# SuperPack: Dishonest Majority MPC with Constant Online Communication

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

[1] J.P. Morgan AI Research & J.P. Morgan AlgoCRYPT CoE, NY, USA
{daniel.escudero,antigoni.polychroniadou}@jpmorgan.com
[2] NTT Research, CA, USA
[3] Carnegie Mellon University, PA, USA vipul@cmu.edu
[4] Tsinghua University, Beijing, China yfsong1995@gmail.com
[5] Northwestern University, IL, USA ckweng@u.northwestern.edu

**Abstract.** In this work we present a novel actively secure dishonest majority MPC protocol, SuperPack, whose efficiency improves as the number of *honest* parties increases. Concretely, let $0 < \epsilon < 1/2$ and consider an adversary that corrupts $t < n(1-\epsilon)$ out of $n$ parties. SuperPack requires $6/\epsilon$ field elements of online communication per multiplication gate across all parties, assuming circuit-dependent preprocessing, and $10/\epsilon$ assuming circuit-independent preprocessing. In contrast, most of previous works such as SPDZ (Damgård *et al*, ESORICS 2013) and its derivatives perform the same regardless of whether there is only one honest party, or a constant (non-majority) fraction of honest parties. The only exception is due to Goyal *et al* (CRYPTO 2022), which achieves $58/\epsilon + 96/\epsilon^2$ field elements assuming circuit-independent preprocessing. Our work improves this result substantially by a factor of at least 25 in the circuit-independent preprocessing model.

Practically, we also compare our work with the best concretely efficient online protocol Turbospeedz (Ben-Efraim *et al*, ACNS 2019), which achieves $2(1-\epsilon)n$ field elements per multiplication gate among all parties. Our online protocol improves over Turbospeedz as $n$ grows, and as $\epsilon$ approaches $1/2$. For example, if there are 90% corruptions ($\epsilon = 0.1$), with $n = 50$ our online protocol is $1.5\times$ better than Turbospeedz and with $n = 100$ this factor is $3\times$, but for 70% corruptions ($\epsilon = 0.3$) with $n = 50$ our online protocol is $3.5\times$ better, and for $n = 100$ this factor is $7\times$.

Our circuit-dependent preprocessing can be instantiated from OLE/VOLE. The amount of OLE/VOLE correlations required in our work is a factor of $\approx \epsilon n/2$ smaller than these required by Le Mans (Rachuri and Scholl, CRYPTO 2022) leveraged to instantiate the preprocesing of Turbospeedz.

Our dishonest majority protocol relies on packed secret-sharing and leverages ideas from the honest majority TurboPack (Escudero *et al*, CCS 2022) protocol to achieve concrete efficiency for any circuit topology, not only SIMD. We implement both SuperPack and Turbospeedz and verify with experimental results that our approach indeed leads to more competitive runtimes in distributed environments with a moderately large number of parties.

## 1 Introduction

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

Communication complexity is a key measure of efficiency for MPC. Over the last few decades, great progress has been made in the design of communication-efficient honest majority protocols [4,13,18,24,9,23,7,20,15]. In particular, the recent work [15] shows that it is possible to achieve *constant* communication complexity among all parties (i.e., $O(1)$) per multiplication gate in the online phase while maintaining *linear* communication complexity in the number of parties (i.e., $O(n)$) per multiplication gate in the offline phase — which is independent of the private inputs.

Dishonest majority protocols provide the best security guarantees in terms of collusion sizes since security will be ensured even if all parties but one jointly collude against the remaining honest party. It is known that in this setting public key cryptography tools are needed. In the seminar work of Beaver [1] it was shown how to push most of the "heavy crypto machinery" to an offline phase, hence allowing for a more efficient online phase that can even be information-theoretically secure, or at least use simpler cryptographic tools such as PRGs and hash functions for efficiency. This approach eventually led to the seminal works of BeDOZa [5] and SPDZ [12,14], which leveraged the Beaver triple technique from [1] together with message authentication codes to achieve a concretely efficient online phase with linear linear communication complexity in the number of parties per gate. The online phase in SPDZ has been very influential, and there is a large body of research that has focused solely on improving the offline phase, leaving the SPDZ online phase almost intact.

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

[6] An example is [10] which achieves slightly sub-linear communication complexity in the *circuit size* at the cost of increasing the preprocessed data size to be quadratic in the circuit size.

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

To be concretely efficient, we refer to protocols that do not rely on heavy Cryptographic tools such as FHE. In particular, we restrict the online phase to be almost information-theoretic except the black-box use of PRGs or hash functions. Perhaps surprisingly, it is not clear what benefits can be achieved if assume instead of all-but-one corruption, but a constant fraction of parties are honest. In fact, in the case that there are $n - t > 1$ honest parties — unless these constitute a majority — the best one can do to optimize communication is removing $(n - t - 1)$ parties so that, in the new set, there is at least *one* honest party, which is the only requirement for dishonest majority protocols to guarantee security. To the best of our knowledge, the only exception to this is [22], which considers the corruption threshold $t = n(1 - \epsilon)$ for a constant $\epsilon$ in the circuit-independent preprocessing model and achieves $58/\epsilon + 96/\epsilon^2$ elements per multiplication gate among all parties in the malicious security setting[7]. Despite the constant communication complexity per multiplication gate achieved in [22], it requires hundreds or even thousands of parties to outperform SPDZ [14].

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

### 1.1 Our Contribution

In this work, we answer the above question affirmatively: we design the first concretely efficient dishonest majority MPC protocol SuperPack that achieves constant online communication among all parties per multiplication gate with comparable efficiency as the state-of-the-art in the honest majority setting [15]. SuperPack tolerates any number of corruptions and becomes more efficient as the number of honest parties increases, or put differently, it becomes more efficient as the percentage of corrupted parties decreases.

More concretely, we show the following theorem.

**Theorem 1 (Informal).** *Let $n$ be a positive integer, $\epsilon \in (0, 1/2)$ be a constant, and $\kappa$ be the security parameter. For an arithmetic circuit $C$ that computes an $n$-ary functionality $\mathcal{F}$, there exists an $n$-party protocol that computes $C$ with computational security against a fully malicious adversary who can control at most $t = n(1 - \epsilon)$ corrupted parties. The protocol has total communication*

[7] The work [22] does not analyze the concrete cost of their malicious protocol. We obtain this number by counting the amount of communication in their construction. We note that the protocol in [22] also needs to interact for addition gates. Our reported number assumes that the amount of addition gates is the same as the amount of multiplication gates.

*$O(6|C|n + 45|C|/\epsilon)$ elements (ignoring the terms that are independent of the circuit size or only related to the circuit depth[8]) with splitting cost:*

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

Our construction has the following features:

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

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

**Circuit-independent offline phase (Section 6).** The circuit-independent preprocessing is instantiated by a vector oblivious linear evaluation (VOLE) functionality and an oblivious linear evaluation (OLE) functionality. These two functionalities are realized by protocols in Le Mans [25], which can achieve sub-linear communication complexity in the amount of preprocessed data. In addition, we manage to reduce the amount of preprocessed data by a factor of $\epsilon n/2$ compared with that in [25]. More discussion can be found in Section 2.

