsymbol{r})$ and sends these values to $P_1$. Here $\boldsymbol{\lambda}_{\boldsymbol{\alpha}}$ are the random values associated with the wires $\boldsymbol{\alpha}$.
5. **Preparing Degree-$(n-1)$ Packed Shamir Sharings**: $\mathcal{F}_{\mathsf{PrepMal}}$ will prepare degree-$(n-1)$ packed Shamir sharings for the following batches of wires:
   – For the input layer, all input gates are divided into groups of size $k$ such that the input gates of each group belong to the same client. For each group of input gates with output wires $\boldsymbol{\alpha}$, $\mathcal{F}_{\mathsf{PrepMal}}$ will prepare a degree-$(n-1)$ packed Shamir sharing of $\boldsymbol{\lambda}_{\boldsymbol{\alpha}}$.
   – For each group of multiplication gates with output wires $\boldsymbol{\gamma}$, $\mathcal{F}_{\mathsf{PrepMal}}$ will prepare a degree-$(n-1)$ packed Shamir sharing of $\boldsymbol{\lambda}_{\boldsymbol{\gamma}}$.

   For each batch of wires $\boldsymbol{\alpha}$ in the above scenarios, $\mathcal{F}_{\mathsf{PrepMal}}$ does the following: $\mathcal{F}_{\mathsf{PrepMal}}$ receives from the adversary a set of shares $\{u_j\}_{j \in \mathcal{C}orr}$. $\mathcal{F}_{\mathsf{PrepMal}}$ receives from the adversary a vector $\Delta(\boldsymbol{\lambda}_{\boldsymbol{\alpha}})$. $\mathcal{F}_{\mathsf{PrepMal}}$ samples a random degree-$(n-1)$ packed Shamir sharing $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \Delta(\boldsymbol{\lambda}_{\boldsymbol{\alpha}})]\!]_{n-1}$ such that for all $P_j \in \mathcal{C}orr$, the $j$-th share of $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \Delta(\boldsymbol{\lambda}_{\boldsymbol{\alpha}})]\!]_{n-1}$ is $u_j$. Then, $\mathcal{F}_{\mathsf{PrepMal}}$ distributes the shares of $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \Delta(\boldsymbol{\lambda}_{\boldsymbol{\alpha}})]\!]_{n-1}$ to honest parties.

Now we describe the protocol $\Pi_{\mathsf{PrepMal}}$ that realizes $\mathcal{F}_{\mathsf{PrepMal}}$. The communication complexity of $\Pi_{\mathsf{PrepMal}}$ remains the same as its semi-honest version described in Section C, i.e., $8|C|$ elements among all parties.

**Protocol 9: $\Pi_{\mathsf{PrepMal}}$**

1. **Circuit-Independent Preprocessing Phase**: All parties invoke $\mathcal{F}_{\mathsf{PrepIndMal}}$ to receive correlated randomness.

2. **Computing a Random Sharing for Each Wire**: For each output wire $\alpha$ of an input gate or a multiplication gate, all parties receive $[\![\lambda_\alpha \cdot \mathbf{1}]\!]_{n-k}$ from $\mathcal{F}_{\mathsf{PrepInd}}$. All parties follow Step 1 of $\mathcal{F}_{\mathsf{Prep}}$ and compute $[\![\lambda_\alpha \cdot \mathbf{1}]\!]_{n-k}$ for each wire $\alpha$ in the circuit $C$.
3. **Preparing Beaver Triples**: For each group of multiplication gates, all parties output $\{([\![a_i|_i]\!]_t, [\![b_i|_i]\!]_t, [\![c_i|_i]\!]_t)\}_{i=1}^k$, which are prepared in $\mathcal{F}_{\mathsf{PrepIndMal}}$.
4. **Preparing Degree-$t$ Shamir Sharings**: For each group of input gates or output gates, all parties output $\{[\![r_i|_i]\!]_t\}_{i=1}^k$, which are prepared in $\mathcal{F}_{\mathsf{PrepIndMal}}$.
5. **Reconstructing $\boldsymbol{\lambda_\alpha} + \boldsymbol{a}$ to $P_1$**: For each group of multiplication gates with input wires $\boldsymbol{\alpha} = (\alpha_1, \dots, \alpha_k)$, $\boldsymbol{\beta} = (\beta_1, \dots, \beta_k)$, recall that all parties have computed $\{[\![\lambda_{\alpha_i} \cdot \mathbf{1}]\!]\}_{i=1}^k$ and $\{[\![\lambda_{\beta_i} \cdot \mathbf{1}]\!]\}_{i=1}^k$ in the last step. All parties also receive from $\mathcal{F}_{\mathsf{PrepIndMal}}$ $\{([\![a_i|_i]\!]_t, [\![b_i|_i]\!]_t, [\![c_i|_i]\!]_t)\}_{i=1}^k$ and $[\![\boldsymbol{o}^{(1)}]\!]_{n-1}, [\![\boldsymbol{o}^{(2)}]\!]_{n-1}$. Let $\boldsymbol{a} = (a_1, \dots, a_k)$ and $\boldsymbol{b} = (b_1, \dots, b_k)$. Let $\boldsymbol{e}_i \in \mathbb{F}^k$ be the $i$-th unit vector, i.e., all entries of $\boldsymbol{e}_i$ are 0 except the $i$-th entry is 1.
   (a) All parties locally compute
$$[\![\boldsymbol{\lambda_\alpha} + \boldsymbol{a}]\!]_{n-1} = \sum_{i=1}^k \boldsymbol{e}_i * [\![\lambda_{\alpha_i} \cdot \mathbf{1}]\!]_{n-k} + \sum_{i=1}^k [\![\boldsymbol{e}_i]\!]_{k-1} * [\![a_i|_i]\!]_t + [\![\boldsymbol{o}^{(1)}]\!]_{n-1}$$
$$[\![\boldsymbol{\lambda_\beta} + \boldsymbol{b}]\!]_{n-1} = \sum_{i=1}^k \boldsymbol{e}_i * [\![\lambda_{\beta_i} \cdot \mathbf{1}]\!]_{n-k} + \sum_{i=1}^k [\![\boldsymbol{e}_i]\!]_{k-1} * [\![b_i|_i]\!]_t + [\![\boldsymbol{o}^{(2)}]\!]_{n-1}$$
   (b) $P_1$ collects the whole sharings $[\![\boldsymbol{\lambda_\alpha} + \boldsymbol{a}]\!]_{n-1}, [\![\boldsymbol{\lambda_\beta} + \boldsymbol{b}]\!]_{n-1}$ and reconstructs the secrets $\boldsymbol{\lambda_\alpha} + \boldsymbol{a}, \boldsymbol{\lambda_\beta} + \boldsymbol{b}$.

   The same step is also done for each group of output gates with input wires $\boldsymbol{\alpha}$ by using $\{[\![r_i|_i]\!]_t\}_{i=1}^k$ and $[\![\boldsymbol{o}]\!]_{n-1}$ prepared in $\mathcal{F}_{\mathsf{PrepIndMal}}$. As a result, $P_1$ receives $\boldsymbol{\lambda_\alpha} + \boldsymbol{r}$.
6. **Preparing Degree-$(n-1)$ Packed Shamir Sharings**: For each group of input gates, let $\boldsymbol{\alpha} = (\alpha_1, \dots, \alpha_k)$ be the output wires of these gates. Recall that all parties have computed $\{[\![\lambda_{\alpha_i} \cdot \mathbf{1}]\!]\}_{i=1}^k$. All parties also receive from $\mathcal{F}_{\mathsf{PrepIndMal}}$ $[\![\boldsymbol{o}]\!]_{n-1}$. All parties locally compute
$$[\![\boldsymbol{\lambda_\alpha}]\!]_{n-1} = \sum_{i=1}^k \boldsymbol{e}_i * [\![\lambda_{\alpha_i} \cdot \mathbf{1}]\!]_{n-k} + [\![\boldsymbol{o}]\!]_{n-1}.$$
   The same step is also done for the output wires of each group of multiplication gates (by using $[\![\boldsymbol{o}^{(3)}]\!]_{n-1}$ prepared for this group of multiplication gates in $\mathcal{F}_{\mathsf{PrepIndMal}}$).

**Lemma 5.** *Protocol $\Pi_{\mathsf{PrepMal}}$ securely computes $\mathcal{F}_{\mathsf{PrepMal}}$ in the $\mathcal{F}_{\mathsf{PrepIndMal}}$-hybrid model against a fully malicious adversary who controls $t$ 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.

1. In Step 1, $\mathcal{S}$ emulates the ideal functionality $\mathcal{F}_{\mathsf{PrepIndMal}}$.
2. In Step 2, for each output wire $\alpha$ of an input gate or a multiplication gate, $\mathcal{S}$ receives from the adversary the shares of $[\![\lambda_\alpha \cdot \mathbf{1}]\!]_{n-k}$ of corrupted parties and an additive error $\delta_\alpha \cdot \boldsymbol{e}_1$ when emulating $\mathcal{F}_{\mathsf{PrepIndMal}}$. Here $\boldsymbol{e}_1 = (1, 0, \dots, 0) \in \mathbb{F}^k$. Recall that in $\mathcal{F}_{\mathsf{PrepIndMal}}$, the adversary can only add a constant error to the first secret of $[\![\lambda_\alpha \cdot \mathbf{1}]\!]_{n-k}$.
   Then $\mathcal{S}$ follows the protocol. For each wire $\alpha$, $\mathcal{S}$ computes the shares of $[\![\lambda_\alpha \cdot \mathbf{1}]\!]_{n-k}$ of corrupted parties and the corresponding error $\delta_\alpha \cdot \boldsymbol{e}_1$.
3. In Step 3, for each multiplication triple $([\![a_i|_i]\!]_t, [\![b_i|_i]\!]_t, [\![c_i|_i]\!]_t)$, $\mathcal{S}$ receives from the adversary the shares of corrupted parties and the additive error $\eta_i$ when emulating $\mathcal{F}_{\mathsf{PrepIndMal}}$. $\mathcal{S}$ sends these values to $\mathcal{F}_{\mathsf{PrepMal}}$.
4. In Step 4, for each degree-$t$ Shamir sharing $[\![r_i|_i]\!]_t$, $\mathcal{S}$ receives from the adversary the shares of corrupted parties when emulating $\mathcal{F}_{\mathsf{PrepIndMal}}$. $\mathcal{S}$ sends the shares of corrupted parties in $\mathcal{F}_{\mathsf{PrepMal}}$.
5. In Step 5, for each group of multiplication gates with input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$, $\mathcal{S}$ follows the protocol and computes the shares of $[\![\boldsymbol{\lambda_\alpha} + \boldsymbol{a}]\!]_{n-1}$ and $[\![\boldsymbol{\lambda_\beta} + \boldsymbol{b}]\!]_{n-1}$ held by corrupted parties.
   – If $P_1$ is an honest party, $\mathcal{S}$ receives from the adversary the shares of $[\![\boldsymbol{\lambda_\alpha} + \boldsymbol{a}]\!]_{n-1}$ and $[\![\boldsymbol{\lambda_\beta} + \boldsymbol{b}]\!]_{n-1}$ of corrupted parties, which can be different from those computed by $\mathcal{S}$. Let $[\![\overline{\boldsymbol{\lambda_\alpha} + \boldsymbol{a}}]\!]_{n-1}$ denote the degree-$(n-1)$ packed Shamir sharing where the shares of corrupted parties are those computed by $\mathcal{S}$. $\mathcal{S}$ computes the shares of
$$[\![\rho(\boldsymbol{\lambda_\alpha} + \boldsymbol{a})]\!]_{n-1} = [\![\boldsymbol{\lambda_\alpha} + \boldsymbol{a}]\!]_{n-1} - [\![\overline{\boldsymbol{\lambda_\alpha} + \boldsymbol{a}}]\!]_{n-1}$$

of corrupted parties and sets the shares of honest parties to be all $0$. Then, $\mathcal{S}$ reconstructs the secrets $\rho(\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a})$. Finally, $\mathcal{S}$ sets $\Delta(\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a})=\rho(\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a})+\delta_{\alpha_1}\cdot\boldsymbol{e}_1$. $\mathcal{S}$ sends to $\mathcal{F}_{\mathsf{PrepMal}}$ the vector of additive errors $\Delta(\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a})$.
  Similarly, $\mathcal{S}$ computes the vector of additive errors $\Delta(\boldsymbol{\lambda}_{\boldsymbol{\beta}}+\boldsymbol{b})$ and sends it to $\mathcal{F}_{\mathsf{PrepMal}}$.
- If $P_1$ is corrupted, $\mathcal{S}$ sets $\Delta(\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a})=\Delta(\boldsymbol{\lambda}_{\boldsymbol{\beta}}+\boldsymbol{b})=\boldsymbol{0}\in\mathbb{F}^k$. Then $\mathcal{S}$ sends to $\mathcal{F}_{\mathsf{PrepMal}}$ the two vectors of additive errors and receives $\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a},\boldsymbol{\lambda}_{\boldsymbol{\beta}}+\boldsymbol{b}$. $\mathcal{S}$ generates a random degree-$(n-1)$ packed Shamir sharing $[\![\overline{\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a}}]\!]_{n-1}$ based on the secrets $\overline{\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a}}=\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a}+\delta_{\alpha_1}\cdot\boldsymbol{e}_1$ and the shares of corrupted parties computed by $\mathcal{S}$. Finally, $\mathcal{S}$ sends the shares of $[\![\overline{\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a}}]\!]_{n-1}$ of honest parties to $P_1$.
  Similarly, $\mathcal{S}$ generates a random degree-$(n-1)$ packed Shamir sharing $[\![\overline{\boldsymbol{\lambda}_{\boldsymbol{\beta}}+\boldsymbol{b}}]\!]_{n-1}$ based on the secrets $\overline{\boldsymbol{\lambda}_{\boldsymbol{\beta}}+\boldsymbol{b}}=\boldsymbol{\lambda}_{\boldsymbol{\beta}}+\boldsymbol{b}+\delta_{\beta_1}\cdot\boldsymbol{e}_1$ and the shares of corrupted parties computed by $\mathcal{S}$. $\mathcal{S}$ sends the shares of $[\![\overline{\boldsymbol{\lambda}_{\boldsymbol{\beta}}+\boldsymbol{b}}]\!]_{n-1}$ of honest parties to $P_1$.

 For each group of output gates with input wires $\boldsymbol{\alpha}$, $\mathcal{S}$ simulates honest parties in the same way when reconstructing $\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{r}$ to $P_1$.
6. In Step 5, $\mathcal{S}$ follows the protocol and computes the shares of $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}]\!]_{n-1}$ of corrupted parties. Then, $\mathcal{S}$ sets $\Delta(\boldsymbol{\lambda}_{\boldsymbol{\alpha}})=\delta_{\alpha_1}\cdot\boldsymbol{e}_1$. $\mathcal{S}$ sends the shares of $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}]\!]_{n-1}$ of corrupted parties computed by $\mathcal{S}$ and the vector of additive errors $\Delta(\boldsymbol{\lambda}_{\boldsymbol{\alpha}})$ to $\mathcal{F}_{\mathsf{PrepMal}}$.

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

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

**Hybrid$_0$**: In this hybrid, $\mathcal{S}$ honestly follows the protocol.

**Hybrid$_1$**: In this hybrid, for each wire $\alpha$, $\mathcal{S}$ computes the shares of $[\![\lambda_\alpha\cdot\boldsymbol{1}]\!]_{n-k}$ of corrupted parties and the corresponding error $\delta_\alpha\cdot\boldsymbol{e}_1$. Note that this hybrid does not change the behaviors of honest parties. Therefore, the distribution of **Hybrid$_1$** is identical to that of **Hybrid$_0$**.

**Hybrid$_2$**: In this hybrid, Step 3 is simulated by $\mathcal{S}$ as described above. In **Hybrid$_1$**, each multiplication triple $([\![a_i|_i]\!]_t,[\![b_i|_i]\!]_t,[\![c_i|_i]\!]_t)$ is generated by $\mathcal{F}_{\mathsf{PrepIndMal}}$. In particular, corrupted parties choose their shares and the additive error $\eta_i$. In **Hybrid$_2$**, $\mathcal{S}$ provides the shares of corrupted parties and the additive error to $\mathcal{F}_{\mathsf{PrepMal}}$. Then $\mathcal{F}_{\mathsf{PrepMal}}$ generates $([\![a_i|_i]\!]_t,[\![b_i|_i]\!]_t,[\![c_i|_i]\!]_t)$ in the same way as that in $\mathcal{F}_{\mathsf{PrepIndMal}}$. Therefore, the distribution of **Hybrid$_2$** is identical to that of **Hybrid$_1$**.

**Hybrid$_3$**: In this hybrid, Step 4 is simulated by $\mathcal{S}$ as described above. In **Hybrid$_2$**, each degree-$t$ Shamir sharing $[\![r_i|_i]\!]_t$ is generated by $\mathcal{F}_{\mathsf{PrepIndMal}}$. In particular, corrupted parties choose their shares. In **Hybrid$_3$**, $\mathcal{S}$ provides the shares of corrupted parties to $\mathcal{F}_{\mathsf{PrepMal}}$. Then $\mathcal{F}_{\mathsf{PrepMal}}$ generates $[\![r_i|_i]\!]_t$ in the same way as that in $\mathcal{F}_{\mathsf{PrepIndMal}}$. Therefore, the distribution of **Hybrid$_3$** is identical to that of **Hybrid$_2$**.

**Hybrid$_4$**: In this hybrid, Step 5 is simulated by $\mathcal{S}$ when $P_1$ is honest. In **Hybrid$_3$**, $P_1$ reconstructs $\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a}$ by using the shares of $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a}]\!]_{n-1}$ he received from all parties. Note that honest parties always send the correct shares to $P_1$. Observe the following two facts.

- Let $[\![\overline{\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a}}]\!]_{n-1}$ denote the degree-$(n-1)$ packed Shamir sharing where the shares of corrupted parties are those computed by $\mathcal{S}$. This is to distinguish the degree-$(n-1)$ packed Shamir sharing $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a}]\!]_{n-1}$ that $P_1$ receives from all parties. We claim that the secrets $\overline{\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a}}$ are the correct secrets $\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a}$ plus a constant vector $\delta_{\alpha_1}\cdot\boldsymbol{e}_1$.
  This is because in $\mathcal{F}_{\mathsf{PrepIndMal}}$, for each output wire $\alpha$ of an input gate or a multiplication gate, an adversary has added an error $\delta_\alpha$ to the first secret of $[\![\lambda_\alpha\cdot\boldsymbol{1}]\!]_{n-k}$, i.e., a vector of additive errors $\delta_\alpha\cdot\boldsymbol{e}_1$. The additive errors propagate to the secrets of $[\![\lambda_\alpha\cdot\boldsymbol{1}]\!]_{n-k}$ for other wires $\alpha$ in the circuit in Step 2. Following the equation that computes $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a}]\!]_{n-1}$ in Step 4, the secrets are equal to the correct secrets $\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a}$ plus $\delta_{\alpha_1}\cdot\boldsymbol{e}_1$. (The rest of errors $\delta_{\alpha_i}\cdot\boldsymbol{e}_1$ are zeroed out when multiplying $\boldsymbol{e}_i$).
- In **Hybrid$_3$**, $P_1$ reconstructs the secrets of $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a}]\!]_{n-1}$. By the linearity of the packed Shamir secret sharing scheme, the secrets of

$$[\![\rho(\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a})]\!]_{n-1}=[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a}]\!]_{n-1}-[\![\overline{\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a}}]\!]_{n-1}$$

  are the additive errors to the secrets $\overline{\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a}}$ due to the malicious behaviors of corrupted parties. Thus, the secrets of $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a}]\!]_{n-1}$ are equal to $\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a}$ plus $\delta_{\alpha_1}\cdot\boldsymbol{e}_1+\rho(\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a})$. That is, $\Delta(\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a})=\delta_{\alpha_1}\cdot\boldsymbol{e}_1+\rho(\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a})$.

In **Hybrid$_4$**, $\mathcal{S}$ has computed $\delta_\alpha$ for all wires in **Hybrid$_1$**. Thus, $\mathcal{S}$ learns $\delta_{\alpha_1}\cdot\boldsymbol{e}_1$. $\mathcal{S}$ can also learn the shares of $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a}]\!]_{n-1}$ and $[\![\overline{\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a}}]\!]_{n-1}$ held by corrupted parties. Thus, $\mathcal{S}$ can compute the shares of $[\![\rho(\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a})]\!]_{n-1}$ of corrupted parties. Also note that for both $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a}]\!]_{n-1}$ and $[\![\overline{\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a}}]\!]_{n-1}$, the shares of honest parties are identical. Therefore, the shares of $[\![\rho(\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a})]\!]_{n-1}$ of honest parties are all $0$. In this way, $\mathcal{S}$ reconstructs the secrets $\rho(\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a})$. Thus, $\mathcal{S}$ computes and sends the vector of additive errors $\Delta(\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a}) = \delta_{\alpha_1}\cdot\boldsymbol{e}_1 + \rho(\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a})$ to $\mathcal{F}_{\mathsf{PrepMal}}$, which has the same distribution as that in **Hybrid**$_3$.

The same argument works for $[\![\boldsymbol{\lambda}_{\boldsymbol{\beta}}+\boldsymbol{b}]\!]_{n-1}$ for the other batch of input wires of each group of multiplication gates, and works for $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{r}]\!]_{n-1}$ for the batch of input wires of each group of output gates.

**Hybrid**$_5$: In this hybrid, Step 5 is emulated by $\mathcal{S}$ when $P_1$ is corrupted. In **Hybrid**$_4$, honest parties need to send their shares of $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a}]\!]_{n-1}$ to corrupted parties. As we have argued above, the shares of $[\![\overline{\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a}}]\!]_{n-1}$ held by honest parties are equal to the shares of $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a}]\!]_{n-1}$ of honest parties. Also, the secrets of $[\![\overline{\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a}}]\!]_{n-1}$ are equal to the correct secrets $\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a}$ plus $\delta_{\alpha_1}\cdot\boldsymbol{e}_1$. Furthermore, since $[\![\boldsymbol{o}^{(1)}]\!]_{n-1}$ is a random degree-$(n-1)$ packed Shamir sharing of $\mathbf{0}\in\mathbb{F}^k$, $[\![\overline{\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a}}]\!]_{n-1}$ is a random degree-$(n-1)$ packed Shamir sharing of $\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a}+\delta_{\alpha_1}\cdot\boldsymbol{e}_1$.

In **Hybrid**$_5$, $\mathcal{S}$ learns $\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a}$ from $\mathcal{F}_{\mathsf{PrepMal}}$ and has computed $\delta_{\alpha_1}\cdot\boldsymbol{e}_1$ in **Hybrid**$_1$. $\mathcal{S}$ generates a random degree-$(n-1)$ packed Shamir sharing as $[\![\overline{\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a}}]\!]_{n-1}$ based on the secrets $\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a}+\delta_{\alpha_1}\cdot\boldsymbol{e}_1$ and the shares of corrupted parties computed by $\mathcal{S}$. Then $\mathcal{S}$ sends the shares of $[\![\overline{\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a}}]\!]_{n-1}$ of honest parties to $P_1$, which have the same distribution as that in **Hybrid**$_4$.

The same argument works for $[\![\boldsymbol{\lambda}_{\boldsymbol{\beta}}+\boldsymbol{b}]\!]_{n-1}$ for the other batch of input wires of each group of multiplication gates, and works for $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{r}]\!]_{n-1}$ for the batch of input wires of each group of output gates.

**Hybrid**$_6$: In this hybrid, Step 6 is emulated by $\mathcal{S}$. Let $[\![\overline{\boldsymbol{\lambda}_{\boldsymbol{\alpha}}}]\!]_{n-1}$ denote the degree-$(n-1)$ packed Shamir sharing where the shares of corrupted parties are those computed by $\mathcal{S}$. In **Hybrid**$_5$, following a similar argument, the secrets of $[\![\overline{\boldsymbol{\lambda}_{\boldsymbol{\alpha}}}]\!]_{n-1}$ are equal to the correct secrets $\boldsymbol{\lambda}_{\boldsymbol{\alpha}}$ plus $\delta_{\alpha_1}\cdot\boldsymbol{e}_1$. And $[\![\overline{\boldsymbol{\lambda}_{\boldsymbol{\alpha}}}]\!]_{n-1}$ is a random degree-$(n-1)$ packed Shamir sharing of $\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\delta_{\alpha_1}\cdot\boldsymbol{e}_1$.

In **Hybrid**$_6$, $\mathcal{S}$ sends to $\mathcal{F}_{\mathsf{PrepMal}}$ the shares of $[\![\overline{\boldsymbol{\lambda}_{\boldsymbol{\alpha}}}]\!]_{n-1}$ of corrupted parties and the vector of additive errors $\Delta(\boldsymbol{\lambda}_{\boldsymbol{\alpha}}) = \delta_{\alpha_1}\cdot\boldsymbol{e}_1$. $\mathcal{F}_{\mathsf{PrepMal}}$ generates a random degree-$(n-1)$ packed Shamir sharing of $\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\delta_{\alpha_1}\cdot\boldsymbol{e}_1$ based on the shares of corrupted parties. Therefore, the shares of honest parties generated by $\mathcal{F}_{\mathsf{PrepMal}}$ has the same distribution as that in **Hybrid**$_5$.

Observe that **Hybrid**$_6$ is the execution in the ideal world. Therefore, $\Pi_{\mathsf{PrepMal}}$ securely computes $\mathcal{F}_{\mathsf{PrepMal}}$ in the $\mathcal{F}_{\mathsf{PrepInd}}$-hybrid model against a fully malicious adversary who controls $t$ corrupted parties.

