ATLAS: Efficient and Scalable MPC in the Honest Majority Setting

Vipul Goyal
Carnegie Mellon University
and NTT Research
goyal@cs.cmu.edu

Hanjun Li
University of Washington
lihanjun1212@gmail.com

Rafail Ostrovsky
University of California, Los Angeles
rafail@cs.ucla.edu

Antigoni Polychroniadou
J.P. Morgan AI Research
antigonipoly@gmail.com

Yifan Song
Carnegie Mellon University
yifans2@andrew.cmu.edu

Abstract

In this work, we address communication, computation, and round efficiency of unconditionally secure multi-party computation for arithmetic circuits in the honest majority setting. We achieve both algorithmic and practical improvements:

- The best known result in the semi-honest setting has been due to Damgård and Nielsen (CRYPTO 2007). Over the last decade, their construction has played an important role in the progress of efficient secure computation. However despite a number of follow-up works, any significant improvements to the basic semi-honest protocol have been hard to come by. We show 33% improvement in communication complexity of this protocol. We show how to generalize this result to the malicious setting, leading to the best known unconditional honest majority MPC with malicious security.

- We focus on the round complexity of the Damgård and Nielsen protocol and improve it by a factor of 2. Our improvement relies on a novel observation relating to an interplay between Damgård and Nielsen multiplication and Beaver triple multiplication. An implementation of our constructions shows an execution run time improvement compared to the state of the art ranging from 30% to 50%.

1 Introduction

Secure Multi-Party Computation (MPC) allows $n \geq 2$ parties to compute a function on privately held inputs, such that the desired output is correctly computed and is the only new information released. This should hold even if t out of n parties have been corrupted by a semi-honest or malicious adversary. Since its introduction in the 1980s [Yao82, GMW87], a lot of research has been done to improve the efficiency of MPC protocols. Thanks to these efforts, MPC has rapidly moved from theory to practice.

In this work, our focus is on honest majority protocols in the presence of a malicious adversary. We note that the fastest known implementations of MPC have come in the honest majority setting, which does not necessarily require public key operations. For example, the recent work of Chida et al. [CGH+18] showed that their secure-with-abort protocol can evaluate 1 million multiplication gates within 1 second for up to 7 parties, 4 seconds for 50 parties, and 8 seconds for 110 parties. Another attractive feature of the honest

H. Li—Work done in part while at CMU.

majority setting is that it allows one to achieve the stronger properties of fairness and guaranteed output delivery which are otherwise impossible with dishonest majority.

For over a decade, the most efficient MPC protocol with semi-honest security in the honest majority setting has been the protocol of Damgård and Nielsen [DN07], hereafter known as the DN protocol. By using the Shamir secret sharing scheme [Sha79], addition gates can be evaluated without any communication. To evaluate a multiplication gate, each party only needs to communicate 6 field elements. In the computational setting, the communication complexity can be reduced to 3 field elements by using pseudo-random generators [NV18] (improved further to 1.5 elements by Boneh et al. [BBCG$^+$19] for a constant number of parties). Due to its simplicity and efficiency, many subsequent works have used the DN protocol to achieve security-with-abort [GIP$^+$14, CGH$^+$18, NV18, BBCG$^+$19, GSZ20] or guaranteed output delivery [BSFO12, GSZ20].

Despite the important role played by the DN protocol in the honest majority setting, any improvement to the basic protocol has been hard to come by unless one resorts to other approaches using computational assumptions. An exception is the recent work of Goyal et al. [GSZ20] who proposed a marginal improvement over DN of 6 field elements per multiplication gate to 5.5 field elements.

1.1 Our Contributions

We propose ATLAS, an unconditionally secure MPC protocol in the honest majority setting with reduced communication complexity over the celebrated DN protocol even in the honest but curious setting, as well as malicious setting. Our protocol ATLAS enjoys the following efficiency improvements over the DN protocol:

- We improve the basic DN protocol leading to a communication complexity of 4 field elements per multiplication gate per party. Our results are in the information-theoretic setting assuming a majority of the parties are honest and the adversary is semi-honest. This leads to the most communication-efficient semi-honest MPC protocol with honest majority.

- We note that the recent works [BBCG$^+$19, GSZ20] compiled the DN protocol to get security-with-abort without increasing the communication complexity. We show that our protocol continues to satisfy the properties needed for this compilation to work. It allows us to present a secure-with-abort protocol with only 4 field elements per multiplication gate per party in the information-theoretic setting.

- Next, we focus on the round complexity of the DN protocol. Instead of evaluating multiplication gates of the same layer in parallel, we show how to evaluate all multiplication gates in a two-layer circuit in parallel. This allows us to improve the concrete efficiency even further and reduce the number of rounds by a factor of 2. The achieved amortized communication cost per multiplication gate in this setting is 4.5 field elements per party but halving the number of rounds.

- In the computational setting, where one can use pseudo-random generators based on any one-way function (in practice, one can use an AES based PRG in counter-mode), we show how to further reduce the communication complexity by making black-box use of any pseudo-random generator. The concrete efficiency can be improved to 2 field elements per party per gate in both semi-honest and secure-with-abort settings, and 2.5 field elements for the variant with the improvement of round complexity.

We implement ATLAS in the information-theoretic setting and compare with the previously best-known results [CGH$^+$18, GSZ20] in the setting of security-with-abort. We measure the running time for circuits with 1 million and 10 million multiplication gates, with circuit depth from 20 to 10,000, and the number of parties from 3 to 21. By combining improvements on both communication and round complexity, our protocol shows around 2x speedup comparing with the protocol in [CGH$^+$18], and around 1.4x speedup comparing with the protocol in [GSZ20] in all tested cases.

1.2 Other Related Works

The notion of MPC was first introduced in [Yao82, GMW87] in 1980s. Feasibility results for MPC were obtained by [Yao82, GMW87, CDVdG87] under cryptographic assumptions, and by [BOGW88, CCD88] in the information-theoretic setting. Subsequently, a large number of works have focused on improving the efficiency of MPC protocols in various settings.

A series of works focus on improving the communication efficiency of MPC with guaranteed output delivery in the settings with different thresholds on the number of corrupted parties. In the setting of honest majority setting, assuming the existence of a broadcast channel, the works [BSFO12, GSZ20] have shown that guaranteed output delivery can be achieved efficiently. In the setting where $t < n/3$, a rich line of works [HMP00, HM01, DN07, BTH08, GLS19] have focused on improving the asymptotic communication complexity in this setting. In the setting where $t < (1/3 - \epsilon)n$, packed secret sharing can be used to hide a batch of values, resulting in more efficient protocols. E.g., Damgård et al. [DIK10] introduced a protocol with communication complexity of $O(C \log C \log n \cdot \kappa + D_M^2 \text{poly}(n, \log C)\kappa)$ bits.

A rich line of works have also focused on the performance of MPC in practice for two parties [LP12, NNOB12], or three parties [FLNW17, ABF+17].

1.3 Road Map

The overall structure of our paper is as follows. In Section 2, we give an overview of our techniques. In Section 3, we introduce our model, the standard Shamir secret sharing scheme [Sha79], and several building blocks used in our main construction. In Section 4, we introduce our improvements of the secure-with-abort MPC protocol of [GSZ20]. In Section 5, we show the experiment results of our secure-with-abort MPC protocols comparing with the previous works [CGH+18, GSZ20].

2 Technical Overview

We give an overview of our techniques in this section. In the following, we will use n to denote the number of parties and t to denote the number of corrupted parties. In the setting of the honest majority, we have $n = 2t + 1$. Our construction is based on the standard Shamir Secret Sharing Scheme [Sha79]. We will use $[x]_d$ to denote a degree-d Shamir sharing, or a $(d + 1)$-out-of-n Shamir sharing. It requires at least $d + 1$ shares to reconstruct the secret and any d shares do not leak any information about the secret.