*Comparison to Best Previous Works.* When comparing with [22], which achieves $58/\epsilon + 96/\epsilon^2$ elements per multiplication gate among all parties in the circuit-independent preprocessing phase, our protocol achieves a factor of at least 25 improvement in the same setting, and a factor of at least 40 improvement in the circuit-dependent preprocessing phase. Since [22] does not realize the circuit-independent preprocessing phase, we do not compare the cost in the circuit-independent preprocessing phase.

Since our goal is to optimize the online phase of dishonest majority protocols where there is a constant fraction of honest parties, we take as a baseline for comparison the existing dishonest majority protocol with the best concrete efficiency in the online phase. This corresponds to the Turbospeedz protocol [3],

[8] The only term that is related to the circuit depth is in the form of $O(n \cdot \texttt{Depth})$. This is because of the use of packed secret sharing which requires to evaluate at least $O(n)$ gates per layer. A similar term also occurs in previous works that use packed secret sharings [11,17,2,21,22,15].

which is set in the circuit-dependent preprocessing model. To instantiate the preprocessing, we utilize the state-of-the-art [25]. Details on this protocol are given in the full version of this paper. The resulting protocol has the following communication complexity: $2(1-\epsilon)n$ in the online phase, $4(1-\epsilon)n$ in the circuit-dependent offline phase, and $6(1-\epsilon)n$ in the circuit-independent offline phase when instantiated using Le Mans [25] (ignoring the calls to the VOLE and OLE functionalities). Again, the VOLE and OLE functionalities can be properly instantiated with sub-linear communication complexity in the preprocessed data size. And our protocol even reduce this size by a factor of $\epsilon n/2$.

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

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

Regarding the complete offline phase (ignoring calls to $\mathcal{F}_{\mathsf{OLE}}^{\mathrm{prog}}$ and $\mathcal{F}_{\mathsf{nVOLE}}$), our complexity is $6n + 39/\epsilon$, while in Turbospeedz it is $10(1-\epsilon)n$. In the limit as $n \to \infty$, our offline protocol is approximately a factor of $10(1-\epsilon)/6$ times better than Turbospeedz/Le Mans, which ranges between $10/6 \approx 1.6$ for $\epsilon = 0$, to $5/6 \approx 0.83$ for $\epsilon = 1/2$. As a result, *in the limit*, our offline phase is only $1/0.83 = 1.2\times$ less efficient than that of Turbospeedz (and for $\epsilon$ close to zero it can be even up to 1.6 better), which is a reasonable cost taking into account the benefits in the online phase. A more thorough discussion on the communication complexity and its implications is given in the full version of this paper.

*Implementation and experimental results.* Finally, we implement all of our protocol—except for the OLE/VOLE functionalities—and verify that, experimentally, our protocol outperforms Turbospeedz by the expected amount based on the communication measures when the runtimes are not computation bound. For example, in a 100 mbps network our online phase is more than $\approx 4.5\times$ better than that of Turbospeedz for 80 parties, where 60% of them are corrupted. If

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

**Table 1.** Communication complexity in terms of field elements per multiplication gate of SuperPack, and comparison to the previous work with the best concrete efficiency in the online phase, which is Turbospeedz [3] (with its offline phase instantiated by Le Mans [25]), referred to as Turbospeedz$^*$. The cost of the calls to $\mathcal{F}_{\mathsf{OLE}}^{\mathrm{prog}}$ and $\mathcal{F}_{\mathsf{nVOLE}}$ in the circuit-independent offline phase is ignored.

the network is too fast, then computation becomes a more noticeable bottleneck, and our improvements are less noticeable. This is discussed in Section 7.

## 2 Overview of the Techniques

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

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

1. First, we lift the online protocol of TurboPack [15] from the honest majority setting to the dishonest majority setting. Our starting point is the observation that the passive version of the online protocol from TurboPack [15] also works for a dishonest majority by setting the parameters correctly. To achieve malicious security, however, the original techniques do not work. This is because in TurboPack, all parties will prepare a degree-$t$ Shamir sharing for each wire value in the circuit. In the honest majority setting, a degree-$t$ Shamir sharing satisfies that the shares of honest parties can fully determine the secret, and the most that malicious parties can do is to change their local shares and cause the whole sharing inconsistent (in the sense that the shares do not lie on a degree-$t$ polynomial). Malicious parties however cannot change the secret by changing their shares. This property unfortunately does not hold in the dishonest majority setting.

Instead, in our case, we rely on a different type of redundancy widely used in the dishonest majority setting: We make use of message authentication codes, or MACs, to ensure that corrupted parties cannot change the secrets by changing their local shares without being caught. While a similar technique has also been used in [22], their way of using MACs increases the online communication complexity by a factor of at least 2 compared with their passive protocol.
We will show how to use MACs in a way such that the online communication complexity remains the same as our passive protocol.

2. Second, we have to reinvent the circuit-independent preprocessing protocol for SuperPack as the corresponding protocol from TurboPack highly relies on the assumption of honest majority, plus that we also need the preprocessed sharings to be authenticated due to the larger corruption threshold.
The main preprocessing data we need to prepare is referred to as *Packed Beaver Triples*, which are first introduced in [22]. At a high level, a packed Beaver triple contains three packed Shamir sharings $([\boldsymbol{a}], [\boldsymbol{b}], [\boldsymbol{c}])$ such that $\boldsymbol{a}, \boldsymbol{b}$ are random vectors in $\mathbb{F}^k$ and $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$. To prepare such a packed Beaver triple, a direct approach would be first preparing standard Beaver triples using additive sharings and then transform them to packed Shamir sharings. In this way, we may reuse the previous work of generating standard Beaver triples in a black box way. However, this idea requires us to not only pay the cost of preparing standard Beaver triples, but also pay the cost of doing the sharing transformation. The direct consequence is that the overall efficiency of our protocol will be *worse* than that of the state-of-the-art [25] in the dishonest majority setting. (And this is the approach used in TurboPack [15].)
We will show how to take the advantage of the constant fraction of honest parties in the circuit-independent preprocessing phase by carefully using the techniques of [25] in our setting.

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

### 2.1 Starting Point: TurboPack

Our starting point is the observation that the passive version of the online protocol from TurboPack [15], which is set in the *honest majority* setting, also works for a dishonest majority by setting the parameters correctly. We focus mostly on multiplication gates. So we ignore details regarding input and output gates.