## D.3 Online Phase — Evaluation

In the online phase, our goal is to compute degree-$t$ Shamir sharings for input wires of multiplication gates. This is different from the semi-honest protocol where all parties only need to reconstruct $\{\mu_\alpha\}_\alpha$ to $P_1$. Recall that in the semi-honest protocol (the optimized version in Section C), for each group of multiplication gates with input wires $\boldsymbol{\alpha},\boldsymbol{\beta}$ and output wires $\boldsymbol{\gamma}$, $P_1$ distributes $[\![\boldsymbol{v}_{\boldsymbol{\alpha}}+\boldsymbol{a}]\!]_{k-1}$ and $[\![\boldsymbol{v}_{\boldsymbol{\beta}}+\boldsymbol{b}]\!]_{k-1}$ to all parties. These two sharings are used to compute and reconstruct $\boldsymbol{\mu}_{\boldsymbol{\gamma}}$ to $P_1$. We observe that we can reuse these two sharings to compute Shamir sharings for wires $\boldsymbol{\alpha},\boldsymbol{\beta}$:

- Recall that in the circuit-dependent preprocessing phase, all parties will keep $\{[\![a_i|_i]\!]_t, [\![b_i|_i]\!]_t\}_{i=1}^k$.
- For all $i\in\{1,2,\ldots,k\}$, $[\![\boldsymbol{v}_{\boldsymbol{\alpha}}+\boldsymbol{a}]\!]_{k-1}$ can be viewed as $[\![v_{\alpha_i}+a_i|_i]\!]_{k-1}$. Therefore, all parties can locally compute $[\![v_{\alpha_i}|_i]\!] = [\![v_{\alpha_i}+a_i|_i]\!]_{k-1} - [\![a_i|_i]\!]_t$. Similarly, they can locally compute $[\![v_{\beta_i}|_i]\!]$.

Therefore, the online protocol in the malicious security setting follows its semi-honest version to evaluate the circuit and all parties locally compute degree-$t$ Shamir sharings for input wires of multiplication gates as described above. We give more details as follows.

*Input Phase.* In the input phase, we also need to obtain degree-$t$ Shamir sharings for input values. To this end, we choose to use a simple protocol which requires $O(n)$ elements of communication per gate. Although this is asymptotically worse than that in the semi-honest version, the number of input gates is small compared with the circuit size. We believe this will not affect the concrete efficiency of the protocol.

Recall that in $\mathcal{F}_{\mathsf{PrepMal}}$, for each group of input gates that belong to some `Client`, all parties have prepared

- A set of random degree-$t$ Shamir sharings $\{[\![r_i|_i]\!]_t\}_{i=1}^k$.
- A random degree-$(n-1)$ packed Shamir sharing $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}]\!]_{n-1}$, where $\boldsymbol{\alpha}$ are the output wires of these input gates.

Suppose $\boldsymbol{v}_{\boldsymbol{\alpha}}$ are the input values of `Client`. All parties will send their shares of $\{[\![r_i|_i]\!]_t\}_{i=1}^k$ and $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}]\!]_{n-1}$ to `Client`. Then, `Client` distributes a degree-$t$ packed Shamir sharing $[\![\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{r}]\!]_t$ to all parties, where $\boldsymbol{r} = (r_1, \ldots, r_k)$. Here we choose to use a degree-$t$ packed Shamir sharing so that we do not need to verify whether the shares of honest parties form a valid degree-$t$ packed Shamir sharing. This is because a degree-$t$ packed Shamir sharing requires $t+1$ shares to reconstruct the secrets. Since there are $n - t = t + 1$ honest parties, any shares of honest parties form a valid degree-$t$ packed Shamir sharing. `Client` also sends $\boldsymbol{\mu}_{\boldsymbol{\alpha}} = \boldsymbol{v}_{\boldsymbol{\alpha}} - \boldsymbol{\lambda}_{\boldsymbol{\alpha}}$ to $P_1$ as the semi-honest protocol. Finally, all parties use $[\![\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{r}]\!]_t$ and $\{[\![r_i|_i]\!]_t\}_{i=1}^k$ to compute individual degree-$t$ Shamir sharings for inputs of `Client`.

The description of $\Pi_{\mathsf{InputMal}}$ appears in Protocol 10. The communication complexity of $\Pi_{\mathsf{InputMal}}$ is $(k+2) \cdot n/k + 1 = n + 9$ elements per input gate.

**Protocol 10:** $\Pi_{\mathsf{InputMal}}$

1. For each group of input gates that belong to `Client`, let $\boldsymbol{\alpha}$ denote the batch of output wires of these input gates. All parties receive $\{[\![r_i|_i]\!]_t\}_{i=1}^k$ and $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}]\!]_{n-1}$ from $\mathcal{F}_{\mathsf{PrepMal}}$ and `Client` holds inputs $\boldsymbol{v}_{\boldsymbol{\alpha}}$.
2. All parties send to `Client` their shares of $\{[\![r_i|_i]\!]_t\}_{i=1}^k$ and $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}]\!]_{n-1}$.
3. For all $i \in \{1, 2, \ldots, k\}$, `Client` checks whether the shares of $[\![r_i|_i]\!]_t$ lie on a degree-$t$ polynomial. If not, `Client` aborts.
4. `Client` reconstructs the secrets $\boldsymbol{r} = (r_1, \ldots, r_k)$ and $\boldsymbol{\lambda}_{\boldsymbol{\alpha}}$. Then, `Client` samples a random degree-$t$ packed Shamir sharing $[\![\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{r}]\!]_t$ and computes $\boldsymbol{\mu}_{\boldsymbol{\alpha}} = \boldsymbol{v}_{\boldsymbol{\alpha}} - \boldsymbol{\lambda}_{\boldsymbol{\alpha}}$.
5. `Client` distributes the shares of $[\![\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{r}]\!]_t$ to all parties and sends $\boldsymbol{\mu}_{\boldsymbol{\alpha}}$ to $P_1$.
6. For all $i \in \{1, 2, \ldots, k\}$, all parties locally compute $[\![v_{\alpha_i}|_i]\!]_t = [\![\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{r}]\!]_t - [\![r_i|_i]\!]_t$.

*Computation Phase.* As we explained above, we follow the semi-honest version except that all parties locally compute degree-$t$ Shamir sharings for input wires of multiplication gates. Concretely, we will maintain the invariant that $P_1$ learns $\mu_\alpha = v_\alpha - \lambda_\alpha$ for all wire $\alpha$ in the circuit. Note that it holds for the output wires of input gates.

For each addition gate with input wires $\alpha, \beta$ and output wire $\gamma$, $P_1$ computes $\mu_\gamma = \mu_\alpha + \mu_\beta$ as the semi-honest version.

For each group of multiplication gates with input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$ and output wires $\boldsymbol{\gamma}$, recall that $P_1$ receives $\boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \boldsymbol{a}$ and $\boldsymbol{\lambda}_{\boldsymbol{\beta}} + \boldsymbol{b}$ from $\mathcal{F}_{\mathsf{PrepMal}}$. $P_1$ computes $\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{a} = \boldsymbol{\mu}_{\boldsymbol{\alpha}} + (\boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \boldsymbol{a})$ and distributes the degree-$(k-1)$ packed Shamir sharing $[\![\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{a}]\!]_{k-1}$ to all parties. Similarly, $P_1$ computes $\boldsymbol{v}_{\boldsymbol{\beta}} + \boldsymbol{b}$ and distributes $[\![\boldsymbol{v}_{\boldsymbol{\beta}} + \boldsymbol{b}]\!]_{k-1}$ to all parties. Note that these two steps are identical to the semi-honest version. Recall that all parties receive $\{([\![a_i|_i]\!]_t, [\![b_i|_i]\!]_t, [\![c_i|_i]\!]_t)\}_{i=1}^k$ and $[\![\boldsymbol{\lambda}_{\boldsymbol{\gamma}}]\!]_{n-1}$ from $\mathcal{F}_{\mathsf{PrepMal}}$. All parties locally compute

$$\begin{aligned}
[\![\boldsymbol{a}]\!]_{n-k} &= [\![\boldsymbol{e}_1]\!]_{k-1} * [\![a_1|_1]\!]_t + \ldots + [\![\boldsymbol{e}_k]\!]_{k-1} * [\![a_k|_k]\!]_t \\
[\![\boldsymbol{b}]\!]_{n-k} &= [\![\boldsymbol{e}_1]\!]_{k-1} * [\![b_1|_1]\!]_t + \ldots + [\![\boldsymbol{e}_k]\!]_{k-1} * [\![b_k|_k]\!]_t \\
[\![\boldsymbol{c}]\!]_{n-k} &= [\![\boldsymbol{e}_1]\!]_{k-1} * [\![c_1|_1]\!]_t + \ldots + [\![\boldsymbol{e}_k]\!]_{k-1} * [\![c_k|_k]\!]_t
\end{aligned}$$

Here $\boldsymbol{e}_i$ is the $i$-th unit vector in $\mathbb{F}^k$, i.e., all entries are $0$ except the $i$-th entry is $1$. After receiving from $P_1$ $[\![\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{a}]\!]_{k-1}, [\![\boldsymbol{v}_{\boldsymbol{\beta}} + \boldsymbol{b}]\!]_{k-1}$ , all parties locally compute a degree-$(n-1)$ packed Shamir sharing of $\boldsymbol{\mu}_{\boldsymbol{\gamma}}$ as follows:

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

The correctness follows the same argument as the semi-honest version in Section C. Finally all parties use $[\![\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{a}]\!]_{k-1}, [\![\boldsymbol{v}_{\boldsymbol{\beta}} + \boldsymbol{b}]\!]_{k-1}$ and $\{[\![a_i|_i]\!]_t, [\![b_i|_i]\!]_t\}_{i=1}^k$ to compute individual degree-$t$ Shamir sharings for input wires of multiplication gates.

The description of $\Pi_{\mathsf{MultMal}}$ appears in Protocol 11. The communication complexity of $\Pi_{\mathsf{MultMal}}$ is $3n/k = 12$ elements per gate among all parties.

**Protocol 11:** $\Pi_{\mathsf{MultMal}}$

1. For each group of multiplication gates with input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$ and output wires $\boldsymbol{\gamma}$, all parties receive from $\mathcal{F}_{\mathsf{PrepMal}}$
   - A set of Beaver triples $\{([\![a_i|_i]\!]_t, [\![b_i|_i]\!]_t, [\![c_i|_i]\!]_t)\}_{i=1}^k$,

- A random degree-$(n-1)$ packed Shamir sharing $[\![\boldsymbol{\lambda}_\gamma]\!]_{n-1}$.

$P_1$ receives from $\mathcal{F}_{\mathsf{PrepMal}}$ two vectors $\boldsymbol{d}_1 = \boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \boldsymbol{a}, \boldsymbol{d}_2 = \boldsymbol{\lambda}_\beta + \boldsymbol{b}$. $P_1$ also learns $\boldsymbol{\mu}_{\boldsymbol{\alpha}}, \boldsymbol{\mu}_\beta$ during the online phase.

2. $P_1$ locally computes $\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{a} = \boldsymbol{\mu}_{\boldsymbol{\alpha}} + \boldsymbol{d}_1$ and $\boldsymbol{v}_\beta + \boldsymbol{b} = \boldsymbol{\mu}_\beta + \boldsymbol{d}_2$. Then, $P_1$ computes $[\![\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{a}]\!]_{k-1}$ and $[\![\boldsymbol{v}_\beta + \mathbf{b}]\!]_{k-1}$ and distributes the shares to all parties.
3. All parties locally compute

$$\begin{aligned}
[\![\boldsymbol{a}]\!]_{n-k} &= [\![\boldsymbol{e}_1]\!]_{k-1} * [\![a_1|_1]\!]_t + \ldots + [\![\boldsymbol{e}_k]\!]_{k-1} * [\![a_k|_k]\!]_t \\
[\![\boldsymbol{b}]\!]_{n-k} &= [\![\boldsymbol{e}_1]\!]_{k-1} * [\![b_1|_1]\!]_t + \ldots + [\![\boldsymbol{e}_k]\!]_{k-1} * [\![b_k|_k]\!]_t \\
[\![\boldsymbol{c}]\!]_{n-k} &= [\![\boldsymbol{e}_1]\!]_{k-1} * [\![c_1|_1]\!]_t + \ldots + [\![\boldsymbol{e}_k]\!]_{k-1} * [\![c_k|_k]\!]_t \\
[\![\boldsymbol{\mu}_\gamma]\!]_{n-1} &= [\![\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{a}]\!]_{k-1} * [\![\boldsymbol{v}_\beta + \mathbf{b}]\!]_{k-1} - [\![\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{a}]\!]_{k-1} * [\![\boldsymbol{b}]\!]_{n-k} \\
&\quad - [\![\boldsymbol{v}_\beta + \mathbf{b}]\!]_{k-1} * [\![\boldsymbol{a}]\!]_{n-k} + [\![\mathbf{c}]\!]_{n-k} - [\![\boldsymbol{\lambda}_\gamma]\!]_{n-1}.
\end{aligned}$$

4. $P_1$ collects the whole sharing $[\![\boldsymbol{\mu}_\gamma]\!]_{n-1}$ from all parties and reconstructs $\boldsymbol{\mu}_\gamma$.
5. For all $i \in \{1, 2, \ldots, k\}$, all parties locally compute $[\![v_{\alpha_i}|_i]\!]_t = [\![\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{a}]\!]_{k-1} - [\![a_i|_i]\!]_t$ and $[\![v_{\beta_i}|_i]\!]_t = [\![\boldsymbol{v}_\beta + \mathbf{b}]\!]_{k-1} - [\![b_i|_i]\!]_t$.

*Output Phase and Validity Check.* After evaluating the whole circuit, we will compute a degree-$t$ Shamir sharing for each output gate. For each group of output gates with input wires $\boldsymbol{\alpha}$, recall that

- All parties receive a set of degree-$t$ Shamir sharings $\{[\![r_i|_i]\!]_t\}_{i=1}^k$ from $\mathcal{F}_{\mathsf{PrepMal}}$.
- $P_1$ receives $\boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \boldsymbol{r}$ from $\mathcal{F}_{\mathsf{PrepMal}}$ where $\boldsymbol{r} = (r_1, r_2, \ldots, r_k)$.
- $P_1$ learns $\boldsymbol{\mu}_{\boldsymbol{\alpha}} = \boldsymbol{v}_{\boldsymbol{\alpha}} - \boldsymbol{\lambda}_{\boldsymbol{\alpha}}$.

Similarly to the input wires of multiplication gates, $P_1$ locally computes $\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{r} = \boldsymbol{\mu}_{\boldsymbol{\alpha}} + (\boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \boldsymbol{r})$ and distributes the degree-$(k-1)$ packed Shamir sharing $[\![\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{r}]\!]$ to all parties. In this way, all parties can locally compute degree-$t$ Shamir sharings for input wires of output gates.

Before reconstructing the outputs to clients, we need to verify the correctness of the computation. The verification contains two parts

- First, we need to verify that the degree-$(k-1)$ packed Shamir sharings distributed by $P_1$ are valid. That is, for each degree-$(k-1)$ packed Shamir sharing distributed by $P_1$, the shares of honest parties lie on a degree-$(k-1)$ polynomial.
- Second, we need to check that for each input wire of a multiplication gate or an output gate, the secret of the corresponding degree-$t$ Shamir sharing is the correct wire value.

We will only do the first step in this part. As we will prove later, after the first check, the degree-$t$ Shamir sharings all parties hold are valid. In particular, any attack of the adversary can be reduced to an additive attack. That is, what an adversary can do is to add a constant error to the secret of each degree-$t$ Shamir sharing. We will discuss how to verify the correctness of the secrets in the next part.

To verify the degree-$(k-1)$ packed Shamir sharings distributed by $P_1$, we simply compute a random linear combination of all degree-$(k-1)$ packed Shamir sharings and then let each party check the validity of the resulting sharing. To this end, we will need a functionality $\mathcal{F}_{\mathsf{Coin}}$ that samples a random field element to all parties. An instantiation of $\mathcal{F}_{\mathsf{Coin}}$ can be found in [GS20], which has communication complexity $O(n^2)$ elements.

**Functionality 9: $\mathcal{F}_{\mathsf{Coin}}$**

1. $\mathcal{F}_{\mathsf{Coin}}$ samples a random field element $r$.
2. $\mathcal{F}_{\mathsf{Coin}}$ sends $r$ to the adversary.
   - If the adversary replies `continue`, $\mathcal{F}_{\mathsf{Coin}}$ sends $r$ to honest parties.
   - If the adversary replies `abort`, $\mathcal{F}_{\mathsf{Coin}}$ sends `abort` to honest parties.

Let $\mathbb{K}$ be an extension field of $\mathbb{F}$ such that $|\mathbb{K}| \geq 2^\kappa$, where $\kappa$ is the security parameter. All parties will use $\mathcal{F}_{\mathsf{Coin}}$ to generate a random field element $r \in \mathbb{K}$. Let $\{[\![\boldsymbol{w}_i]\!]_{k-1}\}_{i=1}^m$ denote all degree-$(k-1)$ packed Shamir sharings distributed by $P_1$. All parties will locally compute

$$[\![\boldsymbol{w}]\!]_{k-1} = \sum_{i=1}^m r^{i-1} \cdot [\![\boldsymbol{w}_i]\!]_{k-1}.$$

Then each party collects the whole sharing $[\![\boldsymbol{w}]\!]_{k-1}$ and checks whether the shares form a valid degree-$(k-1)$ packed Shamir sharing. The description of $\Pi_{\mathsf{Consistency}}$ appears in Protocol 12. The communication complexity of $\Pi_{\mathsf{Consistency}}$ is $O(n^2)$ elements in $\mathbb{K}$, which is independent of the number of sharings.

**Protocol 12: $\Pi_{\mathsf{Consistency}}$**

1. Let $\{[\![\boldsymbol{w}_i]\!]_{k-1}\}_{i=1}^{m}$ denote all degree-$(k-1)$ packed Shamir sharings distributed by $P_1$.
2. All parties invoke $\mathcal{F}_{\mathsf{Coin}}$ to generate a random element $r \in \mathbb{K}$.
3. All parties locally compute
$$[\![\boldsymbol{w}]\!]_{k-1} = \sum_{i=1}^{m} r^{i-1} \cdot [\![\boldsymbol{w}_i]\!]_{k-1}.$$
4. Each party $P_i$ sends its share of $[\![\boldsymbol{w}]\!]_{k-1}$ to all other parties. Then each party $P_j$ checks whether the shares of $[\![\boldsymbol{w}]\!]_{k-1}$ lie on a degree-$(k-1)$ polynomial. If true, $P_j$ accepts the check. Otherwise, $P_j$ aborts.

In the following, when we say a degree-$(k-1)$ packed Shamir sharing $[\![\boldsymbol{s}]\!]_{k-1}$ is valid, we means that the shares of $[\![\boldsymbol{s}]\!]_{k-1}$ of honest parties lie on a degree-$(k-1)$ polynomial. We have the following lemma.

**Lemma 6.** *If there exists $i \in \{1,2,\ldots,m\}$ such that $[\![\boldsymbol{w}_i]\!]_{k-1}$ is not a valid degree-$(k-1)$ packed Shamir sharing, then all honest parties abort in $\Pi_{\mathsf{Consistency}}$ with overwhelming probability.*

*Proof.* Consider the following polynomial of sharings in $\mathbb{K}$:
$$\boldsymbol{f}(r) = \sum_{i=1}^{m} r^{i-1} \cdot [\![\boldsymbol{w}_i]\!]_{k-1}.$$
Suppose at least one degree-$(k-1)$ packed Shamir sharing in $\{[\![\boldsymbol{w}_i]\!]_{k-1}\}_{i=1}^{m}$ is invalid. We show that the number of $r$ such that $\boldsymbol{f}(r)$ is a valid degree-$(k-1)$ packed Shamir sharing is bounded by $m-1$.

If not, then there exists $r_1, r_2, \ldots, r_m$ such that $\boldsymbol{f}(r_j)$ is a valid degree-$(k-1)$ packed Shamir sharing. Consider the matrix $\boldsymbol{M} = (r_j^{i-1})_{i,j}$. Then
$$(\boldsymbol{f}(r_1), \ldots, \boldsymbol{f}(r_m))^{\mathrm{T}} = \boldsymbol{M} \cdot ([\![\boldsymbol{w}_1]\!]_{k-1}, \ldots, [\![\boldsymbol{w}_m]\!]_{k-1})^{\mathrm{T}}.$$

Note that $\boldsymbol{M}$ is a Vandermonde matrix of size $m \times m$, which is invertible. Therefore, each $[\![\boldsymbol{w}_i]\!]_{k-1}$ is a linear combination of $\boldsymbol{f}(r_1), \ldots, \boldsymbol{f}(r_m)$, which implies that $[\![\boldsymbol{w}_i]\!]_{k-1}$ is also a valid degree-$(k-1)$ packed Shamir sharing. However, it contradicts with the assumption that at least one degree-$(k-1)$ packed Shamir sharing in $\{[\![\boldsymbol{w}_i]\!]_{k-1}\}_{i=1}^{m}$ is invalid.

Therefore, the number of $r$ such that $\boldsymbol{f}(r)$ is a valid degree-$(k-1)$ packed Shamir sharing is bounded by $m-1$. Since $r$ is generated randomly by $\mathcal{F}_{\mathsf{Coin}}$, the probability that $[\![\boldsymbol{w}]\!]_{k-1}$ is valid is bounded by $\frac{m}{2^\kappa}$, which is negligible. Note that when $[\![\boldsymbol{w}]\!]_{k-1}$ is invalid, all honest parties will abort.

*Summary.* We describe the functionality $\mathcal{F}_{\mathsf{Evaluate}}$ in Functionality 10 for the evaluation of the circuit in the online phase. The realization of $\mathcal{F}_{\mathsf{Evaluate}}$, $\Pi_{\mathsf{Evaluate}}$, appears in Protocol 13. The communication complexity of $\Pi_{\mathsf{Evaluate}}$ is 12 elements per multiplication gate among all parties.

**Functionality 10: $\mathcal{F}_{\mathsf{Evaluate}}$**

1. $\mathcal{F}_{\mathsf{Evaluate}}$ receives the input from all clients. Let $C$ denote the circuit.
2. $\mathcal{F}_{\mathsf{Evaluate}}$ receives the set of corrupted parties, denoted by $\mathcal{C}orr$. For each group of input gates with output wires $\boldsymbol{\alpha}$, let $\boldsymbol{v}_{\boldsymbol{\alpha}}$ denote the input values associated with $\boldsymbol{\alpha}$. For all $i \in \{1,\ldots,k\}$, $\mathcal{F}_{\mathsf{Evaluate}}$ receives from the adversary a set of shares $\{u_{i,j}\}_{j \in \mathcal{C}orr}$. Then $\mathcal{F}_{\mathsf{Evaluate}}$ computes a degree-$t$ Shamir sharing $[v_{\alpha_i}|i]_t$ such that for all $P_j \in \mathcal{C}orr$, the $j$-th share of $[v_{\alpha_i}|i]_t$ is $u_{i,j}$. Finally, $\mathcal{F}_{\mathsf{Evaluate}}$ distributes the shares of $[v_{\alpha_i}|i]_t$ to honest parties.
3. $\mathcal{F}_{\mathsf{Evaluate}}$ evaluates the circuit $C$ layer by layer. For each addition gate with input wires $\alpha, \beta$ and output wire $\gamma$, $\mathcal{F}_{\mathsf{Evaluate}}$ computes $v_\gamma = v_\alpha + v_\beta$. For each group of multiplication gates with input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$,
   (a) $\mathcal{F}_{\mathsf{Evaluate}}$ receives two vectors of additive errors $\Delta(\boldsymbol{v}_{\boldsymbol{\alpha}}), \Delta(\boldsymbol{v}_{\boldsymbol{\beta}})$ from the adversary. Then, $\mathcal{F}_{\mathsf{Evaluate}}$ sets $\boldsymbol{v}_{\boldsymbol{\alpha}} = \boldsymbol{v}_{\boldsymbol{\alpha}} + \Delta(\boldsymbol{v}_{\boldsymbol{\alpha}})$ and $\boldsymbol{v}_{\boldsymbol{\beta}} = \boldsymbol{v}_{\boldsymbol{\beta}} + \Delta(\boldsymbol{v}_{\boldsymbol{\beta}})$.

(b) For all $i \in \{1, 2, \ldots, k\}$, $\mathcal{F}_{\mathsf{Evaluate}}$ receives from the adversary a set of shares $\{(u_{i,j}^{(1)}, u_{i,j}^{(2)})\}_{j \in \mathcal{C}orr}$. Then $\mathcal{F}_{\mathsf{Evaluate}}$ computes degree-$t$ Shamir sharings $[\![v_{\alpha_i}|i]\!]_t$ and $[\![v_{\beta_i}|i]\!]_t$ such that for all $P_j \in \mathcal{C}orr$, the $j$-th share of $[\![v_{\alpha_i}|i]\!]_t$ is $u_{i,j}^{(1)}$ and the $j$-th share of $[\![v_{\beta_i}|i]\!]_t$ is $u_{i,j}^{(2)}$. Finally, $\mathcal{F}_{\mathsf{Evaluate}}$ distributes the shares of $[\![v_{\alpha_i}|i]\!]_t, [\![v_{\beta_i}|i]\!]_t$ to honest parties.

(c) $\mathcal{F}_{\mathsf{Evaluate}}$ computes $\boldsymbol{v_\gamma} = \boldsymbol{v_\alpha} * \boldsymbol{v_\beta}$.