2.1 Review: The Secure-with-abort MPC Protocol in [GSZ20]

In [GIP+14], Genkin et al. showed that the best-known semi-honest protocol [DN07] (hereafter referred to as the DN protocol) is secure up to an additive attack in the presence of a fully malicious adversary. An additive attack means that the adversary is able to change the multiplication result by adding an arbitrary fixed value. As one corollary, the DN protocol provides full privacy of honest parties before reconstructing the output. Therefore, a straightforward strategy to achieve security-with-abort is to (1) run the DN protocol until the output phase, (2) check the correctness of the computation, and (3) reconstruct the output only if the check passes.

In the DN protocol [DN07], all parties compute a degree-t Shamir sharing for each wire. Since the Shamir secret sharing scheme is linearly homomorphic, addition gates can be evaluated without interaction. Therefore, to achieve security-with-abort, the main task is to verify the multiplications. In [GSZ20], Goyal et al. show that multiplications can be verified with sub-linear communication complexity in the number of multiplications. This allows Goyal et al. to obtain the first secure-with-abort MPC protocol which achieves the same concrete efficiency per gate as the best-known semi-honest protocol [DN07].

To make a further improvement in the concrete efficiency, we focus on the multiplication protocol in [DN07] (hereafter referred to as the DN multiplication protocol). Our idea is to reuse the correlated-randomness required in the DN multiplication protocol.

Review of the DN Multiplication Protocol. To evaluate a multiplication gate, all parties first need to prepare a pair of random sharings $([r]_t, [r]_{2t})$ of the same secret r, where the first sharing is a degree-t Shamir sharing and the second sharing is a degree-$2t$ Shamir sharing. Such a pair of sharings is referred to as a pair of double sharings. In [DN07], preparing a pair of random double sharings requires the communication of 4 elements per party.

For a multiplication gate, suppose the input sharings are denoted by $[x]_t, [y]_t$. To compute $[z]_t := [x \cdot y]_t$, a pair of random double sharings $([r]_t, [r]_{2t})$ is consumed. All parties first agree on a special party $P_{\texttt{king}}$. Then, all parties run the following steps:

1. All parties locally compute $[e]_{2t} := [x]_t \cdot [y]_t + [r]_{2t}$.
2. $P_{\texttt{king}}$ collects all shares of $[e]_{2t}$ and reconstructs the secret e. Then $P_{\texttt{king}}$ sends the value e to all other parties.
3. After receiving e from $P_{\texttt{king}}$, all parties locally compute $[z]_t := e - [r]_t$.

Correctness follows from the properties of the Shamir secret sharing scheme. Note that each party needs to send an element to $P_{\texttt{king}}$, and $P_{\texttt{king}}$ needs to send an element to each party. The communication complexity of this protocol is 2 elements per party. Including the communication cost for preparing double sharings, the overall cost per multiplication gate is 6 elements per party.

2.2 Reducing the Communication Complexity via t-wise Independence

Starting Point. In [GSZ20], Goyal et al. observe that in the second step of the DN multiplication protocol, $P_{\texttt{king}}$ can alternatively distribute a degree-t Shamir sharing $[e]_t$. Then in the last step, all parties can still compute $[z]_t := [e]_t - [r]_t$. This observation leads to an improvement from 6 elements to 5.5 elements. We refer the readers to Section 4.2 for more discussion.

Our main observation is that, when $P_{\texttt{king}}$ is an honest party, the corrupted parties only receive several random elements from $P_{\texttt{king}}$ if $[e]_t$ is a random degree-t Shamir sharing. In particular, it holds even if the corrupted parties know the whole sharings $[r]_t$ and $[r]_{2t}$. This is because the corrupted parties only receive t shares of a random degree-t sharing $[e]_t$ from $P_{\texttt{king}}$, which are uniformly random and independent of the secret. Therefore for an honest $P_{\texttt{king}}$, we do not need the double sharings to be uniformly random at all. While for a corrupted $P_{\texttt{king}}$, we still need to use random double sharings, we can split the tasks of handling multiplication gates as $P_{\texttt{king}}$ to all parties. In this way, at least half of multiplication gates are handled by honest $P_{\texttt{king}}$'s. We show that it allows us to reduce the cost of preparing double sharings by a factor of 2.

Relying on t-wise Independence. Suppose we have n multiplication gates and we let each party behave as $P_{\texttt{king}}$ for 1 multiplication gate. When $P_{\texttt{king}}$ is a corrupted party, we still need to use a pair of random double sharings to protect the secrecy of the result. If $P_{\texttt{king}}$ is an honest party, as argued above, the double sharings do not need to be random.

Our idea is to generate n pairs of double sharings such that any t pairs of them are independent and uniformly random. This guarantees that the double sharings used for multiplication gates handled by corrupted parties are uniformly random, which ensures the security of the MPC protocol. On the other hand, given these double sharings, the other double sharings used for multiplication gates handled by honest parties can be fixed and determined. It means that we only need to prepare t pairs of random and independent double sharings for n multiplication gates.

To this end, all parties agree on a fixed hyper-invertible matrix of size $n \times t$, denoted by $\boldsymbol{M}$. The main property of $\boldsymbol{M}$ is that any $t \times t$ sub-matrix of $\boldsymbol{M}$ is invertible. Since the Shamir secret sharing scheme is a linear homomorphism, a linear combination of several pairs of double sharings is still a pair of double sharings. All parties first prepare t pairs of random double sharings using the protocol in [DN07], denoted by

$$([r^{(1)}]_t, [r^{(1)}]_{2t}), \ldots, ([r^{(t)}]_t, [r^{(t)}]_{2t}).$$

Then, we expand these t pairs of double sharings to n pairs by computing

$$\begin{aligned}([\tilde{r}^{(1)}]_t, \ldots, [\tilde{r}^{(n)}]_t)^{\mathrm{T}} &= \boldsymbol{M}([r^{(1)}]_t, \ldots, [r^{(t)}]_t)^{\mathrm{T}} \\ ([\tilde{r}^{(1)}]_{2t}, \ldots, [\tilde{r}^{(n)}]_{2t})^{\mathrm{T}} &= \boldsymbol{M}([r^{(1)}]_{2t}, \ldots, [r^{(t)}]_{2t})^{\mathrm{T}}.\end{aligned}$$

We point out that this expansion can be done locally without interaction. Note that for all $i \in [n]$, $([\tilde{r}^{(i)}]_t, [\tilde{r}^{(i)}]_{2t})$ is a pair of double sharings. Let $\mathcal{C}$ denote the set of corrupted parties. According to the property of $\boldsymbol{M}$, there is a one-to-one map from $\{([\tilde{r}^{(i)}]_t, [\tilde{r}^{(i)}]_{2t})\}_{i\in\mathcal{C}}$ to $\{([r^{(i)}]_t, [r^{(i)}]_{2t})\}_{i\in[t]}$. Since the input double sharings are independent and uniformly random, we conclude that the double sharings in $\{([\tilde{r}^{(i)}]_t, [\tilde{r}^{(i)}]_{2t})\}_{i\in\mathcal{C}}$ are independent and uniformly random.

When $([\tilde{r}^{(i)}]_t, [\tilde{r}^{(i)}]_{2t})$ is used to evaluate a multiplication gate, we require the party P_i to act as $P_{\mathtt{king}}$. In this way, the multiplication gates handled by corrupted parties will use double sharings in $\{([\tilde{r}^{(i)}]_t, [\tilde{r}^{(i)}]_{2t})\}_{i\in\mathcal{C}}$, which are independent and uniformly random. We are able to show that the security still holds.