**Preprocessing.** We consider an arithmetic circuit whose wires are indexed by certain identifiers, which we denote using lowercase Greek letters $\alpha$, $\beta$, $\gamma$, etc. Our work is set in the client-server model where there are input and output gates associated to *clients*, who will be in charge of providing input/receiving output. Each multiplication layer of the circuit is split into *batches* of size $k$. Similarly, each input and output layer assigned to a given client are split into batches of

size $k$. The invariant in TurboPack is the following. First, every wire $\alpha$ that is not the output of an addition gate has associated to it a uniformly random value $\lambda_\alpha$. If a wire $\gamma$ is the output of an addition gate with input wires $\alpha$ with wire $\beta$, then $\lambda_\gamma$ is defined (recursively) as $\lambda_\alpha + \lambda_\beta$.

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

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

*Maintaining the Invariant for Multiplication Gates.* Consider a group of multiplication gates in a given circuit level, having input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$, and output wires $\boldsymbol{\gamma}$. Assume that the invariant holds for the input wires, meaning that $P_1$ knows $\boldsymbol{\mu_\alpha} = \boldsymbol{v_\alpha} - \boldsymbol{\lambda_\alpha}$ and $\boldsymbol{\mu_\beta} = \boldsymbol{v_\beta} - \boldsymbol{\lambda_\beta}$. Recall that the parties have the preprocessed sharings $[\boldsymbol{\lambda_\alpha}]_{n-k}$, $[\boldsymbol{\lambda_\beta}]_{n-k}$, and $[\boldsymbol{\lambda_\gamma}]_{n-1}$. To maintain the invariant, $P_1$ must learn $\boldsymbol{\mu_\gamma} = \boldsymbol{v_\gamma} - \boldsymbol{\lambda_\gamma}$, where $\boldsymbol{v_\gamma} = \boldsymbol{v_\alpha} * \boldsymbol{v_\beta}$. This is achieved by using the techniques of packed Beaver triples introduced in [22]. Recall that all parties also hold a fresh packed Beaver triple $([\boldsymbol{a}]_{n-k}, [\boldsymbol{b}]_{n-k}, [\boldsymbol{c}]_{n-1})$. All parties proceeds as follows:

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

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

and $[\boldsymbol{\mu_\gamma}]_{n-1} = [\boldsymbol{v_\gamma}]_{n-1} - [\boldsymbol{\lambda_\gamma}]_{n-1}$.

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

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

### 2.2 Achieving Active Security

There are multiple places where an active adversary can cheat in the previous protocol, with the most obvious being distributing incorrect (or even invalid) $[\boldsymbol{v_\alpha} - \boldsymbol{a}]_{k-1}$ and $[\boldsymbol{v_\beta} - \boldsymbol{b}]_{k-1}$ at a group of multiplication gates, either by corrupting $P_1$, or by sending incorrect shares in previous gates to $P_1$. This is prevented in TurboPack by explicitly making use of the honest majority assumption: Using the degree-$(k-1)$ packed Shamir sharings distributed by $P_1$, the parties will be able to obtain a certain "individual" (*i.e.* non-packed) degree-$t$ Shamir sharing for each wire value. As we discussed above, a degree-$t$ Shamir sharing in the honest majority setting allows honest parties to fully determine the secret. This enables the use of distributed zero-knowledge techniques [6] to check the correctness of the computation.

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

- With MACs, corrupted parties cannot change the secrets of a degree-$(n-k)$ packed Shamir sharing without being detected except with a negligible probability.
- In addition to adding verifiability to packed Shamir sharings, we show how to allow all parties to directly compute MACs of the secret values that are shared by $P_1$ using degree-$(k-1)$ packed Shamir sharings. This allows us to directly verify whether $\boldsymbol{v_\alpha} - \boldsymbol{a}$ and $\boldsymbol{v_\beta} - \boldsymbol{b}$ are correct without doing distributed zero-knowledge like [15].

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

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

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

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

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

The parties preserve the invariant as follows.

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

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

  1. For $[\boldsymbol{v}_{\boldsymbol{\alpha}} - \boldsymbol{a}]_{k-1} * [\boldsymbol{b}]_{n-k}$ and $[\boldsymbol{v}_{\boldsymbol{\beta}} - \boldsymbol{b}]_{k-1} * [\boldsymbol{a}]_{n-k}$, since all parties also hold $[\Delta \cdot \boldsymbol{a}]_{n-k}$ and $[\Delta \cdot \boldsymbol{b}]_{n-k}$, they may locally compute $[\boldsymbol{v}_{\boldsymbol{\alpha}} - \boldsymbol{a}]_{k-1} * [\Delta \cdot \boldsymbol{b}]_{n-k}$ and $[\boldsymbol{v}_{\boldsymbol{\beta}} - \boldsymbol{b}]_{k-1} * [\Delta \cdot \boldsymbol{a}]_{n-k}$ and convert them locally to $\langle \Delta \cdot (v_{\alpha_i} - a_i) \cdot b_i \rangle$ and $\langle \Delta \cdot (v_{\beta_i} - b_i) \cdot a_i \rangle$.
  2. For $[\boldsymbol{c}]_{n-1}$ and $[\boldsymbol{\lambda}_{\boldsymbol{\gamma}}]_{n-1}$, all parties already hold $\langle \Delta \cdot c_i \rangle$ and $\langle \Delta \cdot \lambda_{\gamma_i} \rangle$.
  3. The problematic part is to obtain $\langle \Delta \cdot (v_{\alpha_i} - a_i) \cdot (v_{\beta_i} - b_i) \rangle$. There we use the degree-$t$ Shamir sharing $[\Delta|_i]_t$ as follows. We note that

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

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

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

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

It is sufficient to check the following two points:

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

Both of these two checks are natural extensions of the checks done in SPDZ [14]. We thus omit the details and refer the readers to Section 4.4 for more details.

### 2.3 Instantiating the Circuit-Dependent Preprocessing

The preprocessing required by the parties is summarized as follows.

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

For the circuit-independent part, we will focus more on the preparation of the packed Beaver triples with authentications in the next section since the size of $[\Delta|_1]_t, \ldots, [\Delta|_k]_t$ is independent of the circuit size. As for the circuit-dependent part, we essentially follow the same idea in TurboPack [15] including the preprocessing data we need from a circuit-independent preprocessing, with the only exception that the preprocessing data should be authenticated. We refer the readers to [15] and Section 5 for more details.

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

### 2.4 Instantiating the Circuit-Independent Preprocessing

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

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

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

To this end, we make use of two functionalities $\mathcal{F}_{\mathsf{nVOLE}}$ and $\mathcal{F}_{\mathsf{OLE}}^{\mathrm{prog}}$ from [25]. In [25], these two functionalities are used to efficiently prepare Beaver triples using additive sharings. At a high level,

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

As we discussed above, one direct solution would be using the above approach in a black box way and then transforming additive sharings to packed Shamir sharings. However, the direct consequence is that we need to not only pay the same cost as that in [25], but pay the additional cost for the sharing transformation as well. In the following we discuss how to take the advantage of the constant fraction of honest parties when preparing packed Beaver triples.