4. For each group of output gates with input wires $\boldsymbol{\alpha}$,

(a) $\mathcal{F}_{\mathsf{Evaluate}}$ receives a vector of additive errors $\Delta(\boldsymbol{v_\alpha})$ from the adversary. Then, $\mathcal{F}_{\mathsf{Evaluate}}$ sets $\boldsymbol{v_\alpha} = \boldsymbol{v_\alpha} + \Delta(\boldsymbol{v_\alpha})$.

(b) For all $i \in \{1, 2, \ldots, k\}$, $\mathcal{F}_{\mathsf{Evaluate}}$ receives from the adversary a set of shares $\{u_{i,j}\}_{j \in \mathcal{C}orr}$. Then $\mathcal{F}_{\mathsf{Evaluate}}$ computes a degree-$t$ Shamir sharing $[\![v_{\alpha_i}|i]\!]_t$ such that for all $P_j \in \mathcal{C}orr$, the $j$-th share of $[\![v_{\alpha_i}|i]\!]_t$ is $u_{i,j}$. Finally, $\mathcal{F}_{\mathsf{Evaluate}}$ distributes the shares of $[\![v_{\alpha_i}|i]\!]_t$ to honest parties.

5. On receiving `abort`, $\mathcal{F}_{\mathsf{Evaluate}}$ sends `abort` to all parties.

**Protocol 13: $\Pi_{\mathsf{Evaluate}}$**

1. **Preprocessing Phase**: All parties invoke $\mathcal{F}_{\mathsf{PrepMal}}$ to receive correlated randomness that will be used in the online phase.
2. **Input Phase**: In the input layer, for each group of $k$ input gates that belong to some `Client`, let $\boldsymbol{\alpha}$ denote the output wires of these input gates. All parties and `Client` invoke $\Pi_{\mathsf{InputMal}}$. At the end of the protocol, all parties hold $\{[\![v_{\alpha_i}|i]\!]_t\}_{i=1}^k$. And $P_1$ learns $\boldsymbol{\mu_\alpha} = \boldsymbol{v_\alpha} - \boldsymbol{\lambda_\alpha}$, where $\boldsymbol{v_\alpha}$ are the input values of `Client`, and $\boldsymbol{\lambda_\alpha}$ are the random values associated with the batch of wires $\boldsymbol{\alpha}$ generated by $\mathcal{F}_{\mathsf{PrepMal}}$.
3. **Computation Phase**: All parties maintain the invariant that for each wire $\alpha$, $P_1$ learns $\mu_\alpha = v_\alpha - \lambda_\alpha$, where $v_\alpha$ is the real value associated with the wire $\alpha$, and $\lambda_\alpha$ is a random value associated with $\alpha$ generated by $\mathcal{F}_{\mathsf{PrepMal}}$. The circuit is evaluated layer by layer. Assume that the invariant holds for wires in previous layers. Consider gates in the current layer.
   For each addition gate with input wires $\alpha, \beta$ and output wire $\gamma$, $P_1$ locally compute $\mu_\gamma = \mu_\alpha + \mu_\beta$.
   For each group of $k$ multiplication gates with input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$ and output wires $\boldsymbol{\gamma}$, all parties invoke $\Pi_{\mathsf{MultMal}}$. At the end of the protocol, all parties hold $\{[\![v_{\alpha_i}|i]\!]_t, [\![v_{\beta_i}|i]\!]_t\}_{i=1}^k$. And $P_1$ learns $\boldsymbol{\mu_\gamma}$.
4. **Output Phase and Validity Check**: For each group of $k$ output gates with input wires $\boldsymbol{\alpha}$, recall that
   – All parties receive $\{[\![r_i|i]\!]_t\}_{i=1}^k$ from $\mathcal{F}_{\mathsf{PrepMal}}$,
   – $P_1$ receives $\boldsymbol{\lambda_\alpha} + \boldsymbol{r}$ from $\mathcal{F}_{\mathsf{PrepMal}}$, where $\boldsymbol{r} = (r_1, \ldots, r_k)$,
   – $P_1$ learns $\boldsymbol{\mu_\alpha} = \boldsymbol{v_\alpha} - \boldsymbol{\lambda_\alpha}$ by the invariant.
   $P_1$ computes $\boldsymbol{v_\alpha} + \boldsymbol{r} = \boldsymbol{\mu_\alpha} + (\boldsymbol{\lambda_\alpha} + \boldsymbol{r})$. Then $P_1$ distributes the degree-$(k-1)$ packed Shamir sharing $[\![\boldsymbol{v_\alpha} + \boldsymbol{r}]\!]_{k-1}$ to all parties. For all $i \in \{1, 2, \ldots, k\}$, all parties locally compute $[\![v_{\alpha_i}|i]\!]_t = [\![\boldsymbol{v_\alpha} + \boldsymbol{r}]\!]_{k-1} - [\![r_i|i]\!]_t$.
   Finally, let $\{[\![\boldsymbol{w}_i]\!]_{k-1}\}_{i=1}^m$ denote all degree-$(k-1)$ packed Shamir sharings distributed by $P_1$. All parties invoke $\Pi_{\mathsf{Consistency}}$ to check the validity of these sharings.

**Lemma 7.** *Protocol $\Pi_{\mathsf{Evaluate}}$ securely computes $\mathcal{F}_{\mathsf{Evaluate}}$ in the $\mathcal{F}_{\mathsf{PrepMal}}$-hybrid model against a fully malicious adversary who controls $t$ parties and up to $\mathsf{c}$ clients.*

*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.

1. In Step 1, $\mathcal{S}$ emulates the ideal functionality $\mathcal{F}_{\mathsf{PrepMal}}$ as follows.
   – For each sharing generated by $\mathcal{F}_{\mathsf{PrepMal}}$, $\mathcal{S}$ receives from the adversary the shares of corrupted parties.
   – For each multiplication triple $([\![a_i|i]\!]_t, [\![b_i|i]\!]_t, [\![c_i|i]\!]_t)$, $\mathcal{S}$ receives from the adversary the corresponding additive error $\eta_i$.
   – For each group of multiplication gates with input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$, $\mathcal{S}$ receives from the adversary two vectors of additive errors $\Delta(\boldsymbol{\lambda_\alpha} + \boldsymbol{a}), \Delta(\boldsymbol{\lambda_\beta} + \boldsymbol{b})$. $\mathcal{S}$ samples two random vectors as $\boldsymbol{\lambda_\alpha} + \boldsymbol{a}, \boldsymbol{\lambda_\beta} + \boldsymbol{b}$ and computes $\boldsymbol{d}_1 = (\boldsymbol{\lambda_\alpha} + \boldsymbol{a}) + \Delta(\boldsymbol{\lambda_\alpha} + \boldsymbol{a}), \boldsymbol{d}_2 = (\boldsymbol{\lambda_\beta} + \boldsymbol{b}) + \Delta(\boldsymbol{\lambda_\beta} + \boldsymbol{b})$. Finally $\mathcal{S}$ sends $\boldsymbol{d}_1, \boldsymbol{d}_2$ to $P_1$.
     Similarly, for each group of output gates with input wires $\boldsymbol{\alpha}$, $\mathcal{S}$ receives from the adversary a vector of additive errors $\Delta(\boldsymbol{\lambda_\alpha} + \boldsymbol{r})$. $\mathcal{S}$ samples a random vector as $\boldsymbol{\lambda_\alpha} + \boldsymbol{r}$ and computes $\boldsymbol{d} = (\boldsymbol{\lambda_\alpha} + \boldsymbol{r}) + \Delta(\boldsymbol{\lambda_\alpha} + \boldsymbol{r})$. Finally $\mathcal{S}$ sends $\boldsymbol{d}$ to $P_1$.

- For each group of input gates with output wires $\boldsymbol{\alpha}$, $\mathcal{S}$ receives from the adversary a vector of additive errors $\Delta(\boldsymbol{\lambda}_{\boldsymbol{\alpha}})$. For each group of multiplication gates with output wires $\boldsymbol{\gamma}$, $\mathcal{S}$ receives from the adversary a vector of additive errors $\Delta(\boldsymbol{\lambda}_{\boldsymbol{\gamma}})$.

2. In Step 2, for each group of $k$ input gates that belong to some `Client`, let $\boldsymbol{\alpha}$ denote the output wires of these input gates. $\mathcal{S}$ simulates $\Pi_{\mathsf{InputMal}}$ as follows.
   - Case 1: `Client` is an honest party.
     (a) $\mathcal{S}$ receives from corrupted parties their shares of $\{[\![r_i|_i]\!]_t\}_{i=1}^k$ and $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}]\!]_{n-1}$. For all $i \in \{1, 2, \ldots, k\}$, $\mathcal{S}$ checks whether the shares of $[\![r_i|_i]\!]_t$ received from corrupted parties are the same as those received in $\mathcal{F}_{\mathsf{PrepMal}}$. If not, $\mathcal{S}$ aborts the protocol on behalf of `Client`. Otherwise, $\mathcal{S}$ generates a random vector as $\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{r}$ and samples a random degree-$t$ packed Shamir sharing $[\![\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{r}]\!]_t$.
     (b) $\mathcal{S}$ samples a random vector as $\boldsymbol{\mu}_{\boldsymbol{\alpha}}$. Let $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}]\!]_{n-1}$ denote the sharing that `Client` receives from all parties, and $[\![\overline{\boldsymbol{\lambda}_{\boldsymbol{\alpha}}}]\!]_{n-1}$ denote the sharing where the shares of corrupted parties are replaced by those learnt by $\mathcal{S}$ when emulating $\mathcal{F}_{\mathsf{PrepMal}}$. $\mathcal{S}$ computes the shares of corrupted parties of the following sharing
     $$[\![\rho(\boldsymbol{\lambda}_{\boldsymbol{\alpha}})]\!]_{n-1} = [\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}]\!]_{n-1} - [\![\overline{\boldsymbol{\lambda}_{\boldsymbol{\alpha}}}]\!]_{n-1}.$$
     $\mathcal{S}$ sets the shares of honest parties to be $0$ and reconstructs the secrets $\rho(\boldsymbol{\lambda}_{\boldsymbol{\alpha}})$.
     (c) Recall that $\mathcal{S}$ receives a vector of additive errors $\Delta(\boldsymbol{\lambda}_{\boldsymbol{\alpha}})$ from the adversary when emulating $\mathcal{F}_{\mathsf{PrepMal}}$. $\mathcal{S}$ sets $\Delta(\boldsymbol{\mu}_{\boldsymbol{\alpha}}) = -\rho(\boldsymbol{\lambda}_{\boldsymbol{\alpha}}) - \Delta(\boldsymbol{\lambda}_{\boldsymbol{\alpha}})$ and sends $\boldsymbol{\mu}_{\boldsymbol{\alpha}} + \Delta(\boldsymbol{\mu}_{\boldsymbol{\alpha}})$ to $P_1$.
     (d) For all $i \in \{1, \ldots, k\}$, $\mathcal{S}$ computes the shares of $[\![v_{\alpha_i}|_i]\!]_t$ of corrupted parties by following Step 6 of $\Pi_{\mathsf{InputMal}}$ and sends the shares of corrupted parties to $\Pi_{\mathsf{Evaluate}}$.
   - Case 2: `Client` is a corrupted party.
     (a) $\mathcal{S}$ generates a random vector as $\boldsymbol{r} = (r_1, \ldots, r_k)$. Then, based on the secret $r_i$ and the shares of corrupted parties, $\mathcal{S}$ computes the shares of $[\![r_i|_i]\!]_t$ held by honest parties. $\mathcal{S}$ sends the shares of honest parties to `Client`.
     (b) $\mathcal{S}$ samples a random vector as $\boldsymbol{\lambda}_{\boldsymbol{\alpha}}$. Recall that $\mathcal{S}$ receives $\Delta(\boldsymbol{\lambda}_{\boldsymbol{\alpha}})$ from the adversary when emulating $\mathcal{F}_{\mathsf{PrepMal}}$. Based on the secrets $\boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \Delta(\boldsymbol{\lambda}_{\boldsymbol{\alpha}})$ and the shares of corrupted parties, $\mathcal{S}$ samples a random degree-$(n-1)$ packed Shamir sharing $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \Delta(\boldsymbol{\lambda}_{\boldsymbol{\alpha}})]\!]_{n-1}$. $\mathcal{S}$ sends the shares of honest parties to `Client`.
     (c) $\mathcal{S}$ receives the shares of $[\![\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{r}]\!]_t$ of honest parties. Then $\mathcal{S}$ reconstructs the whole sharing and learns the secrets $\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{r}$. $\mathcal{S}$ computes the inputs of `Client` by $\boldsymbol{v}_{\boldsymbol{\alpha}} = (\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{r}) - \boldsymbol{r}$ and sends $\boldsymbol{v}_{\boldsymbol{\alpha}}$ to $\Pi_{\mathsf{Evaluate}}$.
     (d) $\mathcal{S}$ computes $\boldsymbol{\mu}_{\boldsymbol{\alpha}} = \boldsymbol{v}_{\boldsymbol{\alpha}} - \boldsymbol{\lambda}_{\boldsymbol{\alpha}}$.
       - If $P_1$ is honest, $\mathcal{S}$ receives $\widetilde{\boldsymbol{\mu}_{\boldsymbol{\alpha}}}$ from `Client`. $\mathcal{S}$ sets $\Delta(\boldsymbol{\mu}_{\boldsymbol{\alpha}}) = \widetilde{\boldsymbol{\mu}_{\boldsymbol{\alpha}}} - \boldsymbol{\mu}_{\boldsymbol{\alpha}}$.
       - If $P_1$ is corrupted, $\mathcal{S}$ sets $\Delta(\boldsymbol{\mu}_{\boldsymbol{\alpha}}) = -\Delta(\boldsymbol{\lambda}_{\boldsymbol{\alpha}})$.
     (e) For all $i \in \{1, \ldots, k\}$, $\mathcal{S}$ computes the shares of $[\![v_{\alpha_i}|_i]\!]_t$ of corrupted parties by following Step 6 of $\Pi_{\mathsf{InputMal}}$ and sends the shares of corrupted parties to $\Pi_{\mathsf{Evaluate}}$.

3. In Step 3, $\mathcal{S}$ will compute $\mu_\alpha$ and $\Delta(\mu_\alpha)$ for each wire $\alpha$ in the circuit. Recall that $\mathcal{S}$ has computed $\mu_\alpha$ and $\Delta(\mu_\alpha)$ for each output wire of an input gate.
   For each addition gate with input wires $\alpha, \beta$ and output wire $\gamma$, $\mathcal{S}$ computes $\mu_\gamma = \mu_\alpha + \mu_\beta$ and $\Delta(\mu_\gamma) = \Delta(\mu_\alpha) + \Delta(\mu_\beta)$.
   For each group of multiplication gates with input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$ and output wires $\boldsymbol{\gamma}$, $\mathcal{S}$ simulates $\Pi_{\mathsf{MultMal}}$ as follows.
   - Case 1: $P_1$ is an honest party.
     (a) Recall that $\mathcal{S}$ has explicitly generated $\boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \boldsymbol{a}$ and $\boldsymbol{\lambda}_{\boldsymbol{\beta}} + \boldsymbol{b}$ when emulating $\mathcal{F}_{\mathsf{PrepMal}}$. Also recall that $\mathcal{S}$ received two vectors of additive errors $\Delta(\boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \boldsymbol{a})$ and $\Delta(\boldsymbol{\lambda}_{\boldsymbol{\beta}} + \boldsymbol{b})$ from the adversary when emulating $\mathcal{F}_{\mathsf{PrepMal}}$.
     (b) For $\boldsymbol{\alpha}, \boldsymbol{\beta}$, $\mathcal{S}$ learns $\boldsymbol{\mu}_{\boldsymbol{\alpha}}, \boldsymbol{\mu}_{\boldsymbol{\beta}}$ and $\Delta(\boldsymbol{\mu}_{\boldsymbol{\alpha}}), \Delta(\boldsymbol{\mu}_{\boldsymbol{\beta}})$. $\mathcal{S}$ computes $\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{a} = \boldsymbol{\mu}_{\boldsymbol{\alpha}} + (\boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \boldsymbol{a})$ and $\boldsymbol{v}_{\boldsymbol{\beta}} + \boldsymbol{b} = \boldsymbol{\mu}_{\boldsymbol{\beta}} + (\boldsymbol{\lambda}_{\boldsymbol{\beta}} + \boldsymbol{b})$. Then $\mathcal{S}$ sets $\Delta(\boldsymbol{v}_{\boldsymbol{\alpha}}) = \Delta(\boldsymbol{\mu}_{\boldsymbol{\alpha}}) + \Delta(\boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \boldsymbol{a})$ and $\Delta(\boldsymbol{v}_{\boldsymbol{\beta}}) = \Delta(\boldsymbol{\mu}_{\boldsymbol{\beta}}) + \Delta(\boldsymbol{\lambda}_{\boldsymbol{\beta}} + \boldsymbol{b})$.
     On behalf of $P_1$, $\mathcal{S}$ computes and distributes $[\![\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{a} + \Delta(\boldsymbol{v}_{\boldsymbol{\alpha}})]\!]_{k-1}$ and $[\![\boldsymbol{v}_{\boldsymbol{\beta}} + \boldsymbol{b} + \Delta(\boldsymbol{v}_{\boldsymbol{\beta}})]\!]_{k-1}$ to all parties.
     (c) $\mathcal{S}$ follows Step 3 of $\Pi_{\mathsf{MultMal}}$ and computes the shares of $[\![\boldsymbol{\mu}_{\boldsymbol{\gamma}}]\!]_{n-1}$ of corrupted parties. Recall that $\mathcal{S}$ receives $\eta_i$ for each multiplication triple $([\![a_i|_i]\!]_t, [\![b_i|_i]\!]_t, [\![c_i|_i]\!]_t)$ from the adversary when emulating $\mathcal{F}_{\mathsf{PrepMal}}$. $\mathcal{S}$ sets $\boldsymbol{\eta} = (\eta_1, \ldots, \eta_k)$. Also recall that $\mathcal{S}$ receives a vector of additive errors $\Delta(\boldsymbol{\lambda}_{\boldsymbol{\gamma}})$ when emulating $\mathcal{F}_{\mathsf{PrepMal}}$.
     $\mathcal{S}$ samples a random vector as $\boldsymbol{\mu}_{\boldsymbol{\gamma}}$.

(d) $\mathcal{S}$ receives from corrupted parties their shares of $[\![\boldsymbol{\mu_\gamma}]\!]_{n-1}$. Let $[\![\overline{\boldsymbol{\mu_\gamma}}]\!]_{n-1}$ denote the sharing where the shares of corrupted parties are replaced by those computed by $\mathcal{S}$. $\mathcal{S}$ computes the shares of corrupted parties of the following sharing

$$[\![\rho(\boldsymbol{\mu_\gamma})]\!]_{n-1} = [\![\boldsymbol{\mu_\gamma}]\!]_{n-1} - [\![\overline{\boldsymbol{\mu_\gamma}}]\!]_{n-1}.$$

$\mathcal{S}$ sets the shares of honest parties to be $0$ and reconstructs the secrets $\rho(\boldsymbol{\mu_\gamma})$. Then $\mathcal{S}$ sets $\Delta(\boldsymbol{\mu_\gamma}) = \rho(\boldsymbol{\mu_\gamma}) + \boldsymbol{\eta} - \Delta(\boldsymbol{\lambda_\gamma})$.

(e) For all $i \in \{1, \dots, k\}$, $\mathcal{S}$ computes the shares of $[\![v_{\alpha_i}|i]\!]_t, [\![v_{\beta_i}|i]\!]_t$ of corrupted parties by following Step 5 of $\Pi_{\mathsf{MultMal}}$. Then $\mathcal{S}$ sends $\Delta(\boldsymbol{v_\alpha}), \Delta(\boldsymbol{v_\beta})$ and the shares of $\{[\![v_{\alpha_i}|i]\!]_t, [\![v_{\beta_i}|i]\!]_t\}_{i=1}^k$ of corrupted parties to $\mathcal{F}_{\mathsf{Evaluate}}$.

– Case 2: $P_1$ is a corrupted party.

(a) $\mathcal{S}$ receives from $P_1$ the shares of $[\![\boldsymbol{v_\alpha} + \boldsymbol{a}]\!]_{k-1}$ and $[\![\boldsymbol{v_\beta} + \boldsymbol{b}]\!]_{k-1}$ of honest parties.
- For each of $[\![\boldsymbol{v_\alpha} + \boldsymbol{a}]\!]_{k-1}$ and $[\![\boldsymbol{v_\beta} + \boldsymbol{b}]\!]_{k-1}$, if the shares of honest parties do not lie on a degree-$(k-1)$ packed Shamir sharing, $\mathcal{S}$ marks the computation as `fail`. From now, each time $\mathcal{S}$ needs to send a value to $\mathcal{F}_{\mathsf{Evaluate}}$, $\mathcal{S}$ sends $0$ to $\mathcal{F}_{\mathsf{Evaluate}}$. In this case, $\mathcal{S}$ will abort on behalf of honest parties at the end of the protocol.
- Otherwise, $\mathcal{S}$ reconstructs the whole sharings $[\![\boldsymbol{v_\alpha} + \boldsymbol{a}]\!]_{k-1}, [\![\boldsymbol{v_\beta} + \boldsymbol{b}]\!]_{k-1}$ and computes the secrets, which are denoted by $\widetilde{\boldsymbol{v_\alpha} + \boldsymbol{a}}, \widetilde{\boldsymbol{v_\beta} + \boldsymbol{b}}$.
Recall that $\mathcal{S}$ has explicitly generated $\boldsymbol{\lambda_\alpha} + \boldsymbol{a}$ and $\boldsymbol{\lambda_\beta} + \boldsymbol{b}$ when emulating $\mathcal{F}_{\mathsf{PrepMal}}$. When the computation is NOT marked as `fail`, for $\boldsymbol{\alpha}, \boldsymbol{\beta}$, $\mathcal{S}$ learns $\boldsymbol{\mu_\alpha}, \boldsymbol{\mu_\beta}$. $\mathcal{S}$ computes $\boldsymbol{v_\alpha} + \boldsymbol{a} = \boldsymbol{\mu_\alpha} + (\boldsymbol{\lambda_\alpha} + \boldsymbol{a})$ and $\boldsymbol{v_\beta} + \boldsymbol{b} = \boldsymbol{\mu_\beta} + (\boldsymbol{\lambda_\beta} + \boldsymbol{b})$. Then $\mathcal{S}$ sets $\Delta(\boldsymbol{v_\alpha}) = \widetilde{\boldsymbol{v_\alpha} + \boldsymbol{a}} - (\boldsymbol{v_\alpha} + \boldsymbol{a})$ and $\Delta(\boldsymbol{v_\beta}) = \widetilde{\boldsymbol{v_\beta} + \boldsymbol{b}} - (\boldsymbol{v_\beta} + \boldsymbol{b})$.

(b) For each honest party, $\mathcal{S}$ samples a random field element as its share of $[\![\boldsymbol{\mu_\gamma}]\!]_{n-1}$. Then $\mathcal{S}$ sends the shares of $[\![\boldsymbol{\mu_\gamma}]\!]_{n-1}$ of honest parties to $P_1$.
If the computation is NOT marked as `fail`, $\mathcal{S}$ follows Step 3 of $\Pi_{\mathsf{MultMal}}$ and computes the shares of $[\![\boldsymbol{\mu_\gamma}]\!]_{n-1}$ of corrupted parties. Then $\mathcal{S}$ reconstructs the secrets $\overline{\boldsymbol{\mu_\gamma}}$. Recall that $\mathcal{S}$ received a vector of additive errors $\Delta(\boldsymbol{\lambda_\gamma})$ from the adversary when emulating $\mathcal{F}_{\mathsf{PrepMal}}$. Also recall that $\mathcal{S}$ receives $\eta_i$ for each multiplication triple $([\![a_i|i]\!]_t, [\![b_i|i]\!]_t, [\![c_i|i]\!]_t)$ from the adversary when emulating $\mathcal{F}_{\mathsf{PrepMal}}$. $\mathcal{S}$ sets $\boldsymbol{\eta} = (\eta_1, \dots, \eta_k)$. $\mathcal{S}$ sets $\Delta(\boldsymbol{\mu_\gamma}) = \boldsymbol{\eta} - \Delta(\boldsymbol{\lambda_\gamma})$ and computes $\boldsymbol{\mu_\gamma} = \overline{\boldsymbol{\mu_\gamma}} - \Delta(\boldsymbol{\mu_\gamma})$.

(c) If the computation is NOT marked as `fail`, $\mathcal{S}$ follows Step 5 of $\Pi_{\mathsf{MultMal}}$ and computes the shares of $\{[\![v_{\alpha_i}|i]\!]_t, [\![v_{\beta_i}|i]\!]_t\}_{i=1}^k$ of corrupted parties. Then $\mathcal{S}$ sends $\Delta(\boldsymbol{v_\alpha}), \Delta(\boldsymbol{v_\beta})$ and the shares of $\{[\![v_{\alpha_i}|i]\!]_t, [\![v_{\beta_i}|i]\!]_t\}_{i=1}^k$ of corrupted parties to $\mathcal{F}_{\mathsf{Evaluate}}$.

4. In Step 4, for each group of output gates with input wires $\boldsymbol{\alpha}$, $\mathcal{S}$ simulates the behaviors of honest parties as follows.

– Case 1: $P_1$ is an honest party.