Concrete Efficiency of Our Improved Multiplication Protocol. Recall that in [DN07], preparing a pair of random double sharings requires the communication of 4 elements per party. Relying on t-wise independence, we only need to prepare t pairs of random double sharings for n multiplications. Thus, the amortized communication cost per pair of double sharings is $4 \cdot t/n \approx 2$ elements per party. Including the communication cost of the multiplication protocol in [DN07], which is 2 elements per party, the overall cost per multiplication is 4 elements per party.

In Section 4.2, we show that our multiplication protocol can be directly used in the secure-with-abort MPC protocol in [GSZ20]. It yields a secure-with-abort MPC protocol with the concrete efficiency of 4 elements per party per gate.

2.3 Reducing the Number of Rounds via Beaver Triples

In the secure-with-abort MPC protocol in [GSZ20], multiplication gates in the same layer of the circuit are evaluated in parallel.Therefore, the number of rounds is linear in the depth of the circuit. To further improve the concrete efficiency, we pay our attention to the round complexity. We note that the question of obtaining information theoretic constant round protocols for a general circuit has been opened for many years. In particular, it has been shown in [DNPR16] that the dependency on the depth in the round complexity is inherent for the DN protocol. Given this, we managed to reduce the number of rounds by a factor of 2 while maintaining the communication efficiency.

To this end, we first consider a two-layer circuit, and try to evaluate all multiplication gates in parallel.

Starting Point. For a two-layer circuit, an input sharing of a multiplication gate in the second layer may come from three places:

- This sharing is an input sharing of the circuit.
- This sharing is an output sharing of an addition gate in the first layer.
- This sharing is an output sharing of a multiplication gate in the first layer.

Note that an addition gate can be evaluated without interaction. Therefore for the first two cases, all parties can locally compute this sharing. However, for the third case, communication is required to evaluate this multiplication gate in the first layer. Therefore, the question becomes how to evaluate multiplication gates in the second layer *without learning the output sharings of multiplication gates in the first layer*.

A Beaver triple [Bea92] consists of three degree-t Shamir sharings $([a]_t, [b]_t, [c]_t)$ such that $c = a \cdot b$. Usually, a Beaver triple is used to transform one multiplication to two reconstructions. Concretely, given two sharings $[x]_t, [y]_t$, suppose we want to compute $[z]_t$ such that $z = x \cdot y$. Since

$$\begin{aligned} z &= x \cdot y \\ &= (x + a - a) \cdot (y + b - b) \\ &= (x + a) \cdot (y + b) - (x + a) \cdot b - (y + b) \cdot a + a \cdot b, \end{aligned}$$

we can compute

$$[z]_t := (x+a) \cdot (y+b) - (x+a) \cdot [b]_t - (y+b) \cdot [a]_t + [c]_t.$$

Therefore, the task of computing $[z]_t$ becomes to reconstruct two degree-t Shamir sharings $[x]_t + [a]_t$ and $[y]_t + [b]_t$. Observe that, if we set $u = x + a$ and $v = y + b$, the above equation allows us to locally compute a degree-t Shamir sharing of $z := (u - a) \cdot (v - b)$ using a Beaver triple $([a]_t, [b]_t, [c]_t)$ once u and v are publicly known.

Beaver-triple Friendly Form. We say a sharing is in the *Beaver-triple friendly form*, if it can be written as $u - [a]_t$, where u is a public element and $[a]_t$ is a degree-t Shamir sharing. Now suppose for each multiplication gate in the second layer, the input sharings are in the Beaver-triple friendly form, say $u - [a]_t$ and $v - [b]_t$. Given the Beaver triple $([a]_t, [b]_t, [c]_t)$, one can *non-interactively* compute the output sharing of this gate by

$$[z]_t := u \cdot v - u \cdot [b]_t - v \cdot [a]_t + [c]_t.$$

Note that the Beaver triple $([a]_t, [b]_t, [c]_t)$ can be prepared without learning u, v. Therefore, if for each multiplication gate in the second layer, the input sharings are in the Beaver-triple friendly form $u - [a]_t, v - [b]_t$, and $[a]_t, [b]_t$ are learnt *before evaluating the first layer*, we can prepare the Beaver triple $([a]_t, [b]_t, [c]_t)$ without evaluating the first layer, and then non-interactively evaluate multiplication gates in the second layer after learning u, v from the first layer.

Of course, the question remains: since the input sharings of the second layer come from the output sharings of the first layer, how do we ensure that the output sharings of the first layer are in the Beaver-triple friendly form?

Evaluating a Two-Layer Circuit. We observe that the original DN multiplication protocol in [DN07] satisfies our requirement! Concretely, to evaluate a multiplication gate with input sharings $[x]_t, [y]_t$ all parties need to first prepare a pair of random double sharings $([r]_t, [r]_{2t})$. In the last step of the DN multiplication protocol, $P_{\texttt{king}}$ sends the reconstruction result of $[e]_{2t} := [x]_t \cdot [y]_t + [r]_{2t}$ to all parties, and all parties can compute the degree-t Shamir sharing $[z]_t := e - [r]_t$. In particular, the output sharing is in the Beaver-triple friendly form, and the sharing $[r]_t$ is prepared before evaluating this multiplication gate. Therefore, we will use the original DN multiplication protocol to evaluate multiplication gates in the first layer.

For a multiplication gate in the second layer, suppose that the two input wires are both the outputs of multiplication gates in the first layer. Let $e_1 - [r_1]_t$ and $e_2 - [r_2]_t$ denote these two output sharings. Now observe that e_1 and e_2 will already be public as part of evaluating the first layer. So to compute a degree-t Shamir sharing of $(e_1 - r_1)(e_2 - r_2)$, all we need is $[r_1 \cdot r_2]_t$. If we can pre-compute and distribute $([r_1]_t, [r_2]_t, [r_1 \cdot r_2]_t)$, we are done! Of course, since r_1 and r_2 are also used in the multiplication gates in the first layer, we simultaneously need to compute degree-$2t$ Shamir sharings of r_1 and r_2 as well. Fortunately, this does not affect the security of the second layer. In other words, the outputs of the first layer feed nicely into the second layer making the second layer non-interactive. At the same time, we are able to ensure that these two different types of multiplication protocols do not destroy the security of each other despite sharing randomness.

As we discussed above, the input sharing of a multiplication gate in the second layer may come from two other places: (1) it may be an input sharing of this two-layer circuit, or (2) it may be an output sharing of an addition gate in the first layer. In both cases, all parties can locally compute this sharing before evaluating the multiplication gates in the first layer. Let $[x]_t$ denote such an input sharing. Note that $[x]_t = 0 - (-[x]_t)$ is already in the Beaver-triple friendly form. Therefore, all the input sharings of multiplication gates in the second layer are in the Beaver-triple friendly form. But now, the problem is that $[x]_t$ is not known before the circuit evaluation starts (unlike $[r_1]_t$ and $[r_2]_t$), and hence $[x]_t$ cannot be part of a Beaver triple pre-computed before the evaluation. Fortunately, as observed earlier, parties hold $[x]_t$ before evaluating any multiplication gates in the first layer. Now our idea is to prepare the Beaver triples for the second layer dependent on $[x]_t$ *in parallel with* the multiplications in the first layer.

After preparing Beaver triples for the second layer and computing the output sharings of the multiplication gates in the first layer, all parties can locally compute the degree-t Shamir sharings associated with the output wires of this two-layer circuit. These sharings will be fed to the next two-layer circuit, which is sufficient to start the evaluation since the original DN multiplication protocol does not require any special property of the input sharings. Therefore in the evaluation of the whole circuit, these two types of multiplication protocols are alternatively used in every two layers.