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

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

sharing when the shares are $(u^1, \ldots, u^n)$. Then we may view that all parties hold the packed Shamir sharing $[\boldsymbol{u}]_{n-1}$. To obtain a degree-$(n-k)$ packed Shamir sharing of $\boldsymbol{u}$, we simply perform a sharing transformation via the standard "mask-open-unmask" approach following from the known techniques [13].

Now the problem is to prepare the MACs for $\boldsymbol{u}$. We observe that in $\mathcal{F}_{\mathsf{nVOLE}}$, for every ordered pair of parties $(P_i, P_j)$, $P_i, P_j$ together hold an additive sharing of $u^i \cdot \Delta^j$. Since each secret $u_\ell$ in $\boldsymbol{u}$ is a linear combination of $(u^1, \ldots, u^n)$, all parties can locally compute an additive sharing of $u_\ell \cdot \Delta^j$ for each $j \in \{1, \ldots, n\}$ and then compute an additive sharing of $\Delta \cdot u_\ell$. To obtain the MACs $[\Delta \cdot \boldsymbol{u}]_{n-k}$, we will perform a sharing transformation again via the standard "mask-open-unmask" approach following from the known techniques [13,22].

In this way, to obtain a pair of authenticated sharings $([\boldsymbol{a}]_{n-k}, [\Delta \cdot \boldsymbol{a}]_{n-k})$, we only need to perform once the transformation from additive sharings to packed Shamir sharings. In addition, we essentially obtain such a pair of authenticated sharing from the same data that is only for one authenticated additive sharing in [25]. As a result, the amount of preprocessing data we need from $\mathcal{F}_{\mathsf{nVOLE}}$ is reduced by a factor of $k = \epsilon n/2$.

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

To this end, we need to reuse the degree-$(n-1)$ packed Shamir sharings $[\boldsymbol{a}]_{n-1}$ and $[\boldsymbol{b}]_{n-1}$ output by $\mathcal{F}_{\mathsf{nVOLE}}$. As that in [25], every ordered pair of parties $(P_i, P_j)$ invokes $\mathcal{F}_{\mathsf{OLE}}^{\mathrm{prog}}$ to compute additive sharings of $a^i \cdot b^j$, where $a^i$ is the $i$-th share of $[\boldsymbol{a}]_{n-1}$ and $b^j$ is the $j$-th share of $[\boldsymbol{b}]_{n-1}$. From additive sharings of $\{a^i \cdot b^j\}_{i,j}$, all parties can locally compute an additive sharing of each $c_\ell = a_\ell \cdot b_\ell$ for all $\ell \in \{1, \ldots, k\}$. Finally, we obtain $[\boldsymbol{c}]_{n-1}$ with authentications by using random sharings $([\boldsymbol{r}]_{n-1}, \{\langle \Delta \cdot r_\ell \rangle\}_{\ell=1}^k)$ and follow the standard "mask-open-unmask" approach. Note that $([\boldsymbol{r}]_{n-1}, \{\langle \Delta \cdot r_\ell \rangle\}_{\ell=1}^k)$ can be directly obtained from $\mathcal{F}_{\mathsf{nVOLE}}$ by properly interpreting the output of $\mathcal{F}_{\mathsf{nVOLE}}$ as we discussed above.

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

*Remarks About Our Techniques.* Note that we essentially follow the same steps as those in [25] but interpreting the output differently, and then perform sharing transformations to obtain sharings in the desired form. We would like to point out that *following the same steps as those in [25]* is crucial since in [25], $\mathcal{F}_{\mathsf{nVOLE}}$ only outputs random seeds to parties and the parties need to compute their shares by locally expanding the seeds using a proper PRG. And the same seeds are fed in $\mathcal{F}_{\mathsf{OLE}}^{\mathrm{prog}}$ to compute the product sharings. Only in this way together with proper realizations of $\mathcal{F}_{\mathsf{nVOLE}}$ and $\mathcal{F}_{\mathsf{OLE}}^{\mathrm{prog}}$, [25] can achieve sub-linear communication complexity in preparing Beaver triples (without authenticating the product sharing $\langle c \rangle$). Thus, to be able to properly use the functionalities in [25], we should follow a similar pattern to that in [25].

*Verification of Packed Beaver Triples.* We note that the packed Beaver triples we obtained may be incorrect. This is because the invocations of $\mathcal{F}_{\mathsf{OLE}}^{\mathrm{prog}}$ are between every pair of parties and the functionality $\mathcal{F}_{\mathsf{OLE}}^{\mathrm{prog}}$ does not force the same party to use the same input across different invocations. Also when the product sharings are authenticated, corrupted parties may introduce additive errors. The same issues also appear in [25].

To obtain correct packed Beaver triples with authentications, our idea is to extend the technique of sacrificing [12] and use one possibly incorrect packed Beaver triple to check another possibly incorrect packed Beaver triple. To improve the concrete efficiency, we show that it is sufficient to have the sacrificed packed Beaver triple prepared in the form:

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

I.e., we do not need to do any sharing transformation for the first two pairs of sharings and only need to authenticate the product sharing. We defer the details to the full version of this paper due to space constraints.

## 3 Preliminaries

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

We focus on functions which can be represented as an arithmetic circuit $C$ over a finite field $\mathbb{F}$ with input, addition, multiplication, and output gates.[9] The circuit $C$ takes inputs $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m)$ and returns $(\boldsymbol{y}_1, \ldots, \boldsymbol{y}_m)$, where $\boldsymbol{x}_i \in \mathbb{F}^{I_i}$ and $\boldsymbol{y}_i \in \mathbb{F}^{O_i}$, for $i \in \{1, \ldots, m\}$. We use the convention of labeling wires by means of greek letters (*e.g.* $\alpha, \beta, \gamma$), and we use $v_\alpha$ to denote the value stored in a wire labeled by $\alpha$ for a given execution. We use $\kappa$ to denote the security parameter, and we assume that $|\mathbb{F}| \geq 2^\kappa$. We assume that the number of parties $n$ and the circuit size $|C|$ are bounded by polynomials of the security parameter $\kappa$.

We study the dishonest majority setting where the adversary corrupts a majority of the parties, but we focus on the case where the number of corruptions may not be equal to $n-1$. Instead, the adversary corrupts $t < n(1-\epsilon)$ parties for some constant $0 < \epsilon < 1/2$. For security we use Canetti's UC framework [8], where security is argued by the indistinguishability of an ideal world, modeled by a *functionality* (denoted in this work by the letter $\mathcal{F}$ and some subscript), and

[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.

the real world, instantiated by a *protocol* (denoted using the letter $\Pi$ and some subscript). Protocols can also use *procedures*, denoted using the lowercase letter $\pi$ and some subscript, which are like protocols except they are not intended to instantiate a given functionality, and instead they are used as "macros" inside other protocols that instantiate some functionality. The details on the security definition will be included in the full version of this paper.