(a) Recall that $\mathcal{S}$ has explicitly generated $\boldsymbol{\lambda_\alpha} + \boldsymbol{r}$ when emulating $\mathcal{F}_{\mathsf{PrepMal}}$. Also recall that $\mathcal{S}$ received a vector of additive errors $\Delta(\boldsymbol{\lambda_\alpha} + \boldsymbol{r})$ from the adversary when emulating $\mathcal{F}_{\mathsf{PrepMal}}$.

(b) For $\boldsymbol{\alpha}$, $\mathcal{S}$ learns $\boldsymbol{\mu_\alpha}$ and $\Delta(\boldsymbol{\mu_\alpha})$. $\mathcal{S}$ computes $\boldsymbol{v_\alpha} + \boldsymbol{r} = \boldsymbol{\mu_\alpha} + (\boldsymbol{\lambda_\alpha} + \boldsymbol{r})$. Then $\mathcal{S}$ sets $\Delta(\boldsymbol{v_\alpha}) = \Delta(\boldsymbol{\mu_\alpha}) + \Delta(\boldsymbol{\lambda_\alpha} + \boldsymbol{r})$.
On behalf of $P_1$, $\mathcal{S}$ computes and distributes $[\![\boldsymbol{v_\alpha} + \boldsymbol{r} + \Delta(\boldsymbol{v_\alpha})]\!]_{k-1}$ to all parties.

(c) For all $i \in \{1, \dots, k\}$, $\mathcal{S}$ computes the shares of $[\![v_{\alpha_i}|i]\!]_t$ of corrupted parties by following the protocol. Then $\mathcal{S}$ sends $\Delta(\boldsymbol{v_\alpha})$ and the shares of $\{[\![v_{\alpha_i}|i]\!]_t\}_{i=1}^k$ of corrupted parties to $\mathcal{F}_{\mathsf{Evaluate}}$.

– Case 2: $P_1$ is a corrupted party.

(a) $\mathcal{S}$ receives from $P_1$ the shares of $[\![\boldsymbol{v_\alpha} + \boldsymbol{r}]\!]_{k-1}$ of honest parties.
- For $[\![\boldsymbol{v_\alpha} + \boldsymbol{r}]\!]_{k-1}$, if the shares of honest parties do not lie on a degree-$(k-1)$ packed Shamir sharing, $\mathcal{S}$ marks the computation as `fail`. From now, each time $\mathcal{S}$ needs to send a value to $\mathcal{F}_{\mathsf{Evaluate}}$, $\mathcal{S}$ sends $0$ to $\mathcal{F}_{\mathsf{Evaluate}}$. In this case, $\mathcal{S}$ will abort on behalf of honest parties at the end of the protocol.
- Otherwise, $\mathcal{S}$ reconstructs the whole sharing $[\![\boldsymbol{v_\alpha} + \boldsymbol{r}]\!]_{k-1}$ and computes the secrets, which are denoted by $\widetilde{\boldsymbol{v_\alpha} + \boldsymbol{r}}$.
Recall that $\mathcal{S}$ has explicitly generated $\boldsymbol{\lambda_\alpha} + \boldsymbol{r}$ when emulating $\mathcal{F}_{\mathsf{PrepMal}}$. When the computation is NOT marked as `fail`, for $\boldsymbol{\alpha}$, $\mathcal{S}$ learns $\boldsymbol{\mu_\alpha}$. $\mathcal{S}$ computes $\boldsymbol{v_\alpha} + \boldsymbol{r} = \boldsymbol{\mu_\alpha} + (\boldsymbol{\lambda_\alpha} + \boldsymbol{r})$. Then $\mathcal{S}$ sets $\Delta(\boldsymbol{v_\alpha}) = \widetilde{\boldsymbol{v_\alpha} + \boldsymbol{r}} - (\boldsymbol{v_\alpha} + \boldsymbol{r})$.

(b) If the computation is NOT marked as `fail`, $\mathcal{S}$ follows the protocol and computes the shares of $\{[\![v_{\alpha_i}|i]\!]_t\}_{i=1}^k$ of corrupted parties. Then $\mathcal{S}$ sends $\Delta(\boldsymbol{v_\alpha})$ and the shares of $\{[\![v_{\alpha_i}|i]\!]_t\}_{i=1}^k$ of corrupted parties to $\mathcal{F}_{\mathsf{Evaluate}}$.

Finally, $\mathcal{S}$ honestly follow the protocol $\Pi_{\mathsf{Consistency}}$. Note that if $P_1$ is honest, $\mathcal{S}$ has explicitly generated each degree-$(k-1)$ packed Shamir sharing that should be distributed by $P_1$. If $P_1$ is corrupted, $\mathcal{S}$ learns the shares of honest parties from $P_1$. In either case, $\mathcal{S}$ can honestly follow the protocol $\Pi_{\mathsf{Consistency}}$. If $\mathcal{S}$ has marked the computation as `fail` but no honest party aborts in $\Pi_{\mathsf{Consistency}}$, $\mathcal{S}$ aborts.

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

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

**Hybrid**$_0$: In this hybrid, $\mathcal{S}$ honestly follows the protocol.

**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 `fail`. $\mathcal{S}$ simulates $\Pi_{\mathsf{Consistency}}$ as described above. If $\mathcal{S}$ has marked the computation as `fail` but no honest party aborts in $\Pi_{\mathsf{Consistency}}$, $\mathcal{S}$ aborts.

By Lemma 6, 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, then all honest parties abort with overwhelming probability. Therefore the probability that $\mathcal{S}$ has marked the computation as `fail` but no honest party aborts in $\Pi_{\mathsf{Consistency}}$ is negligible. Thus, **Hybrid**$_1$ is statistically close to **Hybrid**$_0$.

**Hybrid**$_2$: In this hybrid, for each group of input gates that belong to some `Client`, let $\boldsymbol{\alpha}$ denote the output wires of these group of gates. For each $[\![v_{\alpha_i}|i]\!]_t$, $\mathcal{S}$ computes the secret $v_{\alpha_i}$ and the shares of corrupted parties by using the shares of honest parties. $\mathcal{S}$ sends $\boldsymbol{v_\alpha}$ and the shares of $\{[\![v_{\alpha_i}|i]\!]_t\}_{i=1}^k$ of corrupted parties to $\mathcal{F}_{\mathsf{Evaluate}}$.

Then, $\mathcal{S}$ computes $v_\alpha$ for each wire $\alpha$ and prepares the values for $\mathcal{F}_{\mathsf{Evaluate}}$ as follows:

– For each addition gate with input wires $\alpha, \beta$ and output wire $\gamma$, $\mathcal{S}$ computes $v_\gamma = v_\alpha + v_\beta$.
– For each group of multiplication gates with input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$ and output wire $\boldsymbol{\gamma}$, $\mathcal{S}$ computes the secrets $\{\widetilde{v_{\alpha_i}}, \widetilde{v_{\beta_i}}\}_{i=1}^k$ and the shares of $\{[\![v_{\alpha_i}|i]\!]_t, [\![v_{\beta_i}|i]\!]_t\}_{i=1}^k$ of corrupted parties by using the shares of honest parties. Then $\mathcal{S}$ computes $\Delta(\boldsymbol{v_\alpha}) = \widetilde{\boldsymbol{v_\alpha}} - \boldsymbol{v_\alpha}$ and $\Delta(\boldsymbol{v_\beta}) = \widetilde{\boldsymbol{v_\beta}} - \boldsymbol{v_\beta}$, where $\widetilde{\boldsymbol{v_\alpha}} = (\widetilde{v_{\alpha_1}}, \ldots, \widetilde{v_{\alpha_k}})$ and $\widetilde{\boldsymbol{v_\beta}} = (\widetilde{v_{\beta_1}}, \ldots, \widetilde{v_{\beta_k}})$. $\mathcal{S}$ sends $\Delta(\boldsymbol{v_\alpha}), \Delta(\boldsymbol{v_\beta})$ and the shares of $\{[\![v_{\alpha_i}|i]\!]_t, [\![v_{\beta_i}|i]\!]_t\}_{i=1}^k$ of corrupted parties to $\mathcal{F}_{\mathsf{Evaluate}}$.
  $\mathcal{S}$ computes $\boldsymbol{v_\gamma} = \widetilde{\boldsymbol{v_\alpha}} * \widetilde{\boldsymbol{v_\beta}}$.
– For each group of output gates with input wires $\boldsymbol{\alpha}$, $\mathcal{S}$ computes the secrets $\{\widetilde{v_{\alpha_i}}\}_{i=1}^k$ and the shares of $\{[\![v_{\alpha_i}|i]\!]_t\}_{i=1}^k$ of corrupted parties by using the shares of honest parties. Then $\mathcal{S}$ computes $\Delta(\boldsymbol{v_\alpha}) = \widetilde{\boldsymbol{v_\alpha}} - \boldsymbol{v_\alpha}$, where $\widetilde{\boldsymbol{v_\alpha}} = (\widetilde{v_{\alpha_1}}, \ldots, \widetilde{v_{\alpha_k}})$.
  $\mathcal{S}$ sends $\Delta(\boldsymbol{v_\alpha})$ and the shares of $\{[\![v_{\alpha_i}|i]\!]_t\}_{i=1}^k$ of corrupted parties to $\mathcal{F}_{\mathsf{Evaluate}}$.

Finally, honest parties take the shares from $\mathcal{F}_{\mathsf{Evaluate}}$ as output.

We prove that **Hybrid**$_2$ is identically distributed to **Hybrid**$_1$.

– For each group of input gates with output wires $\boldsymbol{\alpha}$, we show that the shares of $\{[\![v_{\alpha_i}|i]\!]_t\}_{i=1}^k$ of honest parties in both hybrids are identical. In **Hybrid**$_1$, honest parties take their shares computed in $\Pi_{\mathsf{Evaluate}}$ as output. In **Hybrid**$_2$, we first recover the secrets $\boldsymbol{v_\alpha}$ and the shares of $\{[\![v_{\alpha_i}|i]\!]_t\}_{i=1}^k$ of corrupted parties from the shares held by honest parties. These values are sent to $\mathcal{F}_{\mathsf{Evaluate}}$, and $\mathcal{F}_{\mathsf{Evaluate}}$ computes the shares of honest parties based on the secrets $\boldsymbol{v_\alpha}$ and the shares of $\{[\![v_{\alpha_i}|i]\!]_t\}_{i=1}^k$ of corrupted parties. Finally, honest parties take the shares computed by $\mathcal{F}_{\mathsf{Evaluate}}$ as output. Note that for a degree-$t$ Shamir sharing, it is determined by the shares of honest parties, and it is also determined by the shares of corrupted parties plus the secret. Thus, the shares computed by $\mathcal{F}_{\mathsf{Evaluate}}$ in **Hybrid**$_2$ are identical to the original shares held by honest parties.
  Thus, the shares of $\{[\![v_{\alpha_i}|i]\!]_t\}_{i=1}^k$ of honest parties in both hybrids are identical.
– Note that $\mathcal{S}$ computes $v_\alpha$ in the same way as that in $\mathcal{F}_{\mathsf{Evaluate}}$. For each group of multiplication gates with input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$ and output wires $\boldsymbol{\gamma}$, the values $\boldsymbol{v_\alpha}$ and $\boldsymbol{v_\beta}$ computed by $\mathcal{S}$ are identical to those computed by $\mathcal{F}_{\mathsf{Evaluate}}$. In **Hybrid**$_1$, honest parties output their shares of $\{[\![v_{\alpha_i}|i]\!]_t, [\![v_{\beta_i}|i]\!]_t\}_{i=1}^k$. In **Hybrid**$_2$, we first recover the secrets $\widetilde{\boldsymbol{v_\alpha}}, \widetilde{\boldsymbol{v_\beta}}$ and the shares of $\{[\![v_{\alpha_i}|i]\!]_t, [\![v_{\beta_i}|i]\!]_t\}_{i=1}^k$ of corrupted parties from the shares held by honest parties. Then, $\mathcal{S}$ sends the shares of corrupted parties and $\Delta(\boldsymbol{v_\alpha}) = \widetilde{\boldsymbol{v_\alpha}} - \boldsymbol{v_\alpha}, \Delta(\boldsymbol{v_\beta}) = \widetilde{\boldsymbol{v_\beta}} - \boldsymbol{v_\beta}$ to $\mathcal{F}_{\mathsf{Evaluate}}$. Since $\mathcal{F}_{\mathsf{Evaluate}}$ has computed the same values $\boldsymbol{v_\alpha}$ and $\boldsymbol{v_\beta}$, $\mathcal{F}_{\mathsf{Evaluate}}$ computes the shares of honest parties based on the secrets $\boldsymbol{v_\alpha} + \Delta(\boldsymbol{v_\alpha})$ and $\boldsymbol{v_\beta} + \Delta(\boldsymbol{v_\beta})$, which are just $\widetilde{\boldsymbol{v_\alpha}}, \widetilde{\boldsymbol{v_\beta}}$, and the shares of corrupted parties. Thus the shares computed by $\mathcal{F}_{\mathsf{Evaluate}}$ in **Hybrid**$_2$ are identical to the original shares held by honest parties.
  Thus, the shares of $\{[\![v_{\alpha_i}|i]\!]_t, [\![v_{\beta_i}|i]\!]_t\}_{i=1}^k$ of honest parties in both hybrids are identical.

- Following the same argument, we can show that for each group of output gates with input wires $\boldsymbol{\alpha}$, the shares of $\{\llbracket v_{\alpha_i}|i\rrbracket_t\}_{i=1}^k$ of honest parties in both hybrids are identical.

Thus, $\mathbf{Hybrid}_2$ is identically distributed to $\mathbf{Hybrid}_1$.

$\mathbf{Hybrid}_{3,0}$: This hybrid is identical to $\mathbf{Hybrid}_2$. From $\mathbf{Hybrid}_{3,0}$ to $\mathbf{Hybrid}_{3,6}$, we focus on the case where $P_1$ is an honest party.

$\mathbf{Hybrid}_{3,1}$: When $P_1$ is honest, $\mathcal{S}$ computes the shares of corrupted parties as follows.

- For each group of input gates with output wires $\boldsymbol{\alpha}$, recall that each degree-$t$ Shamir sharing $\llbracket v_{\alpha_i}|i\rrbracket_t$ is computed by
$$\llbracket v_{\alpha_i}|i\rrbracket_t = \llbracket \boldsymbol{v_\alpha} + \boldsymbol{r}\rrbracket_t - \llbracket r_i|i\rrbracket_t.$$
Recall that $\llbracket \boldsymbol{v_\alpha} + \boldsymbol{r}\rrbracket_t$ is distributed by the `Client` that holds inputs for these input gates. $\mathcal{S}$ computes the shares of corrupted parties from the shares of honest parties. For $\llbracket r_i|i\rrbracket_t$, $\mathcal{S}$ receives from the adversary the shares of corrupted parties when emulating $\mathcal{F}_{\mathsf{PrepMal}}$. Then, $\mathcal{S}$ computes the shares of $\llbracket v_{\alpha_i}|i\rrbracket_t$ of corrupted parties.
- For each group of multiplication gates with input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$, each degree-$t$ Shamir sharing $\llbracket v_{\alpha_i}|i\rrbracket_t$ is computed by
$$\llbracket v_{\alpha_i}|i\rrbracket_t = \llbracket \boldsymbol{v_\alpha} + \boldsymbol{a}\rrbracket_{k-1} - \llbracket a_i|i\rrbracket_t.$$
Recall that $\llbracket \boldsymbol{v_\alpha} + \boldsymbol{a}\rrbracket_{k-1}$ is distributed by $P_1$. $\mathcal{S}$ computes the shares of corrupted parties from the shares of honest parties. For $\llbracket a_i|i\rrbracket_t$, $\mathcal{S}$ receives from the adversary the shares of corrupted parties when emulating $\mathcal{F}_{\mathsf{PrepMal}}$. Then, $\mathcal{S}$ computes the shares of $\llbracket v_{\alpha_i}|i\rrbracket_t$ of corrupted parties.
Similarly, $\mathcal{S}$ computes the shares of $\llbracket v_{\beta_t}|i\rrbracket_t$ of corrupted parties.
- For each group of output gates with input wires $\boldsymbol{\alpha}$, $\mathcal{S}$ computes the shares of $\llbracket v_{\beta_t}|i\rrbracket_t$ of corrupted parties in the same way as that for the input wires of multiplication gates.

The distribution of $\mathbf{Hybrid}_{3,1}$ is identical to that of $\mathbf{Hybrid}_{3,0}$.

$\mathbf{Hybrid}_{3,2}$: When $P_1$ is honest, $\mathcal{S}$ computes $\Delta(\boldsymbol{v_\alpha}), \Delta(\boldsymbol{v_\beta})$ for each group of multiplication gates and computes $\Delta(\boldsymbol{v_\alpha})$ for each group of output gates as follows.

For each group of multiplication gates with input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$, recall that $\llbracket v_{\alpha_i}|i\rrbracket_t = \llbracket \boldsymbol{v_\alpha} + \boldsymbol{a}\rrbracket_t - \llbracket a_i|i\rrbracket_t$. Since $\llbracket a_i|i\rrbracket_t$ is prepared by $\mathcal{F}_{\mathsf{PrepMal}}$ and the secret is determined by the shares of honest parties, the adversary cannot insert any additive error to the secret $a_i$. We have $\Delta(v_{\alpha_i}) = \Delta(v_{\alpha_i} + a_i)$, which means that $\Delta(\boldsymbol{v_\alpha}) = \Delta(\boldsymbol{v_\alpha} + \boldsymbol{a})$. Recall that $\boldsymbol{v_\alpha} + \boldsymbol{a}$ is computed by $\boldsymbol{\mu_\alpha} + (\boldsymbol{\lambda_\alpha} + \boldsymbol{a})$. $\mathcal{S}$ has received $\Delta(\boldsymbol{\lambda_\alpha} + \boldsymbol{a})$. Thus $\mathcal{S}$ only needs to compute $\Delta(\boldsymbol{\mu_\alpha})$.

Since $P_1$ reconstructs $\{\mu_\alpha\}_\alpha$ for the output wires of input gates and multiplication gates, it is sufficient to first compute $\{\Delta(\mu_\alpha)\}_\alpha$ for the output wires of input gates and multiplication gates, and then compute $\{\Delta(\mu_\alpha)\}_\alpha$ for the input wires of multiplication gates. For each group of input gates with output wires $\boldsymbol{\alpha}$,

- If `Client` is honest, let $\llbracket \boldsymbol{\lambda_\alpha}\rrbracket_{n-1}$ denote the sharing that `Client` receives from all parties, and $\llbracket \overline{\boldsymbol{\lambda_\alpha}}\rrbracket_{n-1}$ denote the sharing where the shares of corrupted parties are replaced by those learnt by $\mathcal{S}$ when emulating $\mathcal{F}_{\mathsf{PrepMal}}$. $\mathcal{S}$ computes the shares of corrupted parties of the following sharing
$$\llbracket \rho(\boldsymbol{\lambda_\alpha})\rrbracket_{n-1} = \llbracket \boldsymbol{\lambda_\alpha}\rrbracket_{n-1} - \llbracket \overline{\boldsymbol{\lambda_\alpha}}\rrbracket_{n-1}.$$
$\mathcal{S}$ sets the shares of honest parties to be $0$ and reconstructs the secrets $\rho(\boldsymbol{\lambda_\alpha})$. Then the secrets of $\llbracket \boldsymbol{\lambda_\alpha}\rrbracket_{n-1}$, $\widetilde{\boldsymbol{\lambda_\alpha}}$, are equal to $\overline{\boldsymbol{\lambda_\alpha}} + \rho(\boldsymbol{\lambda_\alpha})$.
On the other hand, when emulating $\mathcal{F}_{\mathsf{PrepMal}}$, $\mathcal{S}$ receives $\Delta(\boldsymbol{\lambda_\alpha})$ and we have $\overline{\boldsymbol{\lambda_\alpha}} = \boldsymbol{\lambda_\alpha} + \Delta(\boldsymbol{\lambda_\alpha})$.
Therefore, $\widetilde{\boldsymbol{\lambda_\alpha}} = \boldsymbol{\lambda_\alpha} + \rho(\boldsymbol{\lambda_\alpha}) + \Delta(\boldsymbol{\lambda_\alpha})$.
Since `Client` sends $\widetilde{\boldsymbol{\mu_\alpha}} = \boldsymbol{v_\alpha} - \widetilde{\boldsymbol{\lambda_\alpha}} = \boldsymbol{\mu_\alpha} - (\rho(\boldsymbol{\lambda_\alpha}) + \Delta(\boldsymbol{\lambda_\alpha}))$ to $P_1$, we have $\Delta(\mu_\alpha) = -(\rho(\boldsymbol{\lambda_\alpha}) + \Delta(\boldsymbol{\lambda_\alpha}))$. In this way, $\mathcal{S}$ computes $\Delta(\boldsymbol{\mu_\alpha})$ from $\rho(\boldsymbol{\lambda_\alpha})$ and $\Delta(\boldsymbol{\lambda_\alpha})$.
- If `Client` is corrupted, recall that $\mathcal{S}$ has computed $\boldsymbol{v_\alpha}$ in $\mathbf{Hybrid}_{3,1}$. $\mathcal{S}$ computes $\boldsymbol{\mu_\alpha} = \boldsymbol{v_\alpha} - \boldsymbol{\lambda_\alpha}$. After $\mathcal{S}$ receives $\widetilde{\boldsymbol{\mu_\alpha}}$ from `Client`, $\mathcal{S}$ computes $\Delta(\boldsymbol{\mu_\alpha}) = \widetilde{\boldsymbol{\mu_\alpha}} - \boldsymbol{\mu_\alpha}$.

For each group of multiplication gates with output wires $\boldsymbol{\gamma}$, $\mathcal{S}$ computes the shares of $\llbracket \boldsymbol{\mu_\gamma}\rrbracket_{n-1}$ of corrupted parties by following Step (3) of $\Pi_{\mathsf{MultMal}}$. $\mathcal{S}$ receives from corrupted parties their shares of $\llbracket \boldsymbol{\mu_\gamma}\rrbracket_{n-1}$. Let $\llbracket \overline{\boldsymbol{\mu_\gamma}}\rrbracket_{n-1}$ denote the sharing where the shares of corrupted parties are replaced by those computed by $\mathcal{S}$. $\mathcal{S}$ computes the shares of corrupted parties of the following sharing
$$\llbracket \rho(\boldsymbol{\mu_\gamma})\rrbracket_{n-1} = \llbracket \boldsymbol{\mu_\gamma}\rrbracket_{n-1} - \llbracket \overline{\boldsymbol{\mu_\gamma}}\rrbracket_{n-1}.$$

$\mathcal{S}$ sets the shares of honest parties to be $0$ and reconstructs the secrets $\rho(\boldsymbol{\mu}_\gamma)$. Then the secrets of $[\![\boldsymbol{\mu}_\gamma]\!]_{n-1}$, $\widetilde{\boldsymbol{\mu}_\gamma}$, are equal to $\overline{\boldsymbol{\mu}_\gamma} + \rho(\boldsymbol{\mu}_\gamma)$.

On the other hand, when emulating $\mathcal{F}_{\mathsf{PrepMal}}$, $\mathcal{S}$ receives $\Delta(\boldsymbol{\lambda}_\gamma)$ and $\boldsymbol{\eta} = (\eta_1, \ldots, \eta_k)$. We have $\overline{\boldsymbol{\mu}_\gamma} = \boldsymbol{\mu}_\gamma + \boldsymbol{\eta} - \Delta(\boldsymbol{\lambda}_\gamma)$. Therefore, $\widetilde{\boldsymbol{\mu}_\gamma} = \boldsymbol{\mu}_\gamma + \rho(\boldsymbol{\mu}_\gamma) + \boldsymbol{\eta} - \Delta(\boldsymbol{\lambda}_\gamma)$. In this way, $\mathcal{S}$ computes $\Delta(\boldsymbol{\mu}_\gamma)$ from $\rho(\boldsymbol{\mu}_\gamma), \boldsymbol{\eta}, \Delta(\boldsymbol{\lambda}_\gamma)$.

The distribution of $\mathbf{Hybrid}_{3,2}$ is identical to that of $\mathbf{Hybrid}_{3,1}$.

$\mathbf{Hybrid}_{3,3}$: When $P_1$ is honest, for each group of multiplications with input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$, $\mathcal{S}$ randomly samples two vectors as $\boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \boldsymbol{a}, \boldsymbol{\lambda}_{\boldsymbol{\beta}} + \boldsymbol{b}$ and then computes $\boldsymbol{a} = (\boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \boldsymbol{a}) - \boldsymbol{\lambda}_{\boldsymbol{\alpha}}, \boldsymbol{b} = (\boldsymbol{\lambda}_{\boldsymbol{\beta}} + \boldsymbol{b}) - \boldsymbol{\lambda}_{\boldsymbol{\beta}}$. For each group of output gates with input wires $\boldsymbol{\alpha}$, $\mathcal{S}$ randomly samples a vector as $\boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \boldsymbol{r}$ and then computes $\boldsymbol{r} = (\boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \boldsymbol{r}) - \boldsymbol{\lambda}_{\boldsymbol{\alpha}}$.

The difference is that in $\mathbf{Hybrid}_{3,2}$, $\mathcal{S}$ first randomly samples $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{r}$ and then computes $\boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \boldsymbol{a}, \boldsymbol{\lambda}_{\boldsymbol{\beta}} + \boldsymbol{b}$ for multiplication gates and $\boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \boldsymbol{r}$ for output gates. Note that the distribution of these values are unchanged.

The distribution of $\mathbf{Hybrid}_{3,3}$ is identical to that of $\mathbf{Hybrid}_{3,2}$.