Improving the Communication Complexity. While the above helps us make progress, it does not achieve our final goal. In particular, using the original DN protocol requires the communication of 6 elements per party per gate. We note that for multiplications in different layers, we have different requirements:

- For multiplication gates in the first layer, we need the output sharings to have the Beaver-triple friendly form.
- For multiplication gates in the second layer, we compute the Beaver triples in the form of $([a]_t, [b]_t, [c]_t)$. We only need to obtain the degree-t sharing of $[c]_t$ for each Beaver triple.

Therefore for multiplication gates in the second layer, we can use our improved multiplication protocol to compute Beaver triples, which requires the communication of 4 elements per party per multiplication. For multiplication gates in the first layer, however, $P_{\texttt{king}}$ needs to send the same values to all parties. It seems like our trick of using t-wise independence does not work in this scenario.

Having a closer look at our trick of using t-wise independence, for a multiplication gate handled by an honest party, the secret r of the random double sharings is fixed given the double random sharings used for multiplication gates handled by corrupted parties. Revealing the reconstruction result of $[e]_{2t} := [x]_t \cdot [y]_t + [r]_{2t}$ may leak the multiplication result to the adversary. Therefore, to be able to reveal the reconstruction result, r needs to be uniformly random for every multiplication gate. However, we note that r being uniformly random is not equivalent to the pair of double sharings $([r]_t, [r]_{2t})$ being uniformly random.

Therefore, we want to decouple the relation between r and the double sharings. Note that a pair of double sharings $([r]_t, [r]_{2t})$ is equivalent to a pair of sharings $([r]_t, [o]_{2t})$, where the first sharing is a degree-t Shamir sharing of r and the second sharing is a degree-$2t$ Shamir sharing of zero $o = 0$. To see this, given $([r]_t, [r]_{2t})$, we can set $[o]_{2t} := [r]_{2t} - [r]_t$; given $([r]_t, [o]_{2t})$, we can set $[r]_{2t} := [r]_t + [o]_{2t}$. When using a pair of sharings $([r]_t, [o]_{2t})$, the DN multiplication protocol becomes:

1. All parties locally compute $[e]_{2t} := [x]_t \cdot [y]_t + [r]_t + [o]_{2t}$.
2. $P_{\texttt{king}}$ collects all shares of $[e]_{2t}$ and reconstructs the secret e. Then $P_{\texttt{king}}$ sends the value e to all other parties.
3. After receiving e from $P_{\texttt{king}}$, all parties locally compute $[z]_t := e - [r]_t$.

Note that $[o]_{2t}$ is only used to compute $[e]_{2t}$. When $P_{\texttt{king}}$ is an honest party, $[o]_{2t}$ does not need to be a uniformly random degree-$2t$ sharing of 0. Thus, we can use t-wise independent $[o]_{2t}$'s with uniformly random degree-t sharings $[r]_t$'s.

In [DN07], it has been shown that preparing a random degree-t random sharing requires the communication of 2 elements per party. In Section 4.3, following from the same idea of preparing random degree-t Shamir sharings, we show that preparing a random degree-$2t$ sharing of 0 requires the communication of 2 elements per party as well. Then, using our idea of t-wise independence, we expand t random degree-$2t$ sharings of 0 to n sharings with t-wise independence. In this way, the communication cost of preparing correlated-randomness for one multiplication in the first layer is $2 + 2 \cdot t/n \approx 3$ elements. Including the communication cost of the multiplication protocol in [DN07], which is 2 elements per party, the overall cost per multiplication in the first layer is 5 elements per party.

Recall that for multiplication gates in the second layer, we will use our improved multiplication protocol to compute Beaver triples, which requires the communication of 4 elements per party per gate. To evaluate the whole circuit, we first partition it into a sequence of two-layer sub-circuits. Then we use the above strategy to evaluate each two-layer sub-circuit in a predetermined topological order. Assuming that the number of multiplication gates in the first layer is roughly the same as the number of multiplication gates in the second layer, the concrete efficiency is $(4 + 5)/2 = 4.5$ elements per party per gate.

Achieving Security-with-abort. We note that the correctness of the computation requires the following two points:

- $P_{\mathtt{king}}$ parties send the same values to all other parties for multiplication gates in the first layer of all sub-circuits.
- All multiplication tuples are correctly computed.

In the verification phase, all parties first check whether they receive the same values, which corresponds to the first point above. This is done by checking a random linear combination of the values they receive. Then, all parties use the verification of multiplications in [GSZ20] to efficiently check the correctness of all multiplication tuples. In Section 4.3, we show that the communication complexity of the verification phase is sub-linear in the number of multiplication gates. Therefore, the concrete efficiency of our protocol is the same as that for each multiplication gate, i.e., 4.5 elements per party per gate. In particular, comparing with the protocol in [GSZ20], we reduce the number of rounds by a factor of 2.

2.4 Using PRG to Reduce Communication Complexity

We note that the communication complexity can be further reduced by relying on pseudo-random generators. This trick has been used in previous works such as [BBCG+19, LN17, NV18].

At a high-level, each pair of parties will first agree on a random seed, which is unknown to other parties. When some party P_i needs to distribute a degree-t sharing, one can think that P_i first sends random elements to the first t parties as their shares. Then P_i reconstructs the whole sharing using the secret and the first t shares, and distributes the shares to the rest of parties. Relying on the PRG, P_i does not need to send shares to the first t parties. Instead, each of the first t parties and P_i will simply run the PRG on their common seed and take the same piece from the output as the share. In this way, the cost of distributing a degree-t sharing can be reduced by a factor of 2. For a degree-$2t$ sharing, one can think that P_i first sends random elements to all other parties as their shares. Then P_i reconstructs the whole sharing using the secret and the $2t$ shares distributed to other parties. Finally, P_i can compute its own share. Relying on PRG, P_i does not need to communicate with any party. Instead, each party and P_i simply run the PRG on their common seed and take the same piece from the output as the share. In this way, distributing a degree-$2t$ sharing can be done at no cost. Regarding the security, notice that the corrupted parties learn nothing about the secret of a sharing distributed by an honest party even if the shares of corrupted parties are determined by themselves. This is because the corrupted parties only learn t shares of either a degree-t Shamir sharing or a degree-$2t$ Shamir sharing, which are independent of the secret value.

As a result, for our first improvement of using t-wise independence, the concrete efficiency can be improved to 2 elements per party per gate. For our second improvement of using Beaver triples, the communication efficiency can be improved to 2.5 elements per party per gate. More details can be found in Section 4.4.

3 Preliminaries

3.1 Model

In this work, we focus on functions that can be represented as arithmetic circuits over a finite field $\mathbb{F}$ (with $|\mathbb{F}| \geq 2n$)[1] with input, addition, multiplication, and output gates. Let $\phi = \log |\mathbb{F}|$ be the size of an element in $\mathbb{F}$. We use κ to denote the security parameter and let $\mathbb{K}$ be an extension field of $\mathbb{F}$ (with $|\mathbb{K}| \geq 2^{\kappa}$). For simplicity, we assume that κ is the size of an element in $\mathbb{K}$. Let c_I, c_M, c_O be the number of input gates, multiplication gates, and output gates respectively. We set $C = c_I + c_M + c_O$ to be the size of the circuit.