We denote by $\mathcal{F}_{\mathsf{MPC}}$ the functionality that receives inputs from the clients, evaluates the function $f$, and returns output to the clients. This is given in detail in the full version of this paper. Security with unanimous abort, where all honest parties may jointly abort in the computation, is the best that can be achieved in the dishonest majority setting. Here we achieve security with selective abort, where the adversary can choose which honest parties abort, which can be compiled to unanimous abort using a broadcast channel [19]. To accommodate for aborts, every functionality in this work implicitly allows the adversary to send an abort signal to a specific honest party. We do not write this explicitly.

**Packed Shamir Secret Sharing.** In our work, we make use of packed Shamir secret sharing, introduced by Franklin and Yung [16]. This is a generalization of the standard Shamir secret sharing scheme [26]. Let $n$ be the number of parties and $k$ be the number of secrets to pack in one sharing. A *degree-$d$* ($d \geq k-1$) packed Shamir sharing of $\boldsymbol{x} = (x_1, \ldots, x_k) \in \mathbb{F}^k$ 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 $\boldsymbol{x}$. If $d-(k-1) \geq t$, then knowing $t$ of the shares does not leak anything about the $k$ secrets. In particlar, a sharing of degree $t+(k-1)$ keeps hidden the underlying $k$ secret.

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

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

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

When the packing parameter $k = 1$, a packed Shamir sharing degrades to a Shamir sharing. Generically, a Shamir sharing uses the default evaluation point 0 to store the secret. In our work, we are interested in using different evaluation points in different Shamir secret sharings. Concretely, for all $i \in \{1, \ldots, k\}$, 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 $f(-i + 1) = x$.

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

**Some Functionalities.** For our protocols we assume the existence of two widely used functionalities. One is $\mathcal{F}_{\mathsf{Coin}}$, which upon being called provides the parties with a uniformly random value $r \in \mathbb{F}$. This can be easily implemented by having the parties open some random shared value $\langle r \rangle$, and if more coins are needed these can be expanded with the help of a PRG. The second functionality is $\mathcal{F}_{\mathsf{Commit}}$, which enables the parties to commit to some values of their choice without revealing them to the other parties. At a later point, the parties can open their committed values with the guarantee that these opened terms are exactly the same that were committed to initially. This can be instantiated with the help of a hash function, modeled as a random oracle (*cf.* [12]).

## 4 Online Protocol

We begin by describing the online phase of SuperPack.

### 4.1 Circuit-Dependent Preprocessing Functionality

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

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

1. **Assign Random Values to Wires in $C$**: $\mathcal{F}_{\mathsf{PrepMal}}$ receives the circuit $C$ from all parties. Then $\mathcal{F}_{\mathsf{PrepMal}}$ assigns random values to wires in $C$ as follows.
   (a) For each output wire $\alpha$ 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. **Settling Authentication Keys**: $\mathcal{F}_{\mathsf{PrepMal}}$ samples a random value $\Delta$. Then $\mathcal{F}_{\mathsf{PrepMal}}$ samples $k$ random degree-$t$ Shamir sharings $([\Delta|_1]_t, \ldots, [\Delta|_k]_t)$ and distributes the shares to all parties.
3. **Preparing Packed Beaver Triples with Authentications**: For each group of $k$ multiplication gates, $\mathcal{F}_{\mathsf{PrepMal}}$ samples a random packed Beaver triple with authentications as follows:
   (a) $\mathcal{F}_{\mathsf{PrepMal}}$ samples two random vectors $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{F}_p^k$ and computes $\Delta \cdot \boldsymbol{a}, \Delta \cdot \boldsymbol{b}$. Then $\mathcal{F}_{\mathsf{PrepMal}}$ samples two pairs of random degree-$(n-k)$ packed Shamir sharings $[\![\boldsymbol{a}]\!]_{n-k} = ([\boldsymbol{a}]_{n-k}, [\Delta \cdot \boldsymbol{a}]_{n-k})$, $[\![\boldsymbol{b}]\!]_{n-k} = ([\boldsymbol{b}]_{n-k}, [\Delta \cdot \boldsymbol{b}]_{n-k})$.
   (b) $\mathcal{F}_{\mathsf{PrepMal}}$ computes $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$ and $\Delta \cdot \boldsymbol{c}$. Then $\mathcal{F}_{\mathsf{PrepMal}}$ samples a random degree-$(n-1)$ packed Shamir sharing $[\boldsymbol{c}]_{n-1}$. For all $i \in \{1, \ldots, k\}$, $\mathcal{F}_{\mathsf{PrepMal}}$ samples a random additive sharing $\langle \Delta \cdot c_i \rangle$.
   $\mathcal{F}_{\mathsf{PrepMal}}$ distributes the shares of $([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, ([\boldsymbol{c}]_{n-1}, \{\langle \Delta \cdot c_i \rangle\}_{i=1}^k))$ to all parties.
4. **Distributing $\boldsymbol{\lambda_\alpha} - \boldsymbol{a}$ and $\boldsymbol{\lambda_\beta} - \boldsymbol{b}$ to $P_1$**: For each group of multiplication gates, let $\boldsymbol{\alpha}, \boldsymbol{\beta}$ denote the batch of first input wires and that of the second input wires respectively. Let $([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, ([\boldsymbol{c}]_{n-1}, \{\langle \Delta \cdot c_i \rangle\}_{i=1}^k)$ be the packed Beaver triple with authentications associated with these gates.
   $\mathcal{F}_{\mathsf{PrepMal}}$ receives two vectors of additive errors $\boldsymbol{\delta_\alpha}, \boldsymbol{\delta_\beta}$ from the adversary, computes $\boldsymbol{\lambda_\alpha} - \boldsymbol{a} + \boldsymbol{\delta_\alpha}$ and $\boldsymbol{\lambda_\beta} - \boldsymbol{b} + \boldsymbol{\delta_\beta}$, and sends them to $P_1$. Here $\boldsymbol{\lambda_\alpha}$ and $\boldsymbol{\lambda_\beta}$ are the random values associated with the wires $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$.
   $\mathcal{F}_{\mathsf{PrepMal}}$ also samples random additive sharings $\{\langle \Delta \cdot (\lambda_{\alpha_i} - a_i) \rangle, \langle \Delta \cdot (\lambda_{\beta_i} - b_i) \rangle\}_{i=1}^k$ and distributes the shares to all parties.
5. **Preparing Authenticated Packed Sharings for Multiplication Gates**: For each group of multiplication gates with output wires $\boldsymbol{\gamma}$, $\mathcal{F}_{\mathsf{PrepMal}}$ samples
   – A random degree-$(n-1)$ packed Shamir sharing $[\boldsymbol{\lambda_\gamma}]_{n-1}$,
   – $k$ additive sharings $\{\langle \Delta \cdot \lambda_{\gamma_i} \rangle\}_{i=1}^k$,
   and distributes the shares to honest parties.
6. **Preparing Random Sharings for Input and Output Gates**: For each group of $k$ input gates or output gates, $\mathcal{F}_{\mathsf{PrepMal}}$ prepares the following random sharings.
   (a) Let $\boldsymbol{\alpha}$ be the output wires of these $k$ input gates or the input wires of these $k$ output gates. $\mathcal{F}_{\mathsf{PrepMal}}$ samples
      – A random degree-$(n-1)$ packed Shamir sharing $[\boldsymbol{\lambda_\alpha}]_{n-1}$,
      – $k$ additive sharings $\{\langle \Delta \cdot \lambda_{\alpha_i} \rangle\}_{i=1}^k$,
      and distributes the shares to honest parties.
   (b) $\mathcal{F}_{\mathsf{PrepMal}}$ also prepares a random packed Beaver triple with authentications $([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, ([\boldsymbol{c}]_{n-1}, \{\langle \Delta \cdot c_i \rangle\}_{i=1}^k)$ in the same way as Step 3. Later, we will view $\boldsymbol{b}$ as an authentication key and $\boldsymbol{c}$ as the MAC of $\boldsymbol{a}$. This allows the input holder to verify the correctness of $\boldsymbol{a}$.