$\mathbf{Hybrid}_{3,4}$: When $P_1$ is honest, for each group of multiplication gates with output wires $\gamma$, $\mathcal{S}$ samples random values as the shares of $[\![\overline{\boldsymbol{\mu}_\gamma}]\!]_{n-1}$ (defined in $\mathbf{Hybrid}_{3,2}$) of honest parties. Then, $\mathcal{S}$ uses the shares of corrupted parties (computed in $\mathbf{Hybrid}_{3,2}$) to compute $\overline{\boldsymbol{\mu}_\gamma}$. Next, $\mathcal{S}$ computes $\boldsymbol{\mu}_\gamma = \overline{\boldsymbol{\mu}_\gamma} - \boldsymbol{\eta} + \Delta(\boldsymbol{\lambda}_\gamma)$ and computes $\boldsymbol{\lambda}_\gamma = \boldsymbol{v}_\gamma - \boldsymbol{\mu}_\gamma$.

In $\mathbf{Hybrid}_{3,3}$, the degree-$(n-1)$ packed Shamir sharing $[\![\boldsymbol{\lambda}_\gamma]\!]_{n-1}$ generated by $\mathcal{F}_{\mathsf{PrepMal}}$ is a random degree-$(n-1)$ packed Shamir sharing given the shares of corrupted parties. This is because the secrets are equal to $\boldsymbol{\lambda}_\gamma + \Delta(\boldsymbol{\lambda}_\gamma)$ and $\boldsymbol{\lambda}_\gamma$ are uniformly random. Therefore, $[\![\overline{\boldsymbol{\mu}_\gamma}]\!]_{n-1}$ is a random degree-$(n-1)$ packed Shamir sharing given the shares of corrupted parties. In particular, the shares of honest parties are uniformly distributed and independent of the shares of corrupted parties. Thus, the shares of $[\![\overline{\boldsymbol{\mu}_\gamma}]\!]_{n-1}$ are identically distributed in both $\mathbf{Hybrid}_{3,3}$ and $\mathbf{Hybrid}_{3,4}$.

By using the shares of $[\![\overline{\boldsymbol{\mu}_\gamma}]\!]_{n-1}$ of corrupted parties, we can reconstruct the secrets $\overline{\boldsymbol{\mu}_\gamma}$. From the argument in $\mathbf{Hybrid}_{3,2}$, we have $\overline{\boldsymbol{\mu}_\gamma} = \boldsymbol{\mu}_\gamma + \boldsymbol{\eta} - \Delta(\boldsymbol{\lambda}_\gamma)$. We also have $\boldsymbol{v}_\gamma = \boldsymbol{\mu}_\gamma + \boldsymbol{\lambda}_\gamma$. Thus, we can compute $\boldsymbol{\lambda}_\gamma$ from $\overline{\boldsymbol{\mu}_\gamma}$.

The distribution of $\mathbf{Hybrid}_{3,4}$ is identical to that of $\mathbf{Hybrid}_{3,3}$.

$\mathbf{Hybrid}_{3,5}$: When $P_1$ is honest, $\mathcal{S}$ simulates $\Pi_{\mathsf{InputMal}}$ as described above.

- When `Client` is honest, $\mathcal{S}$ first checks the shares of $\{[\![r_i|_i]\!]_t\}_{i=1}^k$ on behalf of `Client`. $\mathcal{S}$ aborts on behalf of `Client` if the shares of corrupted parties are different from those received from the adversary when emulating $\mathcal{F}_{\mathsf{PrepMal}}$. Note that a degree-$t$ Shamir sharing is determined by the shares of honest parties. In $\mathbf{Hybrid}_{3,4}$, honest parties always use the correct shares. If corrupted parties use different shares from those sent to $\mathcal{F}_{\mathsf{PrepMal}}$, `Client` will abort. Thus, $\mathcal{S}$ aborts on behalf of `Client` in $\mathbf{Hybrid}_{3,5}$ if and only if `Client` aborts in $\mathbf{Hybrid}_{3,4}$.
  Then, $\mathcal{S}$ samples a random vector as $\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{r}$ and generates a random degree-$t$ packed Shamir sharing $[\![\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{r}]\!]_t$. In $\mathbf{Hybrid}_{3,4}$, $\boldsymbol{r}$ are uniformly random. Therefore, $\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{r}$ are also uniformly random. Also $[\![\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{r}]\!]_t$ is a random degree-$t$ packed Shamir sharing of $[\![\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{r}]\!]_t$. Thus, the distribution of $[\![\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{r}]\!]_t$ is identical in both hybrids.
  Next, $\mathcal{S}$ samples a random vector as $\boldsymbol{\mu}_{\boldsymbol{\alpha}}$ and computes $\boldsymbol{\lambda}_{\boldsymbol{\alpha}} = \boldsymbol{v}_{\boldsymbol{\alpha}} - \boldsymbol{\mu}_{\boldsymbol{\alpha}}$. In $\mathbf{Hybrid}_{3,4}$, $\boldsymbol{\lambda}_{\boldsymbol{\alpha}}$ are uniformly random. Therefore, $\boldsymbol{\mu}_{\boldsymbol{\alpha}}$ are also uniformly random. Thus, the vector $\boldsymbol{\mu}_{\boldsymbol{\alpha}}$ has the same distribution in both hybrids.
  Finally, $\mathcal{S}$ computes $\boldsymbol{\mu}_{\boldsymbol{\alpha}} + \Delta(\boldsymbol{\mu}_{\boldsymbol{\alpha}})$ and sends them to $P_1$. Here $\Delta(\boldsymbol{\mu}_{\boldsymbol{\alpha}})$ are computed in the same way as that in $\mathbf{Hybrid}_{3,4}$ (described in $\mathbf{Hybrid}_{3,2}$).
- When `Client` is corrupted, $\mathcal{S}$ first generates a random vector $\boldsymbol{r}$ and then computes the shares of $\{[\![r_i|_i]\!]_t\}_{i=1}^k$ of honest parties based on the shares of corrupted parties. Note that the way of generating $\{[\![r_i|_i]\!]_t\}_{i=1}^k$ is identical to that in $\mathcal{F}_{\mathsf{PrepMal}}$ in $\mathbf{Hybrid}_{3,4}$. Thus, the shares of $\{[\![r_i|_i]\!]_t\}_{i=1}^k$ of honest parties have the same distribution in both hybrids.
  Then, $\mathcal{S}$ first generates a random vector $\boldsymbol{\lambda}_{\boldsymbol{\alpha}}$ and samples a random degree-$(n-1)$ packed Shamir sharing $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \Delta(\boldsymbol{\lambda}_{\boldsymbol{\alpha}})]\!]$, where $\Delta(\boldsymbol{\lambda}_{\boldsymbol{\alpha}})$ is received from the adversary when emulating $\mathcal{F}_{\mathsf{PrepMal}}$. Note that the way of generating $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \Delta(\boldsymbol{\lambda}_{\boldsymbol{\alpha}})]\!]$ is identical to that in $\mathcal{F}_{\mathsf{PrepMal}}$ in $\mathbf{Hybrid}_{3,4}$. Thus, the shares of $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \Delta(\boldsymbol{\lambda}_{\boldsymbol{\alpha}})]\!]$ of honest parties have the same distribution in both hybrids.
  Next, $\mathcal{S}$ receives the shares of $[\![\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{r}]\!]_t$ of honest parties and reconstructs $\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{r}$. $\mathcal{S}$ computes $\boldsymbol{v}_{\boldsymbol{\alpha}} = (\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{r}) - \boldsymbol{r}$. In $\mathbf{Hybrid}_{3,4}$, $\boldsymbol{v}_{\boldsymbol{\alpha}}$ are computed from the shares of $\{[\![v_{\alpha_i}|_i]\!]_t\}_{i=1}^k$ of honest parties. Recall that $[\![v_{\alpha_i}|_i]\!]_t = [\![\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{r}]\!]_t - [\![r_i|_i]\!]_t$. Therefore, $v_{\alpha_i} = (v_{\alpha_i} + r_i) - r_i$, which means that $\boldsymbol{v}_{\boldsymbol{\alpha}} = (\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{r}) - \boldsymbol{r}$. The inputs extracted by $\mathcal{S}$ have the same distribution as those computed in $\mathbf{Hybrid}_{3,4}$.

In summary, the distribution of $\mathbf{Hybrid}_{3,5}$ is identical to that of $\mathbf{Hybrid}_{3,4}$.

$\mathbf{Hybrid}_{3,6}$: We observe that when $P_1$ is honest, all messages sent to corrupted parties and corrupted clients, and all values sent to $\mathcal{F}_{\mathsf{Evaluate}}$ have been simulated by $\mathcal{S}$ *without knowing the inputs of honest parties* in $\mathbf{Hybrid}_{3,5}$. The inputs of honest parties are only used to generate the views of honest parties. In this hybrid, when $P_1$ is honest, $\mathcal{S}$ simulates the whole protocol $\Pi_{\mathsf{Evaluate}}$ as described above. The distribution of $\mathbf{Hybrid}_{3,6}$ is identical to that of $\mathbf{Hybrid}_{3,5}$.

$\mathbf{Hybrid}_{4,0}$: This hybrid is identical to $\mathbf{Hybrid}_{3,6}$. From $\mathbf{Hybrid}_{4,0}$ to $\mathbf{Hybrid}_{4,5}$, we focus on the case where $P_1$ is a corrupted party.

$\mathbf{Hybrid}_{4,1}$: When $P_1$ is corrupted, $\mathcal{S}$ computes the shares of corrupted parties as follows.

– For each group of input gates with output wires $\boldsymbol{\alpha}$, recall that each degree-$t$ Shamir sharing $[\![v_{\alpha_i}|i]\!]_t$ is computed by

$$[\![v_{\alpha_i}|i]\!]_t = [\![\boldsymbol{v_\alpha} + \boldsymbol{r}]\!]_t - [\![r_i|i]\!]_t.$$

Recall that $[\![\boldsymbol{v_\alpha} + \boldsymbol{r}]\!]_t$ is distributed by the `Client` that holds inputs for these input gates. $\mathcal{S}$ computes the shares of corrupted parties from the shares of honest parties. For $[\![r_i|i]\!]_t$, $\mathcal{S}$ receives from the adversary the shares of corrupted parties when emulating $\mathcal{F}_{\mathsf{PrepMal}}$. Then, $\mathcal{S}$ computes the shares of $[\![v_{\alpha_i}|i]\!]_t$ of corrupted parties.

– For each group of multiplication gates with input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$, each degree-$t$ Shamir sharing $[\![v_{\alpha_i}|i]\!]_t$ is computed by

$$[\![v_{\alpha_i}|i]\!]_t = [\![\boldsymbol{v_\alpha} + \boldsymbol{a}]\!]_{k-1} - [\![a_i|i]\!]_t.$$

Recall that $[\![\boldsymbol{v_\alpha} + \boldsymbol{a}]\!]_{k-1}$ is distributed by $P_1$.
  - If the computation is marked as `fail`, $\mathcal{S}$ sets the shares of $[\![v_{\alpha_i}|i]\!]_t$ of corrupted parties to be all 0.
  - Otherwise, $[\![\boldsymbol{v_\alpha} + \boldsymbol{a}]\!]_{k-1}$ is a valid degree-$(k-1)$ packed Shamir sharing. $\mathcal{S}$ computes the shares of corrupted parties from the shares of honest parties. For $[\![a_i|i]\!]_t$, $\mathcal{S}$ receives from the adversary the shares of corrupted parties when emulating $\mathcal{F}_{\mathsf{PrepMal}}$. Then, $\mathcal{S}$ computes the shares of $[\![v_{\alpha_i}|i]\!]_t$ of corrupted parties.

  Similarly, $\mathcal{S}$ computes the shares of $[\![v_{\beta_t}|i]\!]_t$ of corrupted parties.

– For each group of output gates with input wires $\boldsymbol{\alpha}$, $\mathcal{S}$ computes the shares of $[\![v_{\beta_t}|i]\!]_t$ of corrupted parties in the same way as that for the input wires of multiplication gates.

Observe that the shares of corrupted parties computed in $\mathbf{Hybrid}_{4,1}$ is different from those in $\mathbf{Hybrid}_{4,0}$ if and only if some degree-$(k-1)$ packed Shamir sharing distributed by $P_1$ is invalid in the sense that the shares of honest parties do not lie on a degree-$(k-1)$ polynomial. In this case, both $\mathbf{Hybrid}_{4,0}$ and $\mathbf{Hybrid}_{4,1}$ will abort. Therefore, the distribution of $\mathbf{Hybrid}_{4,1}$ is identical to that of $\mathbf{Hybrid}_{4,0}$.

$\mathbf{Hybrid}_{4,2}$: When $P_1$ is corrupted, $\mathcal{S}$ computes $\Delta(\boldsymbol{v_\alpha}), \Delta(\boldsymbol{v_\beta})$ for each group of multiplication gates and computes $\Delta(\boldsymbol{v_\alpha})$ for each group of output gates as follows.

For each group of multiplication gates with input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$, recall that $[\![v_{\alpha_i}|i]\!]_t = [\![\boldsymbol{v_\alpha} + \boldsymbol{a}]\!]_t - [\![a_i|i]\!]_t$. Since $[\![a_i|i]\!]_t$ is prepared by $\mathcal{F}_{\mathsf{PrepMal}}$ and the secret is determined by the shares of honest parties, the adversary cannot insert any additive error to the secret $a_i$. We have $\Delta(v_{\alpha_i}) = \Delta(v_{\alpha_i} + a_i)$, which means that $\Delta(\boldsymbol{v_\alpha}) = \Delta(\boldsymbol{v_\alpha} + \boldsymbol{a})$. Recall that $[\![\boldsymbol{v_\alpha} + \boldsymbol{a}]\!]_{k-1}$ is distributed by $P_1$.

– If the computation is marked as `fail`, $\mathcal{S}$ sets $\Delta(\boldsymbol{v_\alpha} + \boldsymbol{a}) = \mathbf{0}$.
– Otherwise, $[\![\boldsymbol{v_\alpha} + \boldsymbol{a}]\!]_{k-1}$ is a valid degree-$(k-1)$ packed Shamir sharing. $\mathcal{S}$ computes the secrets $\widetilde{\boldsymbol{v_\alpha} + \boldsymbol{a}}$ and the shares of corrupted parties.

To compute $\Delta(\boldsymbol{v_\alpha} + \boldsymbol{a})$, it is sufficient to compute the correct values $\boldsymbol{v_\alpha} + \boldsymbol{a}$. Recall that $\boldsymbol{v_\alpha} + \boldsymbol{a}$ is computed by $\boldsymbol{\mu_\alpha} + (\boldsymbol{\lambda_\alpha} + \boldsymbol{a})$. $\mathcal{S}$ has generated $\boldsymbol{\lambda_\alpha} + \boldsymbol{a}$ when emulating $\mathcal{F}_{\mathsf{PrepMal}}$. Thus $\mathcal{S}$ only needs to compute $\boldsymbol{\mu_\alpha}$.

$\mathcal{S}$ will first compute $\{\Delta(\mu_\alpha)\}_\alpha$ for the output wires of input gates and multiplication gates, and then compute $\{\mu_\alpha\}_\alpha$ for the input wires of multiplication gates. For each group of input gates with output wires $\boldsymbol{\alpha}$, recall that $\mathcal{S}$ has computed $\boldsymbol{v_\alpha}$ in $\mathbf{Hybrid}_2$. $\mathcal{S}$ has also generated $\boldsymbol{\lambda_\alpha}$ when emulating $\mathcal{F}_{\mathsf{PrepMal}}$. $\mathcal{S}$ computes $\boldsymbol{\mu_\alpha} = \boldsymbol{v_\alpha} - \boldsymbol{\lambda_\alpha}$.

For each group of multiplication gates with input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$ and output wires $\boldsymbol{\gamma}$, we have $\boldsymbol{v_\gamma} = \widetilde{\boldsymbol{v_\alpha}} * \widetilde{\boldsymbol{v_\beta}}$ in $\mathcal{F}_{\mathsf{Evaluate}}$.

– If the computation is marked as `fail`, $\mathcal{S}$ does nothing. Note that in this case, $\mathcal{S}$ will always use $\mathbf{0}$ as the vectors of additive errors. There is no need to compute $\boldsymbol{\mu_\gamma}$.

– Otherwise, $[\![\boldsymbol{v}_{\boldsymbol{\alpha}}+\boldsymbol{a}]\!]_{k-1}$ and $[\![\boldsymbol{v}_{\boldsymbol{\beta}}+\boldsymbol{b}]\!]_{k-1}$ are valid degree-$(k-1)$ packed Shamir sharings. In particular, $\widetilde{\boldsymbol{v}_{\boldsymbol{\alpha}}+\boldsymbol{a}}=\widetilde{\boldsymbol{v}_{\boldsymbol{\alpha}}}+\boldsymbol{a}$ and $\widetilde{\boldsymbol{v}_{\boldsymbol{\beta}}+\boldsymbol{b}}=\widetilde{\boldsymbol{v}_{\boldsymbol{\beta}}}+\boldsymbol{b}$. Again, this is because each value in $\boldsymbol{a},\boldsymbol{b}$ is shared by a degree-$t$ Shamir sharing. The adversary cannot insert additive errors to the values $\boldsymbol{a},\boldsymbol{b}$.

$\mathcal{S}$ computes the shares of $[\![\boldsymbol{\mu}_\gamma]\!]_{n-1}$ of corrupted parties by following Step (3) of $\Pi_{\mathsf{MultMal}}$. Let $[\![\overline{\boldsymbol{\mu}_\gamma}]\!]_{n-1}$ denote the sharing where the shares of corrupted parties are those computed by $\mathcal{S}$. $\mathcal{S}$ reconstructs the secrets $\overline{\boldsymbol{\mu}_\gamma}$ by using the shares of all parties. Note that, $[\![\boldsymbol{v}_{\boldsymbol{\alpha}}+\boldsymbol{a}]\!]_{k-1}$ and $[\![\boldsymbol{v}_{\boldsymbol{\beta}}+\boldsymbol{b}]\!]_{k-1}$ are valid degree-$(k-1)$ packed Shamir sharings, we have

$$\begin{aligned}\overline{\boldsymbol{\mu}_\gamma}&=(\widetilde{\boldsymbol{v}_{\boldsymbol{\alpha}}}+\boldsymbol{a})*(\widetilde{\boldsymbol{v}_{\boldsymbol{\beta}}}+\boldsymbol{b})-(\widetilde{\boldsymbol{v}_{\boldsymbol{\alpha}}}+\boldsymbol{a})*\boldsymbol{b}\\&\quad-(\widetilde{\boldsymbol{v}_{\boldsymbol{\beta}}}+\boldsymbol{b})*\boldsymbol{a}+\boldsymbol{c}-(\boldsymbol{\lambda}_\gamma+\Delta(\boldsymbol{\lambda}_\gamma)).\end{aligned}$$

The last term is because the secrets of $[\![\boldsymbol{\lambda}_\gamma]\!]_{n-1}$ distributed by $\mathcal{F}_{\mathsf{PrepMal}}$ are equal to $\boldsymbol{\lambda}_\gamma+\Delta(\boldsymbol{\lambda}_\gamma)$. Recall that $\mathcal{F}_{\mathsf{Evaluate}}$ receives $\boldsymbol{\eta}=(\eta_1,\ldots,\eta_k)$ from the adversary and computes $c_i=a_i\cdot b_i+\eta_i$. Thus, we have

$$\overline{\boldsymbol{\mu}_\gamma}=\widetilde{\boldsymbol{v}_{\boldsymbol{\alpha}}}*\widetilde{\boldsymbol{v}_{\boldsymbol{\beta}}}+\boldsymbol{\eta}-(\boldsymbol{\lambda}_\gamma+\Delta(\boldsymbol{\lambda}_\gamma)).$$

Therefore, $\boldsymbol{\mu}_\gamma=\overline{\boldsymbol{\mu}_\gamma}-\boldsymbol{\eta}+\Delta(\boldsymbol{\lambda}_\gamma$. Since $\mathcal{S}$ receives $\boldsymbol{\eta}$ and $\Delta(\boldsymbol{\lambda}_\gamma)$ from the adversary when emulating $\mathcal{F}_{\mathsf{PrepMal}}$, $\mathcal{S}$ computes $\boldsymbol{\mu}_\gamma$ from $\overline{\boldsymbol{\mu}_\gamma},\boldsymbol{\eta},\Delta(\boldsymbol{\lambda}_\gamma$.

Observe that the vectors of additive errors computed in **Hybrid**$_{4,2}$ is different from those in **Hybrid**$_{4,1}$ if and only if some degree-$(k-1)$ packed Shamir sharing distributed by $P_1$ is invalid in the sense that the shares of honest parties do not lie on a degree-$(k-1)$ polynomial. In this case, both **Hybrid**$_{4,1}$ and **Hybrid**$_{4,2}$ will abort. Therefore, the distribution of **Hybrid**$_{4,2}$ is identical to that of **Hybrid**$_{4,1}$.

**Hybrid**$_{4,3}$: When $P_1$ is corrupted, for each group of multiplications with input wires $\boldsymbol{\alpha},\boldsymbol{\beta}$, $\mathcal{S}$ randomly samples two vectors as $\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a},\boldsymbol{\lambda}_{\boldsymbol{\beta}}+\boldsymbol{b}$ and then computes $\boldsymbol{a}=(\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a})-\boldsymbol{\lambda}_{\boldsymbol{\alpha}},\boldsymbol{b}=(\boldsymbol{\lambda}_{\boldsymbol{\beta}}+\boldsymbol{b})-\boldsymbol{\lambda}_{\boldsymbol{\beta}}$. For each group of output gates with input wires $\boldsymbol{\alpha}$, $\mathcal{S}$ randomly samples a vector as $\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{r}$ and then computes $\boldsymbol{r}=(\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{r})-\boldsymbol{\lambda}_{\boldsymbol{\alpha}}$.

The difference is that in **Hybrid**$_{4,2}$, $\mathcal{S}$ first randomly samples $\boldsymbol{a},\boldsymbol{b},\boldsymbol{r}$ and then computes $\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{a},\boldsymbol{\lambda}_{\boldsymbol{\beta}}+\boldsymbol{b}$ for multiplication gates and $\boldsymbol{\lambda}_{\boldsymbol{\alpha}}+\boldsymbol{r}$ for output gates. Note that the distribution of these values are unchanged.

The distribution of **Hybrid**$_{4,3}$ is identical to that of **Hybrid**$_{4,2}$.

**Hybrid**$_{4,4}$: When $P_1$ is corrupted, for each group of multiplication gates with output wires $\gamma$, $\mathcal{S}$ samples random values as the shares of $[\![\overline{\boldsymbol{\mu}_\gamma}]\!]_{n-1}$ (defined in **Hybrid**$_{4,2}$) of honest parties.

– If the computation is marked as `fail`, $\mathcal{S}$ does nothing.
– Otherwise, $\mathcal{S}$ uses the shares of corrupted parties (computed in **Hybrid**$_{4,2}$) to compute $\overline{\boldsymbol{\mu}_\gamma}$. Next, $\mathcal{S}$ computes $\boldsymbol{\mu}_\gamma=\overline{\boldsymbol{\mu}_\gamma}-\boldsymbol{\eta}+\Delta(\boldsymbol{\lambda}_\gamma)$.

For each group of multiplication gates, $\mathcal{S}$ does not generate the shares of $\{([\![a_i|_i]\!]_t,[\![b_i|_i]\!]_t,[\![c_i|_i]\!]_t)\}_{i=1}^k$ and $[\![\boldsymbol{\lambda}_\gamma]\!]_{n-1}$ of honest parties and does not compute $(\boldsymbol{a},\boldsymbol{b},\boldsymbol{c})$ and $\boldsymbol{\lambda}_\gamma$. For each group of output gates, $\mathcal{S}$ does not generate the shares of $\{[\![r_i|_i]\!]_t\}_{i=1}^k$ of honest parties and does not compute $\boldsymbol{r}$. These values are no longer needed in **Hybrid**$_{4,4}$.

In **Hybrid**$_{4,3}$, the degree-$(n-1)$ packed Shamir sharing $[\![\boldsymbol{\lambda}_\gamma]\!]_{n-1}$ generated by $\mathcal{F}_{\mathsf{PrepMal}}$ is a random degree-$(n-1)$ packed Shamir sharing given the shares of corrupted parties. This is because the secrets are equal to $\boldsymbol{\lambda}_\gamma+\Delta(\boldsymbol{\lambda}_\gamma)$ and $\boldsymbol{\lambda}_\gamma$ are uniformly random. Then the shares of $[\![\boldsymbol{\lambda}_\gamma]\!]_{n-1}$ of honest parties are uniformly random and independent of the shares of corrupted parties. Since $[\![\overline{\boldsymbol{\mu}_\gamma}]\!]_{n-1}$ is masked by $[\![\boldsymbol{\lambda}_\gamma]\!]_{n-1}$, the shares of honest parties are uniformly distributed and independent of the shares of corrupted parties. Thus, the shares of $[\![\overline{\boldsymbol{\mu}_\gamma}]\!]_{n-1}$ are identically distributed in both **Hybrid**$_{4,3}$ and **Hybrid**$_{4,4}$.

The distribution of **Hybrid**$_{4,4}$ is identical to that of **Hybrid**$_{4,3}$. At this point, the behaviors of honest parties in Step 3 and Step 4 of $\Pi_{\mathsf{Evaluate}}$ are fully simulated by $\mathcal{S}$ without relying on honest parties' inputs.