For the secure multi-party computation, we use the *client-server* model. In the client-server model, clients provide inputs to the functionality and receive outputs, and servers can participate in the computation but

[1]The requirement of the field size is due to the use of so-called hyper-invertible matrices in our construction. See more discussion in Section 3.2 of [BTH08].

do not have inputs or get outputs. Each party may have different roles in the computation. Note that, if every party plays a single client and a single server, this corresponds to a protocol in the standard MPC model. Let c denote the number of clients and $n = 2t + 1$ denote the number of servers. For all clients and servers, we assume that every two of them are connected via a secure (private and authentic) synchronous channel so that they can directly send messages to each other. The communication complexity is measured by the number of bits via private channels.

3.1.1 Security Model.

Let c denote the number of clients and $n = 2t + 1$ denote the number of servers. Let $\mathcal{F}$ be a secure function evaluation functionality. An adversary $\mathcal{A}$ can corrupt at most c clients and t servers, provide inputs to corrupted clients, and receive all messages sent to corrupted clients and servers. Corrupted clients and servers can deviate from the protocol arbitrarily.

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

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

We say that a protocol π securely computes $\mathcal{F}$ if there exists a simulator `Sim`, such that for all adversary $\mathcal{A}$, the distribution of the output of the real world execution is *statistically close* to the distribution of the output of the ideal world execution. If π allows a premature abort, then we say π securely computes $\mathcal{F}$ with abort. We refer the readers to [GIP$^+$14] for a formal definition.

3.1.2 Benefits of the Client-Server Model.

In our construction, the clients only participate in the input phase and the output phase. The main computation is conducted by the servers. For simplicity, we use $\{P_1, \ldots, P_n\}$ to denote the n servers, and refer to the servers as parties. Let $\mathcal{C}$ denote the set of all corrupted parties and $\mathcal{H}$ denote the set of all honest parties. One benefit of the client-server model is the following theorem shown in [GIP$^+$14].

Theorem 1 (Lemma 5.2 [GIP$^+$14])**.** *Let Π be a protocol computing a c-client circuit C using $n = 2t + 1$ parties. Then, if Π is secure against any adversary controlling exactly t parties, then Π is secure against any adversary controlling at most t parties.*

This theorem allows us to only consider the case where the adversary controls exactly t parties. Therefore in the following, we assume that there are exactly t corrupted parties.

3.2 Secret Sharing

In this work, we will use the standard Shamir Secret Sharing Scheme [Sha79]. Let n be the number of parties and $\mathbb{F}$ be a finite field of size $|\mathbb{F}| \geq n + 1$. Let $\alpha_1, \ldots, \alpha_n$ be n distinct non-zero elements in $\mathbb{F}$.

A *degree-d* Shamir sharing of $x \in \mathbb{F}$ is a vector $(x_1, \ldots, x_n)$ which satisfies that there exists a polynomial $f(\cdot) \in \mathbb{F}[X]$ of degree at most d such that $f(0) = x$ and $f(\alpha_i) = x_i$ for $i \in \{1, \ldots, n\}$. Each party P_i holds a share x_i and the whole sharing is denoted by $[x]_d$.

Properties of the Shamir Secret Sharing Scheme. In the following, we will utilize two properties of the Shamir secret sharing scheme.

- Linear Homomorphism:
$$\forall\ [x]_d, [y]_d,\ [x+y]_d = [x]_d + [y]_d.$$
- Multiplying two degree-d sharings yields a degree-$2d$ sharing. The secret value of the new sharing is the product of the original two secrets.
$$\forall\ [x]_d, [y]_d,\ [x \cdot y]_{2d} = [x]_d \cdot [y]_d.$$

For the first property, we equivalently add the underlying two polynomials. Therefore, the degree remains the same and the secret value becomes the summation of the original two secrets. For the second property, we equivalently multiply the underlying two polynomials. As a result, the degree becomes $2d$ and the secret value is the product of the original two secrets.

3.3 Useful Building Blocks

In this part, we will introduce three functionalities which will be used in our main construction.

Preparing Random Degree-t Shamir Sharings. The first functionality $\mathcal{F}_{\text{rand}}$ allows all parties to prepare a random degree-t Shamir sharing $[r]_t$. The description of $\mathcal{F}_{\text{rand}}$ appears in Functionality 1. An instantiation of $\mathcal{F}_{\text{rand}}$ can be found in [DN07, GSZ20] (Protocol 2 in Section 3.3 of [GS20]). At a high-level, the idea is to let each party generate and distribute a random degree-t Shamir sharing to all parties. Then, all parties locally apply (the transpose of) a Vandermonde matrix, as a randomness extractor, on their shares to obtain $n-t$ random degree-t Shamir sharings. The amortized communication cost per sharing is 2 elements per party.

Functionality 1: $\mathcal{F}_{\text{rand}}$

1. $\mathcal{F}_{\text{rand}}$ receives from the adversary the set of shares $\{r_i\}_{i\in\mathcal{C}}$.
2. $\mathcal{F}_{\text{rand}}$ randomly samples r. Based on the secret r and the t shares $\{r_i\}_{i\in\mathcal{C}}$ of corrupted parties, $\mathcal{F}_{\text{rand}}$ reconstructs the whole sharing $[r]_t$ and distributes the shares of $[r]_t$ to honest parties.

Preparing Random Double Sharings. The second functionality $\mathcal{F}_{\text{doubleRand}}$ allows all parties to prepare a pair of random double sharings $([r]_t, [r]_{2t})$. The description of $\mathcal{F}_{\text{doubleRand}}$ appears in Functionality 2. An instantiation of $\mathcal{F}_{\text{doubleRand}}$ can be found in [DN07, GSZ20] (Protocol 4 in Section 3.4 of [GS20]). At a high-level, the idea is to let each party generate and distribute a pair of random double sharings to all parties. Then, all parties locally apply (the transpose of) a Vandermonde matrix, as a randomness extractor, on their shares to obtain $n-t$ pairs of random double sharings. The amortized communication cost per pair of random double sharings is 4 elements per party.

Generating Random Coin. The third functionality $\mathcal{F}_{\text{coin}}$ allows all parties to generate a random field element in $\mathbb{K}$. Recall that $\mathbb{K}$ is an extension field of $\mathbb{F}$ such that $|\mathbb{K}| \geq 2^\kappa$, where κ is the security parameter. The description of $\mathcal{F}_{\text{coin}}$ appears in Functionality 3. An instantiation of $\mathcal{F}_{\text{coin}}$ can be found in [GSZ20] (Protocol 6 in Section 3.5 of [GS20]). At a high-level, the idea is to first invoke $\mathcal{F}_{\text{rand}}$ to obtain a random degree-t Shamir sharing. Then all parties exchange their shares and reconstruct the secret as their output, which is a random field element. The communication complexity of the instantiation is $O(n^2\kappa)$ bits.

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

1. $\mathcal{F}_{\text{doubleRand}}$ receives from the adversary two sets of shares $\{r_i\}_{i\in\mathcal{C}}$ and $\{r'_i\}_{i\in\mathcal{C}}$. $\mathcal{F}_{\text{doubleRand}}$ view the first set as the shares of corrupted party for the degree-t sharing, and the second set as the shares for the degree-$2t$ sharing.

2. $\mathcal{F}_{\text{doubleRand}}$ randomly samples r and prepares the double sharings as follows.
   - For the degree-t sharing, based on the secret r and the t shares $\{r_i\}_{i\in\mathcal{C}}$ of corrupted parties, $\mathcal{F}_{\text{doubleRand}}$ reconstructs the whole sharing $[r]_t$.
   - For the degree-$2t$ sharing, $\mathcal{F}_{\text{doubleRand}}$ randomly samples t elements as the shares of the first t honest parties. Based on the secret r, the t shares of the first t honest parties, and the t shares $\{r'_i\}_{i\in\mathcal{C}}$ of corrupted parties, $\mathcal{F}_{\text{doubleRand}}$ reconstructs the whole sharing $[r]_{2t}$.