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

### 4.2 Input Gates

In this section, we give the description of the procedure $\pi_{\mathsf{Input}}$. This procedure enables $P_1$ to learn $\mu_\alpha = v_\alpha - \lambda_\alpha$ for every input wire $\alpha$, where $v_\alpha$ is the input provided by the client owning the input gate. In addition, the parties output shares of the MAC of this value, namely $\{\langle \Delta \cdot \mu_{\alpha_i} \rangle\}_{i=1}^k$. Recall that in $\mathcal{F}_{\mathsf{PrepMal}}$, we prepared a packed Beaver triple with authentications $([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, ([\boldsymbol{c}]_{n-1}, \{\langle \Delta \cdot c_i \rangle\}_{i=1}^k)$ for each group of input gates. Here $\boldsymbol{b}$ serves as the MAC key and $\boldsymbol{c}$ serves as the MAC of $\boldsymbol{a}$ so that the client can verify that he receives the correct $\boldsymbol{a}$ in $\pi_{\mathsf{Input}}$. The description of $\pi_{\mathsf{Input}}$ appears below.

**Procedure 1:** $\pi_{\mathsf{Input}}$

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

   Let $\boldsymbol{v_\alpha}$ denote the inputs held by `Client`.
2. All parties send to `Client` their shares of $[\boldsymbol{\lambda_\alpha}]_{n-1}$, $[\boldsymbol{a}]_{n-k}, [\boldsymbol{b}]_{n-k}, [\boldsymbol{c}]_{n-1}$.
3. `Client` reconstructs the secrets $\boldsymbol{\lambda_\alpha}, \boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}$ and checks whether $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$. If not, `Client` aborts. Otherwise, `Client` computes $\boldsymbol{\mu_\alpha} = \boldsymbol{v_\alpha} - \boldsymbol{\lambda_\alpha}$ and $[\boldsymbol{v_\alpha} - \boldsymbol{a}]_{2k-2}$.
4. `Client` sends $\boldsymbol{\mu_\alpha}$ to $P_1$ and distributes the shares of $[\boldsymbol{v_\alpha} - \boldsymbol{a}]_{2k-2}$ to all parties.
5. For all $i \in \{1, \ldots, k\}$, all parties locally compute $\langle \Delta \cdot \mu_{\alpha_i} \rangle$ as follows:
   (a) Recall that all parties hold $[\Delta|_i]_t$ generated in $\mathcal{F}_{\mathsf{PrepMal}}$. All parties locally compute $[\Delta \cdot (v_{\alpha_i} - a_i)|_i]_{n-1} = [\Delta|_i]_t * [\boldsymbol{v_\alpha} - \boldsymbol{a}]_{2k-2}$. Then all parties locally transform it to an additive sharing $\langle \Delta \cdot (v_{\alpha_i} - a_i) \rangle$.
   (b) Recall that all parties hold $[\Delta \cdot \boldsymbol{a}]_{n-k}$. All parties locally transform it to an additive sharing $\langle \Delta \cdot a_i \rangle$.
   (c) Recall that all parties hold $\langle \Delta \cdot \lambda_{\alpha_i} \rangle$. All parties locally compute $\langle \Delta \cdot \mu_{\alpha_i} \rangle = \langle \Delta \cdot (v_{\alpha_i} - a_i) \rangle + \langle \Delta \cdot a_i \rangle - \langle \Delta \cdot \lambda_{\alpha_i} \rangle$.

### 4.3 Computing Addition and Multiplication Gates

After receiving the inputs from all clients, all parties start to evaluate the circuit gate by gate. We will maintain the invariant that for each output wire $\alpha$ of an input gate or a multiplication gate, $P_1$ learns $\mu_\alpha$ in clear. In the procedure, $P_1$ distributes shares of certain values, which may be incorrect. To prevent cheating,

the parties get additive shares of the MAC of these values, which are used in a verification step in the output phase to check for correctness.

**Procedure 2:** $\pi_{\mathsf{Mult}}$

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

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

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

### 4.4 Output Gates and Verification

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

Due to space constraints, we describe the procedure $\pi_{\mathsf{Output}}$ in detail in the full version of this paper, including the computation of the output gates, the verification of the computation (degree and MAC check), and the reconstruction of the outputs.

### 4.5 Full Online Protocol

Our final online protocol makes use of the procedures $\pi_{\mathsf{Input}}$ (Proc. 1, Section 4.2) to let the clients distribute their inputs, $\pi_{\mathsf{Mult}}$ (Proc. 2, Section 4.3) to process each group of $k$ multiplication gates, and $\pi_{\mathsf{Output}}$ (Section 4.4) to verify the correctness of the computation and reconstruct output to the clients. $\pi_{\mathsf{Output}}$ and the online protocol $\Pi_{\mathsf{Online}}$ are presented in detail in the full version. We prove the following:

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

**Communication complexity of $\Pi_{\mathsf{Online}}$.** Let $I$ and $O$ be the number of input wires and output wires, and assume that each client owns a number of input and output gates that is a multiple of $k$. We assume for simplicity that $n$ divides each of these terms, and also that $n$ divides the number of multiplication gates in each layer. Let us also denote by $|C|$ the number of multiplication gates in the circuit $C$. The total communication complexity is given by $\frac{4}{\epsilon} \cdot (I+O) + \frac{6}{\epsilon} \cdot |C|$, ignoring small terms that are independent of $I$, $O$ and $|C|$.

## 5 Circuit-Dependent Preprocessing Phase

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

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

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

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

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

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

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

To instantiate the circuit-dependent preprocessing functionality $\mathcal{F}_{\mathsf{PrepMal}}$ using the circuit-independent preprocessing $\mathcal{F}_{\mathsf{PrepIndMal}}$, we follow the idea in [15]. We describe the protocol $\Pi_{\mathsf{PrepMal}}$ below.