**Hybrid**$_{4,5}$: When $P_1$ is corrupted, $\mathcal{S}$ simulates $\Pi_{\mathsf{InputMal}}$ as described above.

– When `Client` is honest, $\mathcal{S}$ first checks the shares of $\{[\![r_i|_i]\!]_t\}_{i=1}^k$ on behalf of `Client`. $\mathcal{S}$ aborts on behalf of `Client` if the shares of corrupted parties are different from those received from the adversary when emulating $\mathcal{F}_{\mathsf{PrepMal}}$. Note that a degree-$t$ Shamir sharing is determined by the shares of honest parties. In **Hybrid**$_{4,4}$, honest parties always use the correct shares. If corrupted

parties use different shares from those sent to $\mathcal{F}_{\mathsf{PrepMal}}$, `Client` will abort. Thus, $\mathcal{S}$ aborts on behalf of `Client` in $\mathbf{Hybrid}_{4,5}$ if and only if `Client` aborts in $\mathbf{Hybrid}_{4,4}$.
Then, $\mathcal{S}$ samples a random vector as $\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{r}$ and generates a random degree-$t$ packed Shamir sharing $[\![\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{r}]\!]_t$. In $\mathbf{Hybrid}_{4,4}$, $\boldsymbol{r}$ are uniformly random. Therefore, $\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{r}$ are also uniformly random. Also $[\![\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{r}]\!]_t$ is a random degree-$t$ packed Shamir sharing of $[\![\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{r}]\!]_t$. Thus, the distribution of $[\![\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{r}]\!]_t$ is identical in both hybrids.
Next, $\mathcal{S}$ samples a random vector as $\boldsymbol{\mu}_{\boldsymbol{\alpha}}$. In $\mathbf{Hybrid}_{4,4}$, $\boldsymbol{\lambda}_{\boldsymbol{\alpha}}$ are uniformly random. Therefore, $\boldsymbol{\mu}_{\boldsymbol{\alpha}} = \boldsymbol{v}_{\boldsymbol{\alpha}} - \boldsymbol{\lambda}_{\boldsymbol{\alpha}}$ are also uniformly random. Thus, the vector $\boldsymbol{\mu}_{\boldsymbol{\alpha}}$ has the same distribution in both hybrids.
Finally, let $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}]\!]_{n-1}$ denote the sharing that `Client` receives from all parties, and $[\![\overline{\boldsymbol{\lambda}_{\boldsymbol{\alpha}}}]\!]_{n-1}$ denote the sharing where the shares of corrupted parties are replaced by those learnt by $\mathcal{S}$ when emulating $\mathcal{F}_{\mathsf{PrepMal}}$. $\mathcal{S}$ computes the shares of corrupted parties of the following sharing

$$[\![\rho(\boldsymbol{\lambda}_{\boldsymbol{\alpha}})]\!]_{n-1} = [\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}]\!]_{n-1} - [\![\overline{\boldsymbol{\lambda}_{\boldsymbol{\alpha}}}]\!]_{n-1}.$$

$\mathcal{S}$ sets the shares of honest parties to be $0$ and reconstructs the secrets $\rho(\boldsymbol{\lambda}_{\boldsymbol{\alpha}})$. Then, we have $\rho(\boldsymbol{\lambda}_{\boldsymbol{\alpha}}) = \widetilde{\boldsymbol{\lambda}_{\boldsymbol{\alpha}}} - \overline{\boldsymbol{\lambda}_{\boldsymbol{\alpha}}}$. On the other hand, recall that $\mathcal{S}$ receives a vector of additive errors $\Delta(\boldsymbol{\lambda}_{\boldsymbol{\alpha}})$ from the adversary when emulating $\mathcal{F}_{\mathsf{PrepMal}}$. We have $\overline{\boldsymbol{\lambda}_{\boldsymbol{\alpha}}} = \boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \Delta(\boldsymbol{\lambda}_{\boldsymbol{\alpha}})$. Thus, $\widetilde{\boldsymbol{\lambda}_{\boldsymbol{\alpha}}} = \boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \Delta(\boldsymbol{\lambda}_{\boldsymbol{\alpha}}) + \rho(\boldsymbol{\lambda}_{\boldsymbol{\alpha}})$. Since `Client` should send $\widetilde{\mu_{\boldsymbol{\alpha}}} = \boldsymbol{v}_{\boldsymbol{\alpha}} - \widetilde{\boldsymbol{\lambda}_{\boldsymbol{\alpha}}}$ to $P_1$, $\mathcal{S}$ sets $\Delta(\boldsymbol{\mu}_{\boldsymbol{\alpha}}) = -\rho(\boldsymbol{\lambda}_{\boldsymbol{\alpha}}) - \Delta(\boldsymbol{\lambda}_{\boldsymbol{\alpha}})$ and sends $\boldsymbol{\mu}_{\boldsymbol{\alpha}} + \Delta(\boldsymbol{\mu}_{\boldsymbol{\alpha}})$ to $P_1$. The distribution of the values sent to $P_1$ is identical in both hybrids.

- When `Client` is corrupted, $\mathcal{S}$ first generates a random vector $\boldsymbol{r}$ and then computes the shares of $\{[\![r_i|i]\!]_t\}_{i=1}^k$ of honest parties based on the shares of corrupted parties. Note that the way of generating $\{[\![r_i|i]\!]_t\}_{i=1}^k$ is identical to that in $\mathcal{F}_{\mathsf{PrepMal}}$ in $\mathbf{Hybrid}_{4,4}$. Thus, the shares of $\{[\![r_i|i]\!]_t\}_{i=1}^k$ of honest parties have the same distribution in both hybrids.
  Then, $\mathcal{S}$ generates a random vector $\boldsymbol{\lambda}_{\boldsymbol{\alpha}}$ and samples a random degree-$(n-1)$ packed Shamir sharing $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \Delta(\boldsymbol{\lambda}_{\boldsymbol{\alpha}})]\!]$, where $\Delta(\boldsymbol{\lambda}_{\boldsymbol{\alpha}})$ is received from the adversary when emulating $\mathcal{F}_{\mathsf{PrepMal}}$. Note that the way of generating $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \Delta(\boldsymbol{\lambda}_{\boldsymbol{\alpha}})]\!]$ is identical to that in $\mathcal{F}_{\mathsf{PrepMal}}$ in $\mathbf{Hybrid}_{4,4}$. Thus, the shares of $[\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \Delta(\boldsymbol{\lambda}_{\boldsymbol{\alpha}})]\!]$ of honest parties have the same distribution in both hybrids.
  Next, $\mathcal{S}$ receives the shares of $[\![\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{r}]\!]_t$ of honest parties and reconstructs $\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{r}$. $\mathcal{S}$ computes $\boldsymbol{v}_{\boldsymbol{\alpha}} = (\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{r}) - \boldsymbol{r}$. In $\mathbf{Hybrid}_{3,4}$, $\boldsymbol{v}_{\boldsymbol{\alpha}}$ are computed from the shares of $\{[\![v_{\alpha_i}|i]\!]_t\}_{i=1}^k$ of honest parties. Recall that $[\![v_{\alpha_i}|i]\!]_t = [\![\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{r}]\!]_t - [\![r_i|i]\!]_t$. Therefore, $v_{\alpha_i} = (v_{\alpha_i} + r_i) - r_i$, which means that $\boldsymbol{v}_{\boldsymbol{\alpha}} = (\boldsymbol{v}_{\boldsymbol{\alpha}} + \boldsymbol{r}) - \boldsymbol{r}$. The inputs extracted by $\mathcal{S}$ have the same distribution as those computed in $\mathbf{Hybrid}_{4,4}$.

In summary, the distribution of $\mathbf{Hybrid}_{4,5}$ is identical to that of $\mathbf{Hybrid}_{4,4}$. Note that $\mathbf{Hybrid}_{4,5}$ is the execution in the ideal world. We have that $\mathbf{Hybrid}_{4,5}$ is statistically close to $\mathbf{Hybrid}_0$, the execution in the real world. Therefore, protocol $\Pi_{\mathsf{Evaluate}}$ securely computes the ideal functionality $\mathcal{F}_{\mathsf{Evaluate}}$ in the $\mathcal{F}_{\mathsf{PrepMal}}$-hybrid model against a fully malicious adversary who controls $t$ corrupted parties and up to c clients.

## D.4 Online Phase — Verification

To check the correctness of the computation, it is sufficient to check whether the adversary launches an additive attack. We describe the functionality $\mathcal{F}_{\mathsf{Verify}}$ for the verification of the computation in Functionality 11.

**Functionality 11:** $\mathcal{F}_{\mathsf{Verify}}$

1. Let $C$ denote the circuit.
   - For each group of input gates with output wires $\boldsymbol{\alpha}$, $\mathcal{F}_{\mathsf{Verify}}$ receives from honest parties their shares of $\{[\![v_{\alpha_i}|i]\!]_t\}_{i=1}^k$. For all $i \in \{1, 2, \ldots, k\}$, $\mathcal{F}_{\mathsf{Verify}}$ recovers the whole sharing $[\![v_{\alpha_i}|i]\!]_t$ and reconstructs the secret $v_{\alpha_i}$. Then $\mathcal{F}_{\mathsf{Verify}}$ sends the shares of $[\![v_{\alpha_i}|i]\!]_t$ of corrupted parties to the adversary.
   - For each group of multiplication gates with input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$, $\mathcal{F}_{\mathsf{Verify}}$ receives from honest parties their shares of $\{[\![v_{\alpha_i}|i]\!]_t, [\![v_{\beta_i}|i]\!]_t\}_{i=1}^k$. For all $i \in \{1, 2, \ldots, k\}$, $\mathcal{F}_{\mathsf{Verify}}$ recovers the whole sharings $[\![v_{\alpha_i}|i]\!]_t, [\![v_{\beta_i}|i]\!]_t$ and reconstructs the secrets $\widetilde{v_{\alpha_i}}, \widetilde{v_{\beta_i}}$. Then $\mathcal{F}_{\mathsf{Verify}}$ sends the shares of $[\![v_{\alpha_i}|i]\!]_t, [\![v_{\beta_i}|i]\!]_t$ of corrupted parties to the adversary.
   - For each group of output gates with input wires $\boldsymbol{\alpha}$, $\mathcal{F}_{\mathsf{Verify}}$ receives from honest parties their shares of $\{[\![v_{\alpha_i}|i]\!]_t\}_{i=1}^k$. For all $i \in \{1, 2, \ldots, k\}$, $\mathcal{F}_{\mathsf{Verify}}$ recovers the whole sharings $[\![v_{\alpha_i}|i]\!]_t$ and

reconstructs the secrets $\widetilde{v_{\alpha_i}}$. Then $\mathcal{F}_{\mathsf{Verify}}$ sends the shares of $[\![v_{\alpha_i}|_i]\!]_t$ of corrupted parties to the adversary.

2. $\mathcal{F}_{\mathsf{Verify}}$ evaluates the circuit $C$ by using the secrets of the degree-$t$ Shamir sharings associated with input gates.
   - For each addition gate with input wires $\alpha, \beta$ and output wire $\gamma$, $\mathcal{F}_{\mathsf{Verify}}$ computes $v_\gamma = v_\alpha + v_\beta$.
   - For each multiplication gate with input wires $\alpha, \beta$ and output wire $\gamma$, $\mathcal{F}_{\mathsf{Verify}}$ computes $\Delta(v_\alpha) = \widetilde{v_\alpha} - v_\alpha$ and $\Delta(v_\beta) = \widetilde{v_\beta} - v_\beta$. Then, $\mathcal{F}_{\mathsf{Verify}}$ sends $\Delta(v_\alpha), \Delta(v_\beta)$ to the adversary. Finally, $\mathcal{F}_{\mathsf{Verify}}$ computes $v_\gamma = \widetilde{v_\alpha} * \widetilde{v_\beta}$.
   - For each output gate with input wire $\alpha$, $\mathcal{F}_{\mathsf{Verify}}$ computes $\Delta(v_\alpha) = \widetilde{v_\alpha} - v_\alpha$. Then, $\mathcal{F}_{\mathsf{Verify}}$ sends $\Delta(v_\alpha)$ to the adversary.
3. $\mathcal{F}_{\mathsf{Verify}}$ checks whether there exists an input wire $\alpha$ of multiplication gates and output gates such that $\Delta(v_\alpha) \neq 0$. If true, $\mathcal{F}_{\mathsf{Verify}}$ sends `abort` to all parties. Otherwise, $\mathcal{F}_{\mathsf{Verify}}$ sends `accept` to all parties.
4. On receiving `abort`, $\mathcal{F}_{\mathsf{Verify}}$ sends `abort` to all parties.

To realize $\mathcal{F}_{\mathsf{Verify}}$, we follow the idea in [BBG+21].

*Verification in [BBG+21].* Recall that our online protocol follows a similar approach to that in [BBG+21]. In particular, in the online protocol in [BBG+21], all parties also only obtain sharings for input wires of multiplication gates but NOT for output wires of multiplication gates. As noted in [BBG+21], for each input wire of multiplication gates and output gates, the wire value should be equal to some linear combination of the inputs of the circuit and the outputs of multiplication gates. Note that the output of each multiplication is equal to the product of its two inputs. Thus, the verification of the computation is transformed to verifying $O(|C|)$ equations, one for each input wire of multiplication gates and output gates. In particular, each equation only contains degree-2 monomials (for the outputs of multiplication gates) and degree-1 monomials (for the inputs of the circuit).

The verification in [BBG+21] is adapted from the techniques in [BBCG+19]. The achieved communication complexity is sub-linear in the circuit size. We observe that we can potentially use a similar approach to that in [BBG+21] to realize $\mathcal{F}_{\mathsf{Verify}}$.

With more details, recall that the work [BBG+21] focuses on the strong honest majority setting, where the number of corrupted parties $t' = (1/2 - \epsilon) \cdot n$. They choose to use a degree-$t$ packed Shamir sharing, where $t = (n-1)/2$, to store $k' = t - t' + 1$ secrets. Note that with $t \leq n/2$, a degree-$t$ packed Shamir sharing can be fully determined by the shares of honest parties, and the multiplication between two degree-$t$ packed Shamir sharings can be done by a natural extension of the DN multiplication protocol [DN07], which works for the single-secret setting. In [BBG+21], the authors note that a degree-$t$ packed Shamir sharing $[\![\boldsymbol{x}]\!]_t$ can be viewed as $k'$ degree-$t$ Shamir sharings $[\![x_1|_1]\!]_t, [\![x_2|_2]\!]_t, \ldots, [\![x_{k'}|_{k'}]\!]_t$. Their verification works on degree-$d$ Shamir sharings, one for each wire value. Recall that we also obtain degree-$t$ Shamir sharings, one for each wire value. Thus, the verification protocol in [BBG+21] can potentially be used in our case.

*Drawbacks of the Verification in [BBG+21].* However, the verification protocol in [BBG+21] does not use the techniques in [BBCG+19] in a black box way. In particular, their protocol has computation complexity $O(|C| \cdot \sqrt{|C|})$ due to the use of the techniques in [BBCG+19] (see an analysis in [BGIN19]), which can be a bottleneck for the concrete efficiency.

Our idea is to use the techniques in [BBCG+19] in a black box way. It allows us to directly use other variants of the techniques in [BBCG+19] in a black box way, for example, the verification protocol in [GS20], which naturally offers a trade-off between the round complexity and the computation complexity. Concretely, for all $d < \sqrt{|C|}$, the verification protocol in [GS20] can achieve $O(|C| \cdot d)$ computation complexity at the cost of $\log_d |C|$ rounds. This trade-off is also explored in the work [BGIN19] for 3-party setting and [BGIN20] for $n$-party setting.

*Step 1: Obtaining a Single Equation.* We label the groups of input gates, multiplication gates, and output gates by $1, 2, \ldots, m$. We have $m \leq |C|/k$. For all $i \in \{1, \ldots, m\}$,

– If the $i$-th group of gates are input gates with output wires $\boldsymbol{\alpha}$, we set

$$([\![v_{i,j}^{(1)}|_j]\!]_t, [\![v_{i,j}^{(2)}|_j]\!]_t) = ([\![v_{\alpha_j}|_j]\!]_t, [\![1|_j]\!]_t).$$

Here the shares of $[\![1|_j]\!]_t$ are all 1.

– If the $i$-th group of gates are multiplication gates with input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$, we set

$$([\![v_{i,j}^{(1)}|_j]\!]_t, [\![v_{i,j}^{(2)}|_j]\!]_t) = ([\![v_{\alpha_j}|_j]\!]_t, [\![v_{\beta_j}|_j]\!]_t).$$

– If the $i$-th group of gates are output gates with input wires $\boldsymbol{\alpha}$, we set

$$([\![v_{i,j}^{(1)}|_j]\!]_t, [\![v_{i,j}^{(2)}|_j]\!]_t) = ([\![v_{\alpha_j}|_j]\!]_t, [\![1|_j]\!]_t).$$

Here the shares of $[\![1|_j]\!]_t$ are all $1$.

Consider the following set

$$T = \left\{ (b,i,j) : \begin{array}{l} [\![v_{i,j}^{(b)}|_j]\!]_t \text{ is associated with some input wire} \\ \text{of multiplication gates and output gates.} \end{array} \right\}$$

Then, the verification of the computation can be represented in the following form: For each $(b,i,j) \in T$, the degree-$t$ Shamir sharing $[\![v_{i,j}^{(b)}|_j]\!]_t$ should satisfy

$$v_{i,j}^{(b)} = \sum_{\ell_1=1}^{|C|/k} \sum_{\ell_2=1}^{k} \theta_{\ell_1,\ell_2}^{(b,i,j)} \cdot (v_{\ell_1,\ell_2}^{(1)} \cdot v_{\ell_1,\ell_2}^{(2)}),$$

where $\{\theta_{\ell_1,\ell_2}^{(b,i,j)}\}_{\ell_1,\ell_2}$ are some known coefficients related to the circuit structure. We transform the verification of these $m$ equations to the verification of a single equation. Recall that $\mathbb{K}$ is an extension field of $\mathbb{F}$ such that $|\mathbb{K}| \geq 2^\kappa$, where $\kappa$ is the security parameter. All parties invoke $\mathcal{F}_{\mathsf{Coin}}$ to generate a random value $r \in \mathbb{K}$. Let $u(b,i,j) = (i-1)\cdot 2k + (j-1)\cdot 2 + b$. Then, all parties multiply $r^{u(b,i,j)-1}$ to the equation for $v_{i,j}^{(b)}$ and sum them up. The final equation is

$$\sum_{(b,i,j)\in T} r^{u(b,i,j)-1} \cdot v_{i,j}^{(b)} = \sum_{\ell_2=1}^{k} \sum_{\ell_1=1}^{|C|/k} \Theta_{\ell_1,\ell_2} \cdot (v_{\ell_1,\ell_2}^{(1)} \cdot v_{\ell_1,\ell_2}^{(2)}) \tag{1}$$

where $\Theta_{\ell_1,\ell_2} = \sum_{(b,i,j)\in T} r^{u(b,i,j)-1} \cdot \theta_{\ell_1,\ell_2}^{(b,i,j)}$.

*Step 2: Performing Inner-Product Operations Via [GS20, BGIN20].* Recall that all parties hold degree-$t$ Shamir sharings $[\![v_{i,j}^{(b)}|_j]\!]_t$ for each $(b,i,j)$. For the LHS of Equation 1, all parties can locally sum up the degree-$t$ Shamir sharings that use the same secret slot. Concretely, for each $\ell_2 \in \{1,2,\ldots,k\}$, all parties compute $[\![x_{\ell_2}|_{\ell_2}]\!]_t = \sum_{(b,i,\ell_2)\in T} r^{u(b,i,\ell_2)-1} \cdot [\![v_{i,\ell_2}^{(b)}|_{\ell_2}]\!]_t$.

For the RHS of Equation 1, we want to compute a degree-$t$ Shamir sharing of the inner-product result $\sum_{\ell_1=1}^{|C|/k} \Theta_{\ell_1,\ell_2} \cdot (v_{\ell_1,\ell_2}^{(1)} \cdot v_{\ell_1,\ell_2}^{(2)})$. We rely on the following two functionalities $\mathcal{F}_{\mathsf{Inner}}$ and $\mathcal{F}_{\mathsf{InnerVerify}}$. The functionality $\mathcal{F}_{\mathsf{Inner}}$ allows all parties efficiently compute the inner-product operation. It can be instantiated by an extension of the DN multiplication protocol (this is a different extension from the one used in [BBG+21]). The communication complexity is $O(n)$ field elements. In particular, the communication complexity is *independent* of the dimension $\ell$. We refer the readers to [GS20] for the description of the protocol that realizes $\mathcal{F}_{\mathsf{Inner}}$ (Protocol 10 in [GS20]).

The functionality $\mathcal{F}_{\mathsf{Inner}}$, however, allows an additive error chosen by the adversary. We need the second functionality $\mathcal{F}_{\mathsf{InnerVerify}}$ to check the correctness of the inner-product result. The second functionality $\mathcal{F}_{\mathsf{InnerVerify}}$ can be realized by techniques in [BBCG+19]. We choose to use the variants presented in [GS20][16] and [BGIN20] which supports offers a trade-off between the round complexity and the computation complexity. Both protocols can achieve $O(\log \ell)$ rounds with communication complexity $O(n^2 \cdot \log \ell \cdot \kappa)$ field elements.

[16] The original protocol in [GS20] is to verify a batch of multiplication triples. However, their first step is to transform a batch of multiplication triples to one inner-product triple. We can simply view the innner-product triple we want to verify as the output of the first step in [GS20].

**Functionality 12: $\mathcal{F}_{\mathsf{Inner}}$**

1. $\mathcal{F}_{\mathsf{Inner}}$ receives the secret position $i$ from all parties. Let $([\![x_1|_i]\!]_t, \ldots, [\![x_\ell|_i]\!]_t)$ and $([\![y_1|_i]\!]_t, \ldots, [\![y_\ell|_i]\!]_t)$ denote the input sharings. For all $j \in \{1, 2, \ldots, \ell\}$, $\mathcal{F}_{\mathsf{Inner}}$ receives from honest parties their shares of $[\![x_j|_i]\!]_t$ and $[\![y_j|_i]\!]_t$. Then $\mathcal{F}_{\mathsf{Inner}}$ reconstructs the secrets $x_j, y_j$. $\mathcal{F}_{\mathsf{Inner}}$ further computes the shares of $[\![x_j|_i]\!]_t, [\![y_j|_i]\!]_t$ held by corrupted parties, and sends these shares to the adversary.
2. $\mathcal{F}_{\mathsf{Inner}}$ receives from the adversary a set of shares $\{z_i\}_{i \in \mathcal{C}orr}$ and an additive error $\eta$.
3. $\mathcal{F}_{\mathsf{Inner}}$ computes $z = \sum_{j=1}^{\ell} x_j \cdot y_j + \eta$. Based on the secret $z$ and the $t$ shares $\{z_i\}_{i \in \mathcal{C}orr}$, $\mathcal{F}_{\mathsf{Inner}}$ reconstructs the whole sharing $[\![z|_i]\!]_t$ and distributes the shares of $[\![z|_i]\!]_t$ to honest parties.

**Functionality 13: $\mathcal{F}_{\mathsf{InnerVerify}}$**

1. $\mathcal{F}_{\mathsf{InnerVerify}}$ receives the secret position $i$ from all parties. The input is denoted by
$$(([\![x_1|_i]\!]_t, \ldots, [\![x_\ell|_i]\!]_t), ([\![y_1|_i]\!]_t, \ldots, [\![y_\ell|_i]\!]_t), [\![z|_i]\!]_t).$$
   - For all $j \in \{1, 2, \ldots, \ell\}$, $\mathcal{F}_{\mathsf{InnerVerify}}$ receives from honest parties their shares of $[\![x_j|_i]\!]_t$ and $[\![y_j|_i]\!]_t$. Then $\mathcal{F}_{\mathsf{InnerVerify}}$ reconstructs the secrets $x_j, y_j$. $\mathcal{F}_{\mathsf{InnerVerify}}$ further computes the shares of $[\![x_j|_i]\!]_t, [\![y_j|_i]\!]_t$ held by corrupted parties, and sends these shares to the adversary.
   - $\mathcal{F}_{\mathsf{InnerVerify}}$ receives from honest parties their shares of $[\![z|_i]\!]_t$. Then $\mathcal{F}_{\mathsf{InnerVerify}}$ reconstructs the secret $z$. $\mathcal{F}_{\mathsf{InnerVerify}}$ further computes the shares of $[\![z|_i]\!]_t$ held by corrupted parties, and sends these shares to the adversary.
   - $\mathcal{F}_{\mathsf{InnerVerify}}$ computes $\eta = z - \sum_{j=1}^{\ell} x_j \cdot y_j$ and sends $\eta$ to the adversary.
2. $\mathcal{F}_{\mathsf{InnerVerify}}$ checks whether $\eta \neq 0$. If true, $\mathcal{F}_{\mathsf{InnerVerify}}$ sends `abort` to all parties. Otherwise, $\mathcal{F}_{\mathsf{InnerVerify}}$ sends `accept` to all parties.
3. On receiving `abort`, $\mathcal{F}_{\mathsf{InnerVerify}}$ sends `abort` to all parties.

By using $\mathcal{F}_{\mathsf{Inner}}$ and $\mathcal{F}_{\mathsf{InnerVerify}}$, all parties can compute a degree-$t$ Shamir sharing $[\![y_{\ell_2}|_{\ell_2}]\!]_t$ such that
$$y_{\ell_2} = \sum_{\ell_1=1}^{|C|/k} \Theta_{\ell_1,\ell_2} \cdot (v^{(1)}_{\ell_1,\ell_2} \cdot v^{(2)}_{\ell_1,\ell_2}).$$

*Step 3: Checking Summation of Sharings.* After step 2, all parties hold $\{[\![x_i|_i]\!]_t, [\![y_i|_i]\!]_t\}_{i=1}^{k}$. In particular, $\sum_{i=1}^{k} x_i$ is equal to the LHS of Equation 1 and $\sum_{i=1}^{k} y_i$ is equal to the RHS of Equation 1.