   Finally, $\mathcal{F}_{\text{doubleRand}}$ distributes the shares of $([r]_t, [r]_{2t})$ to honest parties.

Functionality 3: $\mathcal{F}_{\text{coin}}$

1. $\mathcal{F}_{\text{coin}}$ samples a random field element r in $\mathbb{K}$.

2. $\mathcal{F}_{\text{coin}}$ sends r to the adversary.
   - If the adversary replies `continue`, $\mathcal{F}_{\text{coin}}$ sends r to honest parties.
   - If the adversary replies `abort`, $\mathcal{F}_{\text{coin}}$ sends `abort` to honest parties.

4 ATLAS: Our Unconditional MPC Construction

In this section, we will introduce two improvements to the secure-with-abort MPC protocol in [GSZ20].

- The first improvement reduces the communication cost per multiplication gate per party from 5.5 elements to 4 elements.

- The second improvement reduces the communication cost per multiplication gate per party from 5.5 elements to 4.5 elements *and reduce the number of rounds by a factor of* 2.

Our core idea is to reuse the correlated-randomness prepared for multiplication gates.

We first give a short review of the construction in [GSZ20]. Then we introduce our two improvements. Finally, we show how to use a pseudo-random generator to further reduce the communication complexity.

4.1 Review of the Secure-with-abort MPC Protocol in [GSZ20]

In [GIP$^+$14], Genkin et al. showed that several semi-honest MPC protocols are secure up to an additive attack in the presence of a fully malicious adversary. An additive attack means that the adversary is able to change the multiplication result by adding an arbitrary fixed value. As one corollary, these semi-honest protocols provide full privacy of honest parties before reconstructing the output. Therefore, a straightforward strategy to achieve security-with-abort is to (1) run a semi-honest protocol till the output phase, (2) check the correctness of the computation, and (3) reconstruct the output only if the check passes.

Fortunately, the best-known semi-honest protocol in this setting [DN07] is secure up to an additive attack. At a high-level, the semi-honest protocol in [DN07] computes a degree-t Shamir sharing for each wire. Since the Shamir secret sharing scheme is linear homomorphic, addition gates can be evaluated without interaction. Therefore, the main concern is multiplication gates. In [GSZ20], this kind of attack is modeled in the functionality $\mathcal{F}_{\text{mult}}$, which takes two degree-t Shamir sharings $[x]_t, [y]_t$ and outputs the multiplication result $[x \cdot y]_t$. The description of $\mathcal{F}_{\text{mult}}$ can be found in Functionality 4. The original multiplication protocol in [DN07] requires 6 elements per party per gate. Goyal et al. [GSZ20] improve this protocol and reduce the communication cost to 5.5 elements.

Functionality 4: $\mathcal{F}_{\text{mult}}$

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

Since $\mathcal{F}_{\text{mult}}$ does not guarantee the correctness of the multiplications, all parties need to verify the multiplications computed by $\mathcal{F}_{\text{mult}}$ at the end of the protocol. The functionality $\mathcal{F}_{\text{multVerify}}$ takes N multiplication tuples as input and outputs to all parties a single bit b indicating whether all multiplication tuples are correct. The description of $\mathcal{F}_{\text{multVerify}}$ can be found in Functionality 5.

Functionality 5: $\mathcal{F}_{\text{multVerify}}$

1. Let N denote the number of multiplication tuples. The multiplication tuples are denoted by
$$([x^{(1)}]_t, [y^{(1)}]_t, [z^{(1)}]_t), ([x^{(2)}]_t, [y^{(2)}]_t, [z^{(2)}]_t), \ldots, ([x^{(N)}]_t . [y^{(N)}]_t, [z^{(N)}]_t).$$
2. For all $i \in [N]$, $\mathcal{F}_{\text{multVerify}}$ receives from honest parties their shares of $[x^{(i)}]_t, [y^{(i)}]_t, [z^{(i)}]_t$. Then $\mathcal{F}_{\text{multVerify}}$ reconstructs the secrets $x^{(i)}, y^{(i)}, z^{(i)}$. $\mathcal{F}_{\text{multVerify}}$ further computes the shares of $[x^{(i)}]_t, [y^{(i)}]_t, [z^{(i)}]_t$ held by corrupted parties and sends these shares to the adversary.
3. For all $i \in [N]$, $\mathcal{F}_{\text{multVerify}}$ computes $d^{(i)} = z^{(i)} - x^{(i)} \cdot y^{(i)}$ and sends $d^{(i)}$ to the adversary.
4. Finally, let $b \in \{\texttt{abort}, \texttt{accept}\}$ denote whether there exists $i \in [N]$ such that $d^{(i)} \neq 0$. $\mathcal{F}_{\text{multVerify}}$ sends b to the adversary and waits for its response.
   - If the adversary replies `continue`, $\mathcal{F}_{\text{multVerify}}$ sends b to honest parties.
   - If the adversary replies `abort`, $\mathcal{F}_{\text{multVerify}}$ sends `abort` to honest parties.

In [GSZ20], Goyal et al. provide an instantiation of $\mathcal{F}_{\text{multVerify}}$ which has communication complexity $O(n^2 \cdot \log C \cdot \kappa)$ bits, where n is the number of parties and κ is the security parameter. Note that it is sub-linear in the number of multiplication tuples. Relying on $\mathcal{F}_{\text{mult}}, \mathcal{F}_{\text{multVerify}}$, Goyal et al. [GSZ20] construct a secure-with-abort MPC protocol with communication complexity $O(Cn\phi + n^2 \cdot \log C \cdot \kappa)$ bits. In particular, the concrete efficiency per multiplication gate is the same as the communication cost of the instantiation of $\mathcal{F}_{\text{mult}}$, i.e., 5.5 elements per party.

4.2 Reducing the Communication Complexity via t-wise Independence

Our first improvement comes from a new protocol for $\mathcal{F}_{\text{mult}}$. The amortized communication cost of our new protocol is 4 elements per party. Relying on the secure-with-abort MPC protocol [GSZ20] which uses $\mathcal{F}_{\text{mult}}, \mathcal{F}_{\text{multVerify}}$ as building blocks, we directly obtain a secure-with-abort MPC protocol with the same asymptotic communication complexity, i.e., $O(Cn\phi + n^2 \cdot \log C \cdot \kappa)$ bits. In particular, the concrete efficiency per multiplication gate is 4 elements per party. Our new protocol is based on the multiplication protocol in [DN07]. We first give a quick review of the multiplication protocol in [DN07].

4.2.1 Review of the Multiplication Protocol in [DN07].

To evaluate a multiplication gate, all parties need to prepare a pair of random double sharings $([r]_t, [r]_{2t})$. This is done by invoking $\mathcal{F}_{\text{doubleRand}}$ introduced in Section 3.3. Recall that the amortized communication complexity of the instanciation of $\mathcal{F}_{\text{doubleRand}}$ in [DN07, GSZ20] is 4 elements per party.

For a multiplication gate, suppose the input sharings are denoted by $[x]_t, [y]_t$. To compute $[z]_t := [x \cdot y]_t$, a pair of random double sharings $([r]_t, [r]_{2t})$ is consumed. All parties first agree on a special party $P_{\texttt{king}}$. $P_{\texttt{king}}$ will help do the reconstruction in the multiplication protocol. Then, all parties run the following steps:

1. All parties locally compute $[e]_{2t} := [x]_t \cdot [y]_t + [r]_{2t}$.
2. $P_{\texttt{king}}$ collects all shares of $[e]_{2t}$ and reconstructs the secret e. Then $P_{\texttt{king}}$ sends the value e to all other parties.
3. After receiving e from $P_{\texttt{king}}$, all parties locally compute $[z]_t := e - [r]_t$.