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

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

[a] We will discuss how parties locally refresh an additive sharing in the full version.

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

Lemma 1 is proven in the full version of this paper.

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

## 6 Circuit-Independent Preprocessing Phase

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

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

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

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

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

The functionalities above are presented in full detail in the full version of this paper. At a high level, $\mathcal{F}_{\mathsf{nVOLE}}$ is used to generate authenticated sharings

of a uniformly random value, and $\mathcal{F}_{\mathsf{OLE}}^{\mathrm{prog}}$, which allows the parties to set theit inputs, is used to secure multiply two already-shared secret values. $\mathcal{F}_{\mathsf{nVOLE}}$ can be instantiated using pseudo-random correlator generators, as suggested in [25]. On the other hand, for $\mathcal{F}_{\mathsf{OLE}}^{\mathrm{prog}}$ we can use the implementation from [25]. As we are using *exactly* the same functionalities as in [25], we refer the reader to that work for instantiations and complexity measures.

**Omitted procedures.** For our triple generation protocol we will make use of a series of procedures that are described in full detail in the full version of this paper. These procedures are the following:

- $\pi_{\mathsf{RandSh}}$: this procedure generates sharings (under some secret-sharing scheme, which will be clear from context) of uniformly random values. For this, the trick of using a Vandermonde matrix for randomness extraction from [13] is used.
- $\pi_{\mathsf{DegReduce}}$: this procedure takes as input a sharing $[\boldsymbol{u}]_{n-1}$ and outputs $[\boldsymbol{u}]_{n-k}$. This is achieved by using the trick of masking with a random value $[\boldsymbol{r}]_{n-1}$, opening, and unmasking with $[\boldsymbol{r}]_{n-k}$. This random pair is generated using $\pi_{\mathsf{RandSh}}$.
- $\pi_{\mathsf{AddTran}}$: this procedure takes as input sharings $(\langle \Delta \cdot u_1 \rangle, \ldots, \langle \Delta \cdot u_k \rangle)$ and converts them to $[\Delta \cdot \boldsymbol{u}]_{n-k}$. Once again, the trick of masking with a random sharing $\langle r_1 \rangle, \ldots, \langle r_k \rangle$, opening, and unmasking with $[\boldsymbol{r}]$, is used. The sharing $[\boldsymbol{r}]$ is obtain using $\pi_{\mathsf{RandSh}}$, and each $\langle r_i \rangle$ can be derived from it non-interactively.
- $\pi_{\mathsf{MACKey}}$: this procedure enables the parties to obtain individual Shamir sharings of the global MAC key $[\Delta|_1]_t, \ldots, [\Delta|_k]_t$, starting from additive shares of it $\langle \Delta \rangle$ which are obtained using $\mathcal{F}_{\mathsf{nVOLE}}$. This is done by using the standard trick of masking with a random value $\langle r \rangle$, opening, and unmasking with each $[r|_i]_t$. These random sharings are obtained using $\pi_{\mathsf{RandSh}}$.
- $\pi_{\mathsf{Auth}}$: this procedure takes as input sharings $(\langle u_1 \rangle, \ldots, \langle u_k \rangle)$ to $([\boldsymbol{u}]_{n-1}, \{\langle \Delta \cdot u_i \rangle\}_{i=1}^k)$. The trick here is to mask with $\langle r_1 \rangle, \ldots, \langle r_k \rangle$, open, and adding $[\boldsymbol{r}]_{n-1}$ to obtain $[\boldsymbol{u}]_{n-1}$. The authenticated part can be obtained by first multiplying locally by each $[\Delta|_i]_t$ and then adding each $\langle \Delta \cdot r_i \rangle$. The pair $([\boldsymbol{r}]_{n-1}, \{\langle \Delta \cdot r_i \rangle\}_{i=1}^k)$ is produced using $\pi_{\mathsf{RandSh}}$.

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

**Procedure 4:** $\pi_{\mathsf{Triple}}$

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

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

**Generation**:

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

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

*Communication complexity of* $\pi_{\mathsf{Triple}}$. This is derived as follows

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

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

*Remark 1 (On the output size of* $\mathcal{F}_{\mathsf{OLE}}^{prog}$ *and* $\mathcal{F}_{\mathsf{nVOLE}}$*).* We make the crucial observation that, in order to obtain $m$ packed multiplication triples, we require the `Expand` function used in Functionalities $\mathcal{F}_{\mathsf{OLE}}^{\mathrm{prog}}$ and $\mathcal{F}_{\mathsf{nVOLE}}$ to output $m$ field elements. However, since each such packed triple is used for a group of $k$ multiplication gates, this effectively means that, if there are $|C|$ multiplication gates

in total, we only require `Expand` to output $|C|/k \approx 2|C|/(\epsilon n)$ correlations. In contrast, as we will see in the full version of this paper, the best prior work Turbospeedz [3], when instantiated with the preprocessing from Le Mans [25], would require $|C|$ correlations from the $\mathcal{F}_{\mathsf{nVOLE}}$ and $\mathcal{F}_{\mathsf{OLE}}^{\mathrm{prog}}$. As a result, we manage to reduce by a factor of $k$ the expansion requirements on VOLE/OLE techniques, which has a direct effect on the resulting efficiency since this allows us to choose better parameters for the realizations of $\mathcal{F}_{\mathsf{OLE}}^{\mathrm{prog}}$ and $\mathcal{F}_{\mathsf{nVOLE}}$. We do not explore these concrete effects in efficiency as it goes beyond the scope of our work, but we refer the reader to [25] where an instantiation of $\mathcal{F}_{\mathsf{OLE}}^{\mathrm{prog}}$ and a discussion on PCG-based $\mathcal{F}_{\mathsf{nVOLE}}$ is presented.

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

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

## 7 Implementation and Experimental Results

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

*Implementation setup.* We implement SuperPack by using as a baseline the code of TurboPack [15].[10][11] As TurboPack, our program is written in `C++` with no dependencies beyond the standard library. Our implementation includes fully functional networking code. However, for the experiments, we deploy the protocol as multiple processes in a single machine, and emulate real network conditions using the package `netem`[12], which allows us to set bandwidth and latency constraints. We use the same machine as in [15] for the experiments, namely an AWS `c5.metal` instance with 96 vCPUs and 192 GiB of memory. For our protocol, we use a finite field $\mathbb{F} = \mathbb{F}_p$ where $p = 2^{61} - 1$. We explore how the performance of our protocol is affected by the parameters including the number of parties $n$, the width and depth of the circuit, the network bandwidth and the values of $\epsilon$ such that $t = n(1 - \epsilon)$ is the threshold for corrupted parties.