Let $[\![z_i|_i]\!]_t = [\![x_i|_i]\!]_t - [\![y_i|_i]\!]_t$ for all $i \in \{1, 2, \ldots, k\}$. The problem is reduce to checking whether $\sum_{i=1}^{k} z_i = 0$.

To this end, each party $P_j$ prepares and distributes $k$ random degree-$t$ Shamir sharings $\{[\![o_i^{(j)}|_i]\!]_t\}_{i=1}^{k}$ such that the summation of the secrets $\sum_{i=1}^{k} o_i^{(j)} = 0$. Then, for all $i \in \{1, \ldots, k\}$, all parties locally compute $[\![o_i|_i]\!]_t = \sum_{j=1}^{n} [\![o_i^{(j)}|_i]\!]_t$. We will use $\{[\![o_i|_i]\!]_t\}_{i=1}^{k}$ as random masks.

All parties invoke $\mathcal{F}_{\mathsf{Coin}}$ to generate a random value $r' \in \mathbb{K}$. Then for all $i \in \{1, \ldots, k\}$, all parties compute $r' \cdot [\![z_i|_i]\!]_t + [\![o_i|_i]\!]_t$ and reconstruct secret to every party. Each party checks whether $\sum_{i=1}^{k} (r' \cdot z_i + o_i) = 0$.

*Summary of Our Verification Protocol.* We describe our verification protocol in $\Pi_{\mathsf{Verify}}$. The communication complexity of $\Pi_{\mathsf{Verify}}$ is sub-linear in the circuit size. Therefore, it does not affect the concrete efficiency. Also, by using techniques in [GS20, BGIN20] to instantiate $\mathcal{F}_{\mathsf{InnerVerify}}$, we estimate that the computation complexity of the verification protocol will not become the bottleneck of the running time.

**Protocol 14: $\Pi_{\mathsf{Verify}}$**

1. **Defining Inputs**: Let $C$ denote the circuit. All groups of input gates, multiplication gates, and output gates are labeled by $1, 2, \ldots, m$. We have $m \leq |C|/k$. For all $i \in \{1, \ldots, m\}$,
   - If the $i$-th group of gates are input gates with output wires $\boldsymbol{\alpha}$, all parties hold $\{[\![v_{\alpha_i}|_i]\!]_t\}_{i=1}^{k}$. All parties set
$$([\![v^{(1)}_{i,j}|_j]\!]_t, [\![v^{(2)}_{i,j}|_j]\!]_t) = ([\![v_{\alpha_j}|_j]\!]_t, [\![1|_j]\!]_t).$$
   Here the shares of $[\![1|_j]\!]_t$ are all 1.

- If the $i$-th group of gates are multiplication gates with input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$, all parties hold $\{[\![v_{\alpha_i}|_i]\!]_t, [\![v_{\beta_i}|_i]\!]_t\}_{i=1}^k$. All parties set

$$([\![v_{i,j}^{(1)}|_j]\!]_t, [\![v_{i,j}^{(2)}|_j]\!]_t) = ([\![v_{\alpha_j}|_j]\!]_t, [\![v_{\beta_j}|_j]\!]_t).$$

- If the $i$-th group of gates are output gates with input wires $\boldsymbol{\alpha}$, all parties hold $\{[\![v_{\alpha_i}|_i]\!]_t\}_{i=1}^k$. All parties set

$$([\![v_{i,j}^{(1)}|_j]\!]_t, [\![v_{i,j}^{(2)}|_j]\!]_t) = ([\![v_{\alpha_j}|_j]\!]_t, [\![1|_j]\!]_t).$$

Here the shares of $[\![1|_j]\!]_t$ are all 1.

2. **Step 1 — Obtaining a Single Equation**: Let $T$ be the set defined as below:

$$T = \left\{ (b,i,j) : \begin{array}{l} [\![v_{i,j}^{(b)}|_j]\!]_t \text{ is associated with some input wire} \\ \text{of multiplication gates and output gates.} \end{array} \right\}$$

For all $(b,i,j) \in T$, let $\{\theta_{\ell_1,\ell_2}^{(b,i,j)}\}_{\ell_1,\ell_2}$ be the coefficients such that

$$v_{i,j}^{(b)} = \sum_{\ell_1=1}^{|C|/k} \sum_{\ell_2=1}^{k} \theta_{\ell_1,\ell_2}^{(b,i,j)} \cdot (v_{\ell_1,\ell_2}^{(1)} \cdot v_{\ell_1,\ell_2}^{(2)}).$$

All parties invoke $\mathcal{F}_{\mathsf{Coin}}$ and generates a random value $r \in \mathbb{K}$. Let $u(b,i,j) = (i-1)\cdot 2k + (j-1)\cdot 2 + b$. Then for all $\ell_1 \in \{1,\ldots,m\}, \ell_2 \in \{1,\ldots,k\}$, all parties locally compute

$$\Theta_{\ell_1,\ell_2} = \sum_{(b,i,j)\in T} r^{u(b,i,j)-1} \cdot \theta_{\ell_1,\ell_2}^{(b,i,j)}.$$

All parties will verify

$$\sum_{(b,i,j)\in T} r^{u(b,i,j)-1} \cdot v_{i,j}^{(b)} = \sum_{\ell_2=1}^{k} \sum_{\ell_1=1}^{|C|/k} \Theta_{\ell_1,\ell_2} \cdot (v_{\ell_1,\ell_2}^{(1)} \cdot v_{\ell_1,\ell_2}^{(2)}) \quad (1)$$

3. **Step 2 — Performing Inner-Product Operations**:
   (a) For all $\ell_2 \in \{1,2,\ldots,k\}$, all parties locally compute $[\![x_{\ell_2}|_{\ell_2}]\!]_t = \sum_{(b,i,\ell_2)\in T} r^{u(b,i,\ell_2)-1} \cdot [\![v_{i,\ell_2}^{(b)}|_{\ell_2}]\!]_t$. Then, the LHS of Equation 1 is equal to $\sum_{i=1}^k x_i$.
   (b) For all $\ell_2 \in \{1,2,\ldots,k\}$,
      i. All parties first locally compute $[\![\Theta_{\ell_1,\ell_2} \cdot v_{\ell_1,\ell_2}^{(1)}|_{\ell_2}]\!]_t = \Theta_{\ell_1,\ell_2} \cdot [\![v_{\ell_1,\ell_2}^{(1)}|_{\ell_2}]\!]_t$ for all $\ell_1 \in \{1,\ldots,m\}$.
      ii. All parties invoke $\mathcal{F}_{\mathsf{Inner}}$ with inputs $([\![\Theta_{\ell_1,\ell_2} \cdot v_{\ell_1,\ell_2}^{(1)}|_{\ell_2}]\!]_t)_{\ell_1}$ and $([\![v_{\ell_1,\ell_2}^{(2)}|_{\ell_2}]\!]_t)_{\ell_1}$ and output $[\![y_{\ell_2}|_{\ell_2}]\!]_t$.
      iii. All parties invoke $\mathcal{F}_{\mathsf{InnerVerify}}$ with inputs $([\![\Theta_{\ell_1,\ell_2} \cdot v_{\ell_1,\ell_2}^{(1)}|_{\ell_2}]\!]_t)_{\ell_1}$, $([\![v_{\ell_1,\ell_2}^{(2)}|_{\ell_2}]\!]_t)_{\ell_1}$, and $[\![y_{\ell_2}|_{\ell_2}]\!]_t$. If $\mathcal{F}_{\mathsf{InnerVerify}}$ outputs `accept`, all parties continue. Otherwise, all parties abort.

   After this step, all parties will verify $\sum_{i=1}^k x_i = \sum_{i=1}^k y_i$.
4. **Step 3 — Checking Summation of Sharings**:
   (a) For all $j \in \{1,2,\ldots,n\}$, $P_j$ randomly generates $\{[\![o_i^{(j)}|_i]\!]_t\}_{i=1}^k$ such that $\sum_{i=1}^k o_i^{(j)} = 0$ in the extension field $\mathbb{K}$. Then $P_j$ distributes $\{[\![o_i^{(j)}|_i]\!]_t\}_{i=1}^k$ to other parties. Next for all $i \in \{1,\ldots,k\}$, all parties locally compute $[\![o_i|_i]\!]_t = \sum_{j=1}^n [\![o_i^{(j)}|_i]\!]_t$.
   (b) All parties invoke $\mathcal{F}_{\mathsf{Coin}}$ to generates a random value $r' \in \mathbb{K}$. For all $i \in \{1,\ldots,k\}$, all parties compute $[\![z_i|_i]\!]_t = r' \cdot ([\![x_i|_i]\!]_t - [\![y_i|_i]\!]_t) + [\![o_i|_i]\!]_t$.
   (c) All parties send their shares of $\{[\![z_i|_i]\!]_t\}_{i=1}^k$ to every party $P_j$. Then each party $P_j$ checks that:
      - For all $i \in \{1,\ldots,k\}$, the shares of $[\![z_i|_i]\!]_t$ lie on a degree-$t$ polynomial.
      - The summation $z_1 + \ldots + z_k = 0$.

      $P_j$ accepts the verification if both checks pass. Otherwise, $P_j$ aborts.

**Lemma 8.** *Protocol $\Pi_{\mathsf{Verify}}$ securely computes $\mathcal{F}_{\mathsf{Verify}}$ in the $\{\mathcal{F}_{\mathsf{Coin}}$, $\mathcal{F}_{\mathsf{Inner}}, \mathcal{F}_{\mathsf{InnerVerify}}\}$-hybrid model against a fully malicious adversary who controls $t$ parties.*

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

1. In Step 1, for each degree-$t$ Shamir sharing, $\mathcal{S}$ receives from $\mathcal{F}_{\mathsf{Verify}}$ the shares of corrupted parties.

2. In Step 2, by the definition of $T$, each $(b,i,j) \in T$ satisfies that $[\![v_{i,j}^{(b)}|_j]\!]_t$ is associated with some input wire of multiplication gates and output gates. For each $(b,i,j) \in T$, $\mathcal{S}$ receives from $\mathcal{F}_{\mathsf{Verify}}$ the additive error $\Delta(v_{i,j}^{(b)})$. Let $\widetilde{v_{i,j}^{(b)}}$ denote the secret of $[\![v_{i,j}^{(b)}|_j]\!]_t$. Then the real secret is $v_{i,j}^{(b)} = \widetilde{v_{i,j}^{(b)}} - \Delta(v_{i,j}^{(b)})$.
   Recall that the functionality $\mathcal{F}_{\mathsf{Verify}}$ computes each $v_{i,j}^{(b)}$ by using the wire values with additive errors in previous layers, i.e., $\{\widetilde{v_{\ell_1,\ell_2}^{(0)}}, \widetilde{v_{\ell_1,\ell_2}^{(1)}}\}_{\ell_1<i}$. Therefore,
$$\Delta(v_{i,j}^{(b)}) = \widetilde{v_{i,j}^{(b)}} - \sum_{\ell_1=1}^{|C|/k} \sum_{\ell_2=1}^{k} \theta_{\ell_1,\ell_2}^{(b,i,j)} \cdot (\widetilde{v_{\ell_1,\ell_2}^{(1)}} \cdot \widetilde{v_{\ell_1,\ell_2}^{(2)}}).$$
   $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{Coin}}$ and randomly samples $r \in \mathbb{K}$. Then $\mathcal{S}$ computes $\Delta_0 = \sum_{(b,i,j)\in T} r^{u(b,i,j)-1} \cdot \Delta(v_{i,j}^{(b)})$.
3. In Step 3, $\mathcal{S}$ computes the shares of $[\![x_{\ell_2}|_{\ell_2}]\!]_t$ of corrupted parties for all $\ell_2 \in \{1,\ldots,k\}$.
   For all $\ell_2 \in \{1,\ldots,k\}$, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{Inner}}$ and $\mathcal{F}_{\mathsf{InnerVerify}}$ by using the shares of $\{[\![v_{i,j}^{(b)}|_j]\!]_t\}_{(b,i,j)}$ of corrupted parties. $\mathcal{S}$ receives the shares of $[\![y_{\ell_2}|_{\ell_2}]\!]_t$ of corrupted parties. Concretely,
   – For $\mathcal{F}_{\mathsf{Inner}}$, $\mathcal{S}$ uses the shares of $\{[\![v_{i,j}^{(b)}|_j]\!]_t\}_{(b,i,j)}$ of corrupted parties to computes the values that should be sent to the adversary. Then $\mathcal{S}$ receives the shares of $[\![y_{\ell_2}|_{\ell_2}]\!]_t$ of corrupted parties and the additive error $\eta$.
   – For $\mathcal{F}_{\mathsf{InnerVerify}}$, $\mathcal{S}$ uses the shares of $\{[\![v_{i,j}^{(b)}|_j]\!]_t\}_{(b,i,j)}$ and $[\![y_{\ell_2}|_{\ell_2}]\!]_t$ of corrupted parties and the additive error $\eta$ to computes the values that should be sent to the adversary.

   If all parties abort, $\mathcal{S}$ sends `abort` to $\mathcal{F}_{\mathsf{Verify}}$.
4. In Step 4, for each honest party $P_j$, $\mathcal{S}$ sends random values to the adversary as the shares of $\{[\![o_i^{(j)}|_i]\!]_t\}_{i=1}^k$, and $\mathcal{S}$ sets $\Delta_j = 0$. For each corrupted party $P_j$, $\mathcal{S}$ receives the shares of $\{[\![o_i^{(j)}|_i]\!]_t\}_{i=1}^k$ of honest parties from the adversary. Then, $\mathcal{S}$ recovers the whole sharings and reconstructs the secrets $\{o_i^{(j)}\}_{i=1}^k$. $\mathcal{S}$ sets $\Delta_j = \sum_{i=1}^k o_i^{(j)}$.
   $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{Coin}}$ and randomly samples $r' \in \mathbb{K}$. Then $\mathcal{S}$ computes the shares of $[\![z_i|_i]\!]_t$ of corrupted parties for all $i \in \{1,\ldots,k\}$. $\mathcal{S}$ computes $\Delta = r' \cdot \Delta_0 + \sum_{j=1}^n \Delta_j$ and randomly samples $z_1,\ldots,z_k$ such that $\sum_{i=1}^k z_i = \Delta$. For all $i \in \{1,\ldots,k\}$, based on the secret $z_i$ and the shares of corrupted parties, $\mathcal{S}$ computes the shares of $[\![z_i|_i]\!]_t$ of honest parties.
   $\mathcal{S}$ follows the rest of this step honestly.
   – If some honest party aborts, $\mathcal{S}$ sends `abort` to $\mathcal{F}_{\mathsf{Verify}}$.
   – If all honest parties accept, but there exists $(b,i,j) \in T$ such that $\Delta(v_{i,j}^{(b)}) \neq 0$, $\mathcal{S}$ sends `abort` to $\mathcal{F}_{\mathsf{Verify}}$.

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

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

**Hybrid**$_0$: In this hybrid, $\mathcal{S}$ honestly follows the protocol.

**Hybrid**$_1$: In this hybrid,

– In Step 1, $\mathcal{S}$ computes the shares of $\{[\![v_{i,j}^{(b)}|_j]\!]_t\}_{(b,i,j)}$ of corrupted parties from the shares of honest parties. $\mathcal{S}$ also computes the secrets $\{\widetilde{v_{i,j}^{(b)}}\}_{(b,i,j)}$.
– In Step 2, for each $(b,i,j) \in T$, $\mathcal{S}$ computes
$$\Delta(v_{i,j}^{(b)}) = \widetilde{v_{i,j}^{(b)}} - \sum_{\ell_1=1}^{|C|/k} \sum_{\ell_2=1}^{k} \theta_{\ell_1,\ell_2}^{(b,i,j)} \cdot (\widetilde{v_{\ell_1,\ell_2}^{(1)}} \cdot \widetilde{v_{\ell_1,\ell_2}^{(2)}}).$$
  Then, $\mathcal{S}$ computes $\Delta_0 = \sum_{(b,i,j)\in T} r^{u(b,i,j)-1} \cdot \Delta(v_{i,j}^{(b)})$.
– In Step 4, for each corrupted party $P_j$, $\mathcal{S}$ recovers the whole sharings $\{[\![o_i^{(j)}|_i]\!]_t\}_{i=1}^k$ by using the shares of honest parties and reconstructs the secrets $\{o_i^{(j)}\}_{i=1}^k$. Then $\mathcal{S}$ computes $\Delta_j = \sum_{i=1}^k o_i^{(j)}$. For each honest party $P_j$, $\mathcal{S}$ sets $\Delta_j = 0$.

Note that $\mathcal{S}$ does not change the behaviors of honest parties. The distribution of **Hybrid**$_1$ is identical to that of **Hybrid**$_0$.

**Hybrid**$_2$: In this hybrid, $\mathcal{S}$ prepares the shares of $\{\llbracket z_i|_i\rrbracket_t\}_{i=1}^k$ of honest parties as described above.

In **Hybrid**$_1$, all parties compute $\llbracket z_i|_i\rrbracket_t = r' \cdot (\llbracket x_i|_i\rrbracket_t - \llbracket y_i|_i\rrbracket_t) + \llbracket o_i|_i\rrbracket_t$. We first show that $\sum_{i=1}^k z_i = \Delta$. Recall that $\sum_{i=1}^k x_i = \sum_{(b,i,j)\in T} r^{u(b,i,j)-1} \cdot \widetilde{v^{(b)}_{i,j}}$ and $\sum_{i=1}^k y_i = \sum_{\ell_2=1}^k \sum_{\ell_1=1}^{|C|/k} \Theta_{\ell_1,\ell_2} \cdot (\widetilde{v^{(1)}_{\ell_1,\ell_2}} \cdot \widetilde{v^{(2)}_{\ell_1,\ell_2}})$. We have $\sum_{i=1}^k (x_i - y_i) = \Delta_0$. Also recall that $\sum_{i=1}^k o_i = \sum_{j=1}^n \sum_{i=1}^k o_i^{(j)} = \sum_{j=1}^n \Delta_j$. Therefore, $\sum_{i=1}^k z_i = r' \cdot \Delta_0 + \sum_{j=1}^n \Delta_j = \Delta$.

We then show that $z_1, \ldots, z_k$ are random values subject to $\sum_{i=1}^k z_i = \Delta$. Without loss of generality, suppose $P_1$ is honest. Then $o_1^{(1)}, \ldots, o_k^{(1)}$ are random values subject to $\sum_{i=1}^k o_i^{(1)} = 0$. Since $z_i = r' \cdot (x_i - y_i) + \sum_{j=1}^n o_i^{(j)}$, $z_1, \ldots, z_k$ are random values subject to $\sum_{i=1}^k z_i = \Delta$.

In **Hybrid**$_2$, $\mathcal{S}$ randomly samples $z_1, \ldots, z_k$ subject to $\sum_{i=1}^k z_i = \Delta$. Therefore, the distribution of $z_1, \ldots, z_k$ is identical in both hybrids. Since a degree-$t$ Shamir sharing is determined by the secret and the shares of corrupted parties, the shares of $\{\llbracket z_i|_i\rrbracket_t\}_{i=1}^k$ of honest parties are identically distributed in both hybrids.

Thus, **Hybrid**$_2$ is identically distributed to **Hybrid**$_1$.

**Hybrid**$_3$: In this hybrid, in Step 4, if all honest parties accept, but there exists $(b,i,j) \in T$ such that $\Delta(v_{i,j}^{(b)}) \neq 0$, $\mathcal{S}$ sends `abort` to $\mathcal{F}_{\mathsf{Verify}}$ and aborts on behalf of honest parties. We claim that the probability that all honest parties accept but there exists $(b,i,j) \in T$ such that $\Delta(v_{i,j}^{(b)}) \neq 0$ is negligible.

Suppose there exists $(b,i,j) \in T$ such that $\Delta(v_{i,j}^{(b)}) \neq 0$. We first show that, with overwhelming probability, $\Delta_0$ is non-zero. Recall that $\Delta_0 = \sum_{(b,i,j)\in T} r^{u(b,i,j)-1} \cdot \Delta(v_{i,j}^{(b)})$. This can be viewed as a polynomial in $r$ and the degree is bounded by $2mk = O(|C|)$. With the same argument as that in Lemma 6, the number of $r$ such that $\Delta_0 = 0$ is bounded by $O(|C|)$. Since the field size of $\mathbb{K}$ is $2^\kappa$, with overwhelming probability, $\Delta_0 \neq 0$.

Then we show that, with overwhelming probability $\Delta \neq 0$. Recall that $\Delta = r' \cdot \Delta_0 + \sum_{j=1}^n \Delta_j$. With the same argument as above, the number of $r'$ such that $\Delta = 0$ is at most 1. Therefore, with overwhelming probability, $\Delta \neq 0$.

Thus, if there exists $(b,i,j) \in T$ such that $\Delta(v_{i,j}^{(b)}) \neq 0$, with overwhelming probability, $\Delta \neq 0$. Note that all parties accept only if $\Delta = 0$. Therefore, **Hybrid**$_3$ is statistically close to **Hybrid**$_2$.

**Hybrid**$_4$: In this hybrid,

- In Step 1, $\mathcal{S}$ uses the shares of $\{\llbracket v_{i,j}^{(b)}|_j\rrbracket_t\}_{(b,i,j)}$ of corrupted parties received from $\mathcal{F}_{\mathsf{Verify}}$.
- In Step 2, for each $(b,i,j) \in T$, $\mathcal{S}$ uses $\Delta(v_{i,j}^{(b)})$ received from $\mathcal{F}_{\mathsf{Verify}}$.
- In Step 3, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{Inner}}$ and $\mathcal{F}_{\mathsf{InnerVerify}}$ by using the shares of $\{\llbracket v_{i,j}^{(b)}|_j\rrbracket_t\}_{(b,i,j)}$ of corrupted parties.
- In Step 4, for each honest party $P_j$, $\mathcal{S}$ samples random values as the shares of $\{\llbracket o_i^{(j)}|_i\rrbracket_t\}_{i=1}^k$ of honest parties.

For Step 1 and Step 2, note that these values are computed by $\mathcal{F}_{\mathsf{Verify}}$ in the same way as that in **Hybrid**$_3$. For Step 3, only the shares of corrupted parties are needed to emulate $\mathcal{F}_{\mathsf{Inner}}$ and $\mathcal{F}_{\mathsf{InnerVerify}}$. For Step 4, for each honest party $P_j$, the shares of $\{\llbracket o_i^{(j)}|_i\rrbracket_t\}_{i=1}^k$ of honest parties are uniformly random in **Hybrid**$_3$.

Therefore, **Hybrid**$_4$ is identically distributed to **Hybrid**$_3$. Note that **Hybrid**$_4$ is the execution in the ideal world. We have that **Hybrid**$_4$ is statistically close to **Hybrid**$_0$, the execution in the real world. Therefore, protocol $\Pi_{\mathsf{Verify}}$ securely computes the ideal functionality $\mathcal{F}_{\mathsf{Verify}}$ in the $\{\mathcal{F}_{\mathsf{Coin}}, \mathcal{F}_{\mathsf{Inner}}, \mathcal{F}_{\mathsf{InnerVerify}}\}$-hybrid model against a fully malicious adversary who controls $t$ corrupted parties and up to c clients.

*Computing Coefficients for Step 2.* Recall that in Step 2, all parties need to compute a coefficient $\Theta_{\ell_1,\ell_2}$ for each input gate and multiplication gate. Computing all coefficients $\{\Theta_{\ell_1,\ell_2}\}_{\ell_1,\ell_2}$ directly can occur $O(|C|^2)$ computation complexity: Recall that $\Theta_{\ell_1,\ell_2} = \sum_{(b,i,j)\in T} r^{u(b,i,j)-1} \cdot \theta_{\ell_1,\ell_2}^{(b,i,j)}$. In the worst case each $\Theta_{\ell_1,\ell_2}$ is a summation of $O(|C|)$ terms.

In this part, we give an algorithm which can compute all coefficients $\{\Theta_{\ell_1,\ell_2}\}_{\ell_1,\ell_2}$ with computation complexity $O(|C|)$. Our idea is to assign a value to each wire $\alpha$, denoted by `weight`$(\alpha)$, as the weight of wire $\alpha$. We will maintain the invariant that the weighted sum of all wire values is equal to $\sum_{(b,i,j)\in T} r^{u(b,i,j)-1} \cdot v_{i,j}^{(b)}$, the LHS of Equation 1. Initially, the weights are non-zero only for the input wires of multiplication gates and output gates. The algorithm will gradually change the weight of wires while maintaining the invariant so that finally the weights are non-zero only for the output wires of input gates and multiplication gates. Then the weights associated with the output wires of input gates and multiplication gates are the coefficients we need.

1. **Initialization**: In the beginning, we set $\texttt{weight}(\alpha) = 0$ for all wire $\alpha$. For each input wire $\alpha$ of multiplication gates and output gates, suppose $v_{i,j}^{(b)}$ is the wire value of $\alpha$. We set $\texttt{weight}(\alpha) = \texttt{weight}(\alpha) + r^{u(b,i,j)-1}$. (Note that the output wire of a gate may be used as an input wire for multiple gates.)
   After the initialization, the weighted sum of all wire values is equal to $\sum_{(b,i,j)\in T} r^{u(b,i,j)-1} \cdot v_{i,j}^{(b)}$.
2. **Transformation**: We change the weight of wires layer by layer. We start from the last layer (except the output layer).
   – For each addition gate in the current layer, suppose the input wires are $\alpha, \beta$ and the output wire is $\gamma$. We set

$$\texttt{weight}(\alpha) = \texttt{weight}(\alpha) + \texttt{weight}(\gamma)$$
$$\texttt{weight}(\beta) = \texttt{weight}(\beta) + \texttt{weight}(\gamma).$$

   Then we set $\texttt{weight}(\gamma) = 0$. Note that the weighted sum of all wire values remains unchanged. It follows from the fact that $v_\alpha + v_\beta = v_\gamma$.
   – For each multiplication gate in the current layer, we do nothing. Note that the weight associated with the output wire of this gate is the coefficient of this multiplication gate.