The correctness follows from the properties of the Shamir secret sharing scheme. Note that each party needs to send an element to $P_{\texttt{king}}$, and $P_{\texttt{king}}$ needs to send an element to each party. The communication complexity of this protocol is 2 elements per party. Including the communication cost for preparing double sharings, the overall cost per multiplication gate is 6 elements per party.

In [GSZ20], Goyal et al. observe that in the second step, $P_{\texttt{king}}$ can alternatively distribute a degree-t Shamir sharing $[e]_t$. Then in the last step, all parties can still compute $[z]_t := [e]_t - [r]_t$. Furthermore, since e does not need to be private, $P_{\texttt{king}}$ can set the shares of (a predetermined set of) t parties to be 0 in $[e]_t$. This means that $P_{\texttt{king}}$ need not to communication these shares at all, reducing the communication by half. This observation allows Goyal et al. to reduce the communication cost from 6 elements to 5.5 elements.

4.2.2 Our Observation.

As [GSZ20], we require $P_{\texttt{king}}$ to distribute a degree-t Shamir sharing $[e]_t$ in the second step. However, we further require $P_{\texttt{king}}$ to generate a random sharing $[e]_t$. In this way, when $P_{\texttt{king}}$ is an honest party, corrupted parties only receive t shares of a random degree-t sharing $[e]_t$ from $P_{\texttt{king}}$, which are uniform and independent of the secret. As discussed in Section 2, it means that we do not need to use uniform double sharings when $P_{\texttt{king}}$ is honest.

For n multiplication gates, our idea is to let each party behave as $P_{\texttt{king}}$ for one multiplication gate. Note that only t out of n multiplications are handled by corrupted $P_{\texttt{king}}$'s. To make sure that all parties still use a pair of random double sharings when $P_{\texttt{king}}$ is corrupted, the n pairs of double sharings for these n multiplication gates only need to be t-wise independent. To this end, we will first generate t pairs of random double sharings, and then expand them to n pairs of double sharings with t-wise independence.

Specifically, all parties agree on an $n \times t$ hyper-invertible matrix $\boldsymbol{M}$. Let $([r^{(1)}]_t, [r^{(1)}]_{2t}), \dots, ([r^{(t)}]_t, [r^{(t)}]_{2t})$ be t pairs of random double sharings prepared by $\mathcal{F}_{\text{doubleRand}}$. All parties execute Expand (Protocol 6) to expand these t pairs into n pairs of t-wise independent double sharings.

Recall that $\mathcal{C}$ denotes the set of all corrupted parties. By the property of hyper-invertible matrices, there is a one-to-one map from $\{([\tilde{r}^{(i)}]_t, [\tilde{r}^{(i)}]_{2t})\}_{i \in \mathcal{C}}$ to $\{[r^{(i)}]_t, [r^{(i)}]_{2t}\}_{i=1}^{t}$. Thus, $\{([\tilde{r}^{(i)}]_t, [\tilde{r}^{(i)}]_{2t})\}_{i \in \mathcal{C}}$ are t pairs of random double sharings.

Protocol 6: Expand

1. All parties agree on an $n \times t$ hyper-invertible matrix $\boldsymbol{M}$. All parties locally compute

$$\begin{aligned}([\tilde{r}^{(1)}]_t, \ldots, [\tilde{r}^{(n)}]_t)^{\mathrm{T}} &= \boldsymbol{M}([r^{(1)}]_t, \ldots, [r^{(t)}]_t)^{\mathrm{T}} \\ ([\tilde{r}^{(1)}]_{2t}, \ldots, [\tilde{r}^{(n)}]_{2t})^{\mathrm{T}} &= \boldsymbol{M}([r^{(1)}]_{2t}, \ldots, [r^{(t)}]_{2t})^{\mathrm{T}}\end{aligned}$$

2. All parties output $\{([\tilde{r}^{(i)}]_t, [\tilde{r}^{(i)}]_{2t}, P_i)\}_{i=1}^n$, where $([\tilde{r}^{(i)}]_t, [\tilde{r}^{(i)}]_{2t}, P_i)$ will be used for a multiplication gate handled by P_i.

4.2.3 ATLAS Multiplication Protocol.

To evaluate a multiplication gate, a pair of double sharings $([r]_t, [r]_{2t}, P_i)$ is consumed. All parties execute Mult (Protocol 7).

Protocol 7: Mult

1. Let $([r]_t, [r]_{2t}, P_i)$ be the random double sharings which will be used in the protocol. Let $[x]_t, [y]_t$ denote the input sharings.
2. All parties locally compute $[e]_{2t} = [x]_t \cdot [y]_t + [r]_{2t}$.
3. P_i collects all shares and reconstructs the secret $e = x \cdot y + r$. Then P_i randomly generates a degree-t Shamir sharing $[e]_t$ and distributes the shares to other parties.
4. All parties locally compute $[z]_t = [e]_t - [r]_t$.

To show the security of ATLAS multiplication protocol, we consider the scenario where all parties evaluate *a sequence of* N multiplication gates. In particular, the input sharings of each multiplication gate can depend on the input sharings or output sharings of the previous multiplication gates. The functionality $\mathcal{F}'_{\text{mult}}$ appears in Functionality 8, which invokes $\mathcal{F}_{\text{mult}}$ for each multiplication gate. One can view $\mathcal{F}'_{\text{mult}}$ as an interface of $\mathcal{F}_{\text{mult}}$. It allows us to replace the invocation of $\mathcal{F}_{\text{mult}}$ in the secure-with-abort MPC protocol [GSZ20] by the invocation of $\mathcal{F}'_{\text{mult}}$, and thus directly use ATLAS multiplication protocol in the protocol [GSZ20]. The protocol ATLAS-Mult appears in Protocol 9.

Functionality 8: $\mathcal{F}'_{\text{mult}}$

1. $\mathcal{F}'_{\text{mult}}$ receives N from all parties.
2. From $i = 1$ to N, let $[x^{(i)}]_t, [y^{(i)}]_t$ denote the input sharings of the i-th multiplication gate. $\mathcal{F}'_{\text{mult}}$ invokes $\mathcal{F}_{\text{mult}}$ on $[x^{(i)}]_t, [y^{(i)}]_t$.

Lemma 1. *The protocol* ATLAS-Mult *securely computes the functionality* $\mathcal{F}'_{mult}$ *in the* $\mathcal{F}_{doubleRand}$*-hybrid model in the presence of a fully malicious adversary controlling* t *corrupted parties.*

Protocol 9: ATLAS-Mult

1. All parties set N to be the number of multiplication gates to be evaluated.
2. All parties invoke $\mathcal{F}_{\text{doubleRand}}$ to prepare $N \cdot t/n$ pairs of random double sharings, and invoke Expand to obtain N pairs of double sharings in the form of $([r]_t, [r]_{2t}, P_j)$
3. From $i = 1$ to N, let $[x^{(i)}]_t, [y^{(i)}]_t$ denote the input sharings of the i-th multiplication gate. Suppose $([r]_t, [r]_{2t}, P_j)$ is the first pair of unused double sharings. All parties invoke Mult on $[x^{(i)}]_t, [y^{(i)}]_t$ and $([r]_t, [r]_{2t}, P_j)$.