[10] TurboPack is available at https://github.com/deescuderoo/turbopack
[11] SuperPack is available at https://github.com/ckweng/SuperPack
[12] https://wiki.linuxfoundation.org/networking/netem

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

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

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

*Experimental comparison to Turbospeedz.* Now we compare the online phase of our protocol and compare it against that of Turbospeedz [3],[13] for a varying number of parties $n$ and parameter $\epsilon$. We fix the circuit to have width 100k and depth 10, but we vary the bandwidth in $\{500, 100, 50, 10\}$mbps. The results are

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

given in Table 3. Notice that we report the improvement factor of our online phase with respect to that of Turbospeedz. The concrete runtimes will be available in the full version. We also report the communication factors between our protocol and Turbospeedz, for reference.

Table 3 shows interesting patterns. First, as expected (and as analyzed theoretically in the full version), our improvement factor with respect to Turbospeedz improves (*i.e.* increases) as the number of parties grows—since in this case communication in Turbospeedz grows but in our case remains constant—or as the percentage of corruptions decreases—since in this case we can pack more secrets per sharing. Now, notice the following interesting behavior. The last rows next to the "comm. factor" rows represent the improvement factor of our online phase with respect to Turbopeedz, in terms of *communication*. In principle, this is the improvement factor we would expect to see in in terms of runtimes. However, we observe that the expected factor is only reasonably close to the experimental ones for low bandwidths such as 10, 50 and 100 mbps. For the larger bandwidth of 500 mbps, we see that the experimental improvement factors are much lower than the ones we would expect, and in fact, there are several cases where we expect our protocol to be even slightly better, and instead it performs *worse*.

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

## Acknowledgments

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

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

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



## References

1. Beaver, D.: Efficient multiparty protocols using circuit randomization. pp. 420–432 (1992). https://doi.org/10.1007/3-540-46766-1_34
2. Beck, G., Goel, A., Jain, A., Kaptchuk, G.: Order-C secure multiparty computation for highly repetitive circuits. pp. 663–693 (2021). https://doi.org/10.1007/978-3-030-77886-6_23
3. Ben-Efraim, A., Nielsen, M., Omri, E.: Turbospeedz: Double your online SPDZ! Improving SPDZ using function dependent preprocessing. pp. 530–549 (2019). https://doi.org/10.1007/978-3-030-21568-2_26
4. Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). pp. 1–10 (1988). https://doi.org/10.1145/62212.62213

5. Bendlin, R., Damgård, I., Orlandi, C., Zakarias, S.: Semi-homomorphic encryption and multiparty computation. pp. 169–188 (2011). https://doi.org/10.1007/978-3-642-20465-4_11
6. Boneh, D., Boyle, E., Corrigan-Gibbs, H., Gilboa, N., Ishai, Y.: Zero-knowledge proofs on secret-shared data via fully linear PCPs. pp. 67–97 (2019). https://doi.org/10.1007/978-3-030-26954-8_3
7. Boyle, E., Gilboa, N., Ishai, Y., Nof, A.: Efficient fully secure computation via distributed zero-knowledge proofs. pp. 244–276 (2020). https://doi.org/10.1007/978-3-030-64840-4_9
8. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. pp. 136–145 (2001). https://doi.org/10.1109/SFCS.2001.959888
9. Chida, K., Genkin, D., Hamada, K., Ikarashi, D., Kikuchi, R., Lindell, Y., Nof, A.: Fast large-scale honest-majority MPC for malicious adversaries. pp. 34–64 (2018). https://doi.org/10.1007/978-3-319-96878-0_2
10. Couteau, G.: A note on the communication complexity of multiparty computation in the correlated randomness model. pp. 473–503 (2019). https://doi.org/10.1007/978-3-030-17656-3_17
11. Damgård, I., Ishai, Y., Krøigaard, M.: Perfectly secure multiparty computation and the computational overhead of cryptography. pp. 445–465 (2010). https://doi.org/10.1007/978-3-642-13190-5_23
12. Damgård, I., Keller, M., Larraia, E., Pastro, V., Scholl, P., Smart, N.P.: Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. pp. 1–18 (2013). https://doi.org/10.1007/978-3-642-40203-6_1
13. Damgård, I., Nielsen, J.B.: Scalable and unconditionally secure multiparty computation. pp. 572–590 (2007). https://doi.org/10.1007/978-3-540-74143-5_32
14. Damgård, I., Pastro, V., Smart, N.P., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. pp. 643–662 (2012). https://doi.org/10.1007/978-3-642-32009-5_38
15. Escudero, D., Goyal, V., Polychroniadou, A., Song, Y.: TurboPack: Honest majority MPC with constant online communication. pp. 951–964 (2022). https://doi.org/10.1145/3548606.3560633
16. Franklin, M.K., Yung, M.: Communication complexity of secure computation (extended abstract). pp. 699–710 (1992). https://doi.org/10.1145/129712.129780
17. Genkin, D., Ishai, Y., Polychroniadou, A.: Efficient multi-party computation: From passive to active security via secure SIMD circuits. pp. 721–741 (2015). https://doi.org/10.1007/978-3-662-48000-7_35
18. Genkin, D., Ishai, Y., Prabhakaran, M.M., Sahai, A., Tromer, E.: Circuits resilient to additive attacks with applications to secure computation. In: Proceedings of the Forty-sixth Annual ACM Symposium on Theory of Computing. pp. 495–504. STOC '14, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2591796.2591861, http://doi.acm.org/10.1145/2591796.2591861
19. Goldwasser, S., Lindell, Y.: Secure multi-party computation without agreement. J. Cryptol. **18**(3), 247–287 (jul 2005). https://doi.org/10.1007/s00145-005-0319-z, https://doi.org/10.1007/s00145-005-0319-z
20. Goyal, V., Li, H., Ostrovsky, R., Polychroniadou, A., Song, Y.: ATLAS: Efficient and scalable MPC in the honest majority setting. pp. 244–274 (2021). https://doi.org/10.1007/978-3-030-84245-1_9
21. Goyal, V., Polychroniadou, A., Song, Y.: Unconditional communication-efficient MPC via hall's marriage theorem. pp. 275–304 (2021). https://doi.org/10.1007/978-3-030-84245-1_10

22. Goyal, V., Polychroniadou, A., Song, Y.: Sharing transformation and dishonest majority MPC with packed secret sharing. pp. 3–32 (2022). https://doi.org/10.1007/978-3-031-15985-5_1
23. Goyal, V., Song, Y.: Malicious security comes free in honest-majority MPC. Cryptology ePrint Archive, Report 2020/134 (2020), https://eprint.iacr.org/2020/134
24. Lindell, Y., Nof, A.: A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. pp. 259–276 (2017). https://doi.org/10.1145/3133956.3133999
25. Rachuri, R., Scholl, P.: Le mans: Dynamic and fluid MPC for dishonest majority. pp. 719–749 (2022). https://doi.org/10.1007/978-3-031-15802-5_25
26. Shamir, A.: How to Share a Secret. Commun. ACM **22**(11), 612–613 (Nov 1979). https://doi.org/10.1145/359168.359176, http://doi.acm.org/10.1145/359168.359176