   After modifying the weights of all addition gates in the current layer, all parties move to the previous layer. The algorithm terminates when reaching the input layer.

Note that after the above process, only weights associated with the output wires of input gates and multiplication gates are non-zero. The weighted sum of all wire values is equal to

$$\sum_{\ell_2=1}^{k} \sum_{\ell_1=1}^{|C|/k} \Theta_{\ell_1,\ell_2} \cdot (v_{\ell_1,\ell_2}^{(1)} \cdot v_{\ell_1,\ell_2}^{(2)}),$$

the RHS of Equation 1. One can verify that the coefficient of the output wire of an input gate or a multiplication gate is the one we want to compute.

Regarding the computation complexity, note that we only visit each gate once in the above process. Therefore, the computation complexity is $O(|C|)$.

### D.5 Summary: Main Protocol with Malicious Security

Now we are ready to present the main protocol $\Pi_{\mathsf{MainMal}}$ with malicious security. It is simply a combination of $\mathcal{F}_{\mathsf{Evaluate}}$ and $\mathcal{F}_{\mathsf{Verify}}$. The ideal functionality $\mathcal{F}_{\mathsf{MainMal}}$ appears in Functionality 14. The security of $\Pi_{\mathsf{MainMal}}$ follows from $\mathcal{F}_{\mathsf{Evaluate}}$ and $\mathcal{F}_{\mathsf{Verify}}$.

**Functionality 14: $\mathcal{F}_{\mathsf{MainMal}}$**

1. $\mathcal{F}_{\mathsf{MainMal}}$ receives the input from all clients. Let $x$ denote the input and $C$ denote the circuit.
2. $\mathcal{F}_{\mathsf{MainMal}}$ computes $C(x)$. $\mathcal{F}_{\mathsf{MainMal}}$ first distributes the output of corrupted clients to the adversary.
   – If the adversary replies `continue`, $\mathcal{F}_{\mathsf{MainMal}}$ distributes the output to all clients.
   – If the adversary replies `abort`, $\mathcal{F}_{\mathsf{MainMal}}$ sends `abort` to all clients.

**Protocol 15: $\Pi_{\mathsf{MainMal}}$**

1. All parties and clients invoke $\mathcal{F}_{\mathsf{Evaluate}}$ to compute a degree-$t$ Shamir sharing for each output wire of input gates, and for each input wire of multiplication gates and output gates.
2. All parties invoke $\mathcal{F}_{\mathsf{Verify}}$ to check the correctness of the computation.
3. For each output gate that belongs to some `Client`, all parties hold a degree-$t$ Shamir sharing $[\![v_{\alpha_i}|i]\!]_t$ that is associated with this gate.
   (a) All parties send their shares of $[\![v_{\alpha_i}|i]\!]_t$ to `Client`.

(b) `Client` checks whether the shares of $[\![v_{\alpha_i}|i]\!]_t$ lie on a degree-$t$ polynomial. If true, `Client` reconstructs the secret $v_{\alpha_i}$ and takes it as the output of this gate. Otherwise, `Client` aborts.

When we combine our protocols $\Pi_{\mathsf{PrepIndMal}}$, $\Pi_{\mathsf{PrepMal}}$, $\Pi_{\mathsf{Evaluate}}$, $\Pi_{\mathsf{Verify}}$, $\Pi_{\mathsf{MainMal}}$, and instantiate the functionality $\mathcal{F}_{\mathsf{SingleMultMal}}$ by [GLO+21], the functionalities $\mathcal{F}_{\mathsf{Coin}}$, $\mathcal{F}_{\mathsf{Inner}}$, $\mathcal{F}_{\mathsf{InnerVerify}}$ by [GS20], we obtain an information-theoretic MPC protocol in the client-server model with splitting communication complexity as follows:

- In the circuit-independent preprocessing phase, all parties need to communicate $10n+24$ elements per gate.
- In the circuit-dependent preprocessing phase, all parties need to communicate $8$ elements per gate.
- In the online phase, all parties need to communicate $12$ elements per gate.

Observe that this is identical to our optimized semi-honest protocol presented in Section C. We have the following theorem.

**Theorem 3.** *In the client-server model, let* $\mathsf{c}$ *denote the number of clients,* $n$ *denote the number of parties (servers), and* $t = (n-1)/2$ *denote the number of corrupted parties (servers). Let* $\mathbb{F}$ *be a finite field of size* $|\mathbb{F}| \geq 2n$*. For an arithmetic circuit* $C$ *over* $\mathbb{F}$*, there exists an information-theoretic MPC protocol which securely computes the arithmetic circuit* $C$ *(with abort) in the presence of a fully malicious adversary controlling up to* $\mathsf{c}$ *clients and* $t$ *parties. The splitting communication complexity per gate is (1)* $10n+24$ *elements per gate in the circuit-independent preprocessing phase, (2)* $8$ *elements per gate in the circuit-dependent preprocessing phase, and (3)* $12$ *elements per gate in the online phase. (Terms that are independent of or sub-linear in the circuit size are omitted as they only add cost* $o(1)$ *per gate.)*

## E DN07 with Circuit-Dependent Preprocessing

In this section we describe the circuit-dependent preprocessing variant of DN07 that we use for a fair comparison. The standard DN07 protocol [DN07] achieves a total communication complexity of $6n$ field elements per multiplication gate, distributed as $4n$ elements in an offline phase (which is circuit-independent), and $2n$ in an online phase. ATLAS improves the total communication to $4n$ elements, but the online phase still consists of $2n$ elements. In [GSZ20], the online phase of the original DN07 is improved from $2n$ to $1.5n$ elements, while keeping the offline phase to be $4n$ elements, but unfortunately this technique is not compatible with the approach from ATLAS. Since our protocol optimizes the online phase, it is more reasonable to compare against the existing protocol with the most efficient online phsae, so we do not consider ATLAS for our comparison.

Our main observation here is that DN07, with the optimization from [GSZ20], can be tweaked to achieve an online phase of $1n$ elements per multiplication gate, by moving some of the messages from the online phase to a *circuit-dependent* offline phase. This way, the total communication of $5.5n$ elements is distributed as $4n$ elements in the circuit-independent offline phase, $0.5n$ elements in the circuit-dependent offline phase, and $1n$ elements in the online phase. Another interesting property of the resulting protocol is that the last $t$ parties can go offline after the circuit-dependent phase, which may be an important feature in some cases as it can help saving in server costs, and it reduces communication channels. However, we remark that this is only possible for passive security (which is the case we are concerned with here since this protocol is designed solely for experimentally comparing against TurboPack, which we implemented in the semi-honest setting). For active security the last $t$ parties must return for a final verification stage, but we do not discuss how such protocol would work.

*Comparison with [DE21b].* The protocol from [DE21b] also achieves an online phase in the circuit-dependent preprocessing model that involves $1n$ field elements per multiplication gate. However, for our comparison we decided to use the optimized version of DN07 we present in this section since as we now show its total communication complexity is better.

The protocol from [DE21b, Section 4] also allows the online phase to be executed among the first $t+1$ parties only, and it can be seen as an execution among the first $t+1$ parties of the dishonest majority MPC protocol Turbospeedz [BENO19], where the necessary preprocessing is generated by all $n = 2t + 1$ parties. In the passive version, the preprocessing consists of additive triples among the first $t + 1$ parties of the form $(\langle\lambda_\alpha\rangle, \langle\lambda_\beta\rangle, \langle\lambda_\alpha \cdot \lambda_\beta\rangle)$, and then the online phase consists of opening the sharing $\langle\mu_\gamma\rangle$ which is done by sending shares to $P_1$, who reconstructs and send the result back. This takes $2t = n - 1 \leq 1n$ field elements.

The preprocessing requires the *circuit-dependent* triples described above, which are generated by sampling shares of uniformly random values first, and then using an existing multiplication protocol (*e.g.* ATLAS [GLO+21]) to scurely compute the product. This results in a communication for the preprocessing phase of $4n$, coming from ATLAS, plus the costs of generating two random sharings per multiplication gate. This results in a total communication complexity that is larger than the optimized DN07 protocol we propose here, while having the same online complexity.

### E.1 Plain DN07

The original DN07 protocol [DN07] operates as follows. In the (circuit-independent) offline phase, one so-called double share per multiplication is generated, which is a pair of sharings $(\llbracket r\rrbracket_t, \llbracket r\rrbracket_{2t})$ where $r \in \mathbb{F}$ is uniformly random and unknown to the adversary. Then, in the online phase, given $\llbracket x\rrbracket_t$ and $\llbracket y\rrbracket_t$ the parties obtain $\llbracket x \cdot y\rrbracket_t$ by computing locally $\llbracket d\rrbracket_{2t} = \llbracket x\rrbracket_t \cdot \llbracket y\rrbracket_t - \llbracket r\rrbracket_{2t}$, reconstructing $d$, and then again computing locally $\llbracket r\rrbracket_t + d = \llbracket x \cdot y\rrbracket_t$. The reconstruction of $d$ is done by all parties first sending their shares of $\llbracket d\rrbracket_{2t}$ to $P_1$, involving $1n$ field elements,[17] followed by $P_1$ reconstructing $d$ and sending it back to the parties, adding $1n$ more elements. The instantiation of the offline phase in [DN07]—which is the best known that is compatible with this online phase and with the optimization from [GSZ20] described below—requires $4n$ elements.

### E.2 Optimization from [GSZ20]

In [GSZ20], the online phase from above is optimized from $2n$ to $1.5n$ elements as follows. The main observation is that we can regard the step when $P_1$ sends $d$ to the parties as $P_1$ distributing shares of degree $0$ of $d$. However, since these "shares" will be added to $\llbracket r\rrbracket_t$, the parties could afford to receive degree-$t$ shares instead. It turns out that this enables $P_1$ to send less messages, given that degree-$t$ sharings of $d$ can be obtained by setting, say, the share of the last $t$ parties to be $0$, which then, together with the "secret" $d$, determine a polynomial of degree $t$ and hence determine the remaining shares. If $P_1$ computes the shares of $d$ in this way, $P_1$ does not need to communicate with the last $t$ parties, who know their share is $0$, and $P_1$ only needs to communicate to the remaining $t$ parties, which corresponds to $t \leq n/2$ messages. This leads to an online phase that uses $1n + 0.5n = 1.5n$ field elements per multiplication gate.

### E.3 Online Phase with $1n$ Elements using Circuit-Dependent Preprocessing

When allowing for circuit-dependent preprocessing, it turns out that $0.5n$ elements out of the $1.5n$ elements in the online phase from the previous optimized protocol can be moved to a circuit-dependent offline phase. This is achieved as follows. The main observation is that the shares of $\llbracket d\rrbracket_{2t} = \llbracket x\rrbracket_t \cdot \llbracket y\rrbracket_t - \llbracket r\rrbracket_{2t}$ that $P_1$ receives from the last $t$ parties are determined already in the offline phase if the circuit is known, and therefore these can be sent to $P_1$ in a circuit-dependent offline phase. This removes $t = (n - 1)/2$ messages from the online phase, and pushes them into the circuit-dependent offline phase.

To see why this observation holds, we first make a more general claim, which is that the shares of the last $t$ parties corresponding to each wire are already determined in the preprocessing phase (assuming the circuit is known). This suffices for our claim above regarding the messages $P_1$ receives in a multiplication, since these are derived from the shares of each wire held by the last $t$ parties. Let us first describe the protocol by which the clients provide inputs. For each input gate, the parties have shares $\llbracket s\rrbracket_t$ where $s \in \mathbb{F}$ is uniformly random, and the client knows the secret $s$. The client sends $x - s$ to the parties, who locally compute $\llbracket x\rrbracket_t = (x - s) + \llbracket s\rrbracket_t$. As with the multiplication protocol, this can be improved so that the client sends sharings $\llbracket x - s\rrbracket_t$ instead, where the last $t$ shares have been set to be zero. This implies that the shares of the input $\llbracket x\rrbracket_t$ corresponding to the last $t$ parties are equal to their shares of $\llbracket s\rrbracket_t$, which are known from the preprocessing phase. A similar observation holds for the multiplication gates: the resulting shares of the product are given

[17] For simplicity we do count as communication the case when a party sends a message to him/herself. This has little effect in the final complexity.

by $[\![x \cdot y]\!] = [\![d]\!]_t + [\![r]\!]_t$, where the last $t$ shares of $[\![d]\!]_t$ are zero, so the shares of the product of the last $t$ parties are given by the last $t$ shares of $[\![r]\!]_t$, which are known at preprocessing time.

To summarize, the optimization where the online phase consists of $1n$ field elements consists of the last $t$ parties sending their shares of $[\![x]\!]_t \cdot [\![y]\!]_t - [\![r]\!]_{2t}$ in a circuit-dependent offline phase, which is possible since as we argued the last $t$ shares of each wire value (and in particular, these of $[\![x]\!]_t$ and $[\![y]\!]_t$) are already determined in the offline phase. This way, in the online phase $P_1$ only needs to hear from and talk to the first $n - t = t + 1$ parties. Also, notice that this optimization also allows us to run the online phase only among $t + 1$ parties, or in other words, the last $t$ parties can be shut down after they have sent their necessary shares out.[18] This can be a useful feature.

## F More Experimental Results

| Width | Prep. | Number of parties | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 5 | | 13 | | 21 | | 29 | | 37 | |
| | | TP (s) | Factor (×) | TP (s) | Factor (×) | TP (s) | Factor (×) | TP (s) | Factor (×) | TP (s) | Factor (×) |
| 100 | CD | 0.36 / 1.14 | **2.87 / 0.98** | 0.53 / 0.86 | **1.87 / 0.78** | 0.88 / 1.16 | **1.82 / 1.01** | 1.36 / 1.30 | **1.98 / 1.10** | 2.06 / 1.25 | **2.25 / 0.91** |
| | CI | 0.21 / 1.29 | **1.85 / 1.10** | 0.40 / 0.98 | **1.50 / 0.88** | 0.73 / 1.32 | **1.60 / 1.12** | 1.19 / 1.47 | **1.82 / 1.21** | 1.80 / 1.51 | **2.02 / 1.08** |
| 1k | CD | 1.54 / 1.07 | **7.00 / 0.96** | 3.00 / 1.30 | **5.99 / 0.97** | 4.49 / 1.23 | **4.68 / 0.65** | 7.17 / 1.58 | **4.38 / 0.56** | 12.54 / 1.86 | **5.12 / 0.39** |
| | CI | 1.30 / 1.30 | **7.51 / 1.13** | 2.52 / 1.78 | **5.89 / 1.25** | 3.87 / 1.84 | **4.32 / 0.94** | 5.82 / 2.94 | **3.79 / 1.00** | 10.43 / 3.97 | **4.45 / 0.82** |
| 10k | CD | 7.31 / 1.68 | **6.93 / 0.94** | 16.57 / 2.51 | **7.01 / 0.71** | 32.69 / 4.25 | **5.69 / 0.43** | 65.89 / 6.50 | **6.21 / 0.30** | 117.53 / 10.59 | **6.58 / 0.26** |
| | CI | 6.24 / 2.74 | **7.02 / 1.41** | 14.25 / 4.82 | **7.03 / 1.25** | 27.37 / 9.58 | **5.33 / 0.92** | 55.24 / 17.15 | **5.52 / 0.78** | 100.36 / 27.75 | **5.87 / 0.67** |
| 100k | CD | 54.34 / 5.42 | **4.90 / 0.52** | 153.74 / 13.95 | **7.44 / 0.42** | 334.39 / 31.18 | **6.52 / 0.34** | 666.64 / 59.18 | **6.63 / 0.29** | 1167.04 / 97.12 | **6.93 / 0.25** |
| | CI | 47.38 / 12.39 | **5.87 / 0.91** | 131.19 / 36.50 | **7.23 / 1.03** | 282.69 / 82.89 | **5.99 / 0.85** | 564.94 / 160.88 | **5.95 / 0.76** | 997.72 / 266.44 | **6.18 / 0.66** |

Table 3: Running times and comparison of TURBOPACK with DN07, in a setting with 10ms latency and 100Mbps bandwidth, for a circuit of depth 10 and varying width and number of parties. The TP columns refer to the running time of TURBOPACK in seconds. The "factor" columns refer to the ratio between the running time of TURBOPACK and DN07. The format of the timings and ratios is "Offline / Online". In the CD. Prep case our offline and online phases are ①+② and ③, while in the CI. Prep scenario these are ① and ②+③. The entries with N/A correspond to cases where the parties crashed, so we could not obtain the corresponding data.

| Width | Prep. | Number of parties | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 5 | | 13 | | 21 | | 29 | | 37 | |
| | | TP (s) | Factor (×) | TP (s) | Factor (×) | TP (s) | Factor (×) | TP (s) | Factor (×) | TP (s) | Factor (×) |
| 100 | CD | 2.41 / 6.28 | **2.00 / 0.89** | 4.23 / 6.21 | **1.61 / 0.90** | 5.87 / 6.61 | **1.38 / 0.98** | 7.50 / 6.40 | **1.27 / 0.93** | 9.45 / 6.57 | **1.25 / 0.95** |
| | CI | 1.81 / 6.88 | **1.64 / 0.96** | 3.24 / 7.20 | **1.29 / 1.02** | 4.93 / 7.56 | **1.19 / 1.10** | 6.67 / 7.23 | **1.16 / 1.03** | 8.69 / 7.33 | **1.17 / 1.04** |
| 1k | CD | 12.25 / 7.33 | **6.08 / 1.09** | 17.33 / 11.26 | **3.31 / 1.21** | 27.21 / 8.89 | **3.17 / 0.78** | 40.65 / 6.50 | **5.62 / 0.51** | 49.03 / 7.87 | **4.84 / 0.45** |
| | CI | 10.44 / 9.15 | **6.12 / 1.30** | 13.83 / 14.77 | **3.35 / 1.42** | 21.65 / 14.44 | **2.76 / 1.20** | 36.87 / 10.28 | **5.59 / 0.77** | 46.57 / 10.34 | **4.83 / 0.58** |
| 10k | CD | 66.30 / 12.73 | **6.35 / 0.79** | 116.77 / 16.86 | **4.62 / 0.74** | 186.19 / 17.75 | **5.16 / 0.47** | N/A | **N/A** | N/A | **N/A** |
| | CI | 59.84 / 19.18 | **7.57 / 1.03** | 109.72 / 23.91 | **5.54 / 0.84** | 174.21 / 29.73 | **5.68 / 0.69** | N/A | **N/A** | N/A | **N/A** |
| 100k | CD | 474.15 / 43.98 | **4.37 / 0.66** | 1067.8 / 83.15 | **6.03 / 0.63** | 1680.7 / 83.13 | **5.82 / 0.32** | N/A | **N/A** | N/A | **N/A** |
| | CI | 446.07 / 72.05 | **5.44 / 0.78** | 1011 / 139.92 | **5.82 / 1.03** | 1628.6 / 135.17 | **5.76 / 0.50** | N/A | **N/A** | N/A | **N/A** |

Table 4: Running times and comparison of TURBOPACK with DN07, in a setting with 100ms latency and 100Mbps bandwidth, for a circuit of depth 10 and varying width and number of parties. The TP columns refer to the running time of TURBOPACK in seconds. The "factor" columns refer to the ratio between the running time of TURBOPACK and DN07. The format of the timings and ratios is "Offline / Online". In the CD. Prep case our offline and online phases are ①+② and ③, while in the CI. Prep scenario these are ① and ②+③.

As a complement to Section 5, we also include experimental results that measure the performance of TURBOPACK relative to DN07, in other networking settings. In Section 5 we considered a distributed setting with 1ms latency and 60Mbps bandwidth, which mimics a LAN scenario. Here we include results for other two important settings. First, in Table 3 we present results for the same

[18] Regarding output gates, the last $t$ parties can send their shares (which are determined in the circuit-dependent offline phase already) to the corresponding output clients.

| Width | Prep. | Number of parties | | | | | |
|---|---|---|---|---|---|---|---|
| | | 37 | | 45 | | 53 | |
| | | **TP (s)** | **Factor (×)** | **TP (s)** | **Factor (×)** | **TP (s)** | **Factor (×)** |
| 1k | **CD** | 1.71 / 0.15 | **6.80 / 0.30** | 2.51 / 0.16 | **7.22 / 0.21** | 3.67 / 0.23 | **6.17 / 0.19** |
| | **CI** | 1.48 / 0.38 | **6.07 / 0.74** | 2.19 / 0.48 | **6.46 / 0.60** | 3.22 / 0.68 | **5.75 / 0.55** |
| 10k | **CD** | 13.24 / 1.10 | **7.07 / 0.25** | 21.56 / 1.61 | **6.90 / 0.22** | 34.38 / 2.26 | **6.83 / 0.19** |
| | **CI** | 11.39 / 2.94 | **6.31 / 0.67** | 18.80 / 4.37 | **6.18 / 0.59** | 30.52 / 6.12 | **6.19 / 0.52** |
| 100k | **CD** | 123.76 / 10.76 | **6.52 / 0.25** | 201.23 / 15.98 | **7.10 / 0.22** | 321.38 / 22.38 | **6.48 / 0.19** |
| | **CI** | 105.57 / 28.95 | **5.76 / 0.66** | 174.11 / 43.09 | **6.33 / 0.59** | 283.22 / 60.54 | **5.82 / 0.52** |

| Width | Prep. | Number of parties | | | | | |
|---|---|---|---|---|---|---|---|
| | | 61 | | 69 | | 77 | |
| | | **TP (s)** | **Factor (×)** | **TP (s)** | **Factor (×)** | **TP (s)** | **Factor (×)** |
| 1k | **CD** | 5.72 / 0.31 | **6.29 / 0.17** | 7.35 / 0.40 | **6.62 / 0.16** | 10.17 / 0.50 | **7.44 / 0.14** |
| | **CI** | 5.13 / 0.91 | **5.71 / 0.50** | 6.60 / 1.16 | **6.02 / 0.45** | 9.23 / 1.44 | **6.83 / 0.41** |
| 10k | **CD** | 49.97 / 3.05 | **7.02 / 0.17** | 69.28 / 3.96 | **7.52 / 0.16** | 92.78 / 5.01 | **7.92 / 0.15** |
| | **CI** | 44.76 / 8.26 | **6.39 / 0.47** | 62.26 / 10.98 | **6.86 / 0.43** | 84.00 / 13.78 | **7.26 / 0.40** |
| 100k | **CD** | 491.70 / 30.50 | **7.17 / 0.18** | 681.65 / 39.67 | **7.54 / 0.16** | 932.10 / 50.29 | **7.91 / 0.15** |
| | **CI** | 440.25 / 81.94 | **6.52 / 0.47** | 614.41 / 106.91 | **6.89 / 0.42** | 846.57 / 135.82 | **7.29 / 0.39** |

Table 5: Running times and comparison of TurboPack with DN07, in a `localhost` setting (*i.e.* without altering the network), for a circuit of depth 1 and varying width and number of parties. The TP columns refer to the running time of TurboPack in seconds. The "factor" columns refer to the ratio between the running time of TurboPack and DN07. The format of the timings and ratios is "Offline / Online". In the CD. Prep case our offline and online phases are ①+② and ③, while in the CI. Prep scenario these are ① and ②+③.

range of parties as in Table 1, but increasing the latency from 1ms to 10ms This is intended to emulate a distributed system over small distances. Then, in Table 5 we aim at exploring the effect of having an even larger number of parties, for which we modify the networking setting by not applying any limitation to the `localhost` network, and we also lower the depth from 10 to 1. This is because of technical difficulties we encountered when running a large amount of parties in a single machine with modifications to the network using `tc`.

*Setting with 10ms latency* The experiments here are run in the same setup as in Section 5. The experiments for $n = 5, 13, 21, 29$ are the average of five runs, while for $n = 37, 45$ they are the average of two runs. Comparing the results from Tables 1 (1ms latency, depth 10) and 3 (10ms latency, depth 10), we see that the improvement factor of our online phase with respect to that of DN07 remains essentially the same. However, an interesting observation is that the ratio for the offline phase seems to improve as the latency is increased, and this is particularly more noticeable for small widths. In these cases, computation matters more, and this behavior seems to support the idea that, when the latency is larger, more time can be spent while messages are in transit in the extra computations involved in TurboPack which are, for example, packing/unpacking secret-shared elements, or mapping pre-processed data to different parts of the circuit.

*Setting with `localhost` communication* Interprocess communication with TCP has much more bandwidth and less latency than an actual distributed setting such as the ones we have emulated so far for the previous experiments. However, in order to consider an even larger number of parties, we found that we had to remove network emulation. We benchmark, in this setting, a circuit of depth 1 with different widths and a much larger number of parties which ranges over the set $\{29, 37, 45, 53, 61\}$. The results are presented in Table 5. We see that, as expected, the improvement of TurboPack grows noticeably as the number of parties increases, and for $n = 61$, TurboPack shows an improvement of $10\times$ with respect to DN07, even for circuits of small width. We remark that technical difficulties prevented us from carrying out these experiments, but we expect that further considering more realistic networks with constrained bandwidth and latency would lead to even better improvement factors.