Proof. Recall that $\mathcal{C}$ denotes the set of corrupted parties and $\mathcal{H}$ denotes the set of honest parties. We first show that the random degree-$2t$ sharing $[r]_{2t}$ output by $\mathcal{F}_{\text{doubleRand}}$ satisfies that the shares of honest parties are uniformly random, and are independent of the shares chosen by the adversary. Recall that $\mathcal{F}_{\text{doubleRand}}$ receives from the adversary two sets of shares $\{r_i\}_{i\in\mathcal{C}}, \{r'_i\}_{i\in\mathcal{C}}$ and randomly samples $([r]_t, [r]_{2t})$ such that the shares of $[r]_t, [r]_{2t}$ held by corrupted parties are $\{r_i\}_{i\in\mathcal{C}}, \{r'_i\}_{i\in\mathcal{C}}$ respectively. Consider the following sampling process:

1. $\mathcal{F}_{\text{doubleRand}}$ randomly samples $t+1$ shares as the shares of $[r]_{2t}$ held by honest parties. Then, $\mathcal{F}_{\text{doubleRand}}$ reconstructs the whole sharing $[r]_{2t}$ using the shares of honest parties and the shares $\{r'_i\}_{\mathcal{C}}$ of corrupted parties, and computes the secret r.
2. $\mathcal{F}_{\text{doubleRand}}$ reconstructs the whole sharing $[r]_t$ based on the secret r and the shares $\{r_i\}_{i\in\mathcal{C}}$ of corrupted parties.

Note that the above process output a pair of random double sharings with the same distribution as that described in the original $\mathcal{F}_{\text{doubleRand}}$. However, the shares of $[r]_{2t}$ held by honest parties are randomly chosen in the first step and are independent of the shares $\{r_i\}_{i\in\mathcal{C}}, \{r'_i\}_{i\in\mathcal{C}}$.

Recall that in Expand, all parties compute

$$\begin{aligned}([\tilde{r}^{(1)}]_t, \ldots, [\tilde{r}^{(n)}]_t)^{\mathrm{T}} &= \boldsymbol{M}([r^{(1)}]_t, \ldots, [r^{(t)}]_t)^{\mathrm{T}} \\ ([\tilde{r}^{(1)}]_{2t}, \ldots, [\tilde{r}^{(n)}]_{2t})^{\mathrm{T}} &= \boldsymbol{M}([r^{(1)}]_{2t}, \ldots, [r^{(t)}]_{2t})^{\mathrm{T}},\end{aligned}$$

where $\boldsymbol{M}$ is a hyper-invertible matrix. From the above argument, the shares of $\{[r^{(i)}]_{2t}\}_{i=1}^t$ held by honest parties are uniformly random. According to the property of hyper-invertible matrices, there is a one-to-one map from $\{[r^{(i)}]_{2t}\}_{i=1}^t$ to $\{[\tilde{r}^{(i)}]_{2t}\}_{i\in\mathcal{C}}$. Thus, the shares of $\{[\tilde{r}^{(i)}]_{2t}\}_{i\in\mathcal{C}}$ held by honest parties are uniformly random. This means that for all double sharings in the form of $([r]_t, [r]_{2t}, P_j)$ output by Expand, where P_j is a corrupted party, the shares of $[r]_{2t}$ held by honest parties are uniformly random.

Let $\mathcal{A}$ denote the adversary. We will construct a simulator $\mathcal{S}$ to simulate the behaviors of honest parties.

Simulation for ATLAS-Mult. In the first step, $\mathcal{S}$ emulates $\mathcal{F}_{\text{doubleRand}}$ and receives the shares of random double sharings held by corrupted parties. Then $\mathcal{S}$ follows Expand and computes the shares of the double sharings in the form of $([r]_t, [r]_{2t}, P_j)$ held by corrupted parties.

In the second step, we describe the strategy of $\mathcal{S}$ for each invocation of Mult.

1. Let $[x^{(i)}]_t, [y^{(i)}]_t$ denote the input sharings. $\mathcal{S}$ receives from $\mathcal{F}'_{\text{mult}}$ the shares of $[x^{(i)}]_t, [y^{(i)}]_t$ held by corrupted parties. Let $([r]_t, [r]_{2t}, P_j)$ be the first pair of unused double sharings. Recall that $\mathcal{S}$ has learnt the shares of $[r]_t, [r]_{2t}$ held by corrupted parties.
2. $\mathcal{S}$ computes the shares of $[e^{(i)}]_{2t} := [x^{(i)}]_t \cdot [y^{(i)}]_t + [r]_{2t}$ held by corrupted parties. If P_j is corrupted, $\mathcal{S}$ samples random elements as shares of $[e^{(i)}]_{2t}$ of honest parties.

3. Depending on whether P_j is honest, there are two cases:
   - If P_j is honest, $\mathcal{S}$ receives the shares from corrupted parties. Let $[\tilde{e}^{(i)}]_{2t}$ denote the sharing when using the shares received from corrupted parties. This is to distinguish from $[e^{(i)}]_{2t}$ which uses the shares computed by $\mathcal{S}$ for corrupted parties. Then $\mathcal{S}$ computes the shares of $[d^{(i)}]_{2t} := [\tilde{e}^{(i)}]_{2t} - [e^{(i)}]_{2t}$ held by corrupted parties. Note that the shares of $[d^{(i)}]_{2t}$ held by honest parties are 0. $\mathcal{S}$ reconstructs the secret d. Finally for $[e^{(i)}]_t$, $\mathcal{S}$ samples random elements as the shares of corrupted parties.
   - If P_j is corrupted, $\mathcal{S}$ sends the shares of $[e^{(i)}]_{2t}$ of honest parties to P_j. Note that for $[e^{(i)}]_{2t}$, $\mathcal{S}$ knows all the shares. $\mathcal{S}$ reconstructs the secret e. Then $\mathcal{S}$ receives the shares of $[e^{(i)}]_t$ of honest parties from P_j. $\mathcal{S}$ reconstructs the whole sharing and computes the secret of $[e^{(i)}]_t$, denoted by $\tilde{e}^{(i)}$. Finally, $\mathcal{S}$ computes $d^{(i)} := \tilde{e}^{(i)} - e^{(i)}$.
4. In the last step, $\mathcal{S}$ computes the shares of $[z^{(i)}]_t := [e^{(i)}]_t - [\tilde{r}^{(i)}]_t$ held by corrupted parties. Then $\mathcal{S}$ sends the difference $d^{(i)}$ and the shares of $[z^{(i)}]_t$ held by corrupted parties to $\mathcal{F}'_{\mathrm{mult}}$.

Hybrid Arguments. Now we show that $\mathcal{S}$ perfectly simulates the behaviors of honest parties. Consider the following hybrids.

$\mathbf{Hybrid}_0$: The execution in the real world.

$\mathbf{Hybrid}_1$: In this hybrid, $\mathcal{S}$ computes the shares of corrupted parties as described above. For each multiplication tuple $([x^{(i)}]_t, [y^{(i)}]_t, [z^{(i)}]_t)$, $\mathcal{S}$ computes the difference $d^{(i)} := z^{(i)} - x^{(i)} \cdot y^{(i)}$ using the shares of honest parties. Then for all $i \in [N]$, $\mathcal{S}$ sends the difference $d^{(i)}$ and the shares of $[z^{(i)}]_t$ held by corrupted parties to $\mathcal{F}'_{\mathrm{mult}}$. Note that the shares of honest parties are determined by the shares of corrupted parties and the secrets. Therefore, the distribution of $\mathbf{Hybrid}_1$ is identical to the distribution of $\mathbf{Hybrid}_0$.

$\mathbf{Hybrid}_2$: In this hybrid, $\mathcal{S}$ computes the difference as described above. Note that:

- When P_j is honest, corrupted parties can change the multiplication result by sending incorrect shares of $[e^{(i)}]_{2t}$ to P_j. Therefore, the difference of this multiplication tuple is the secret of the sharing $[\tilde{e}^{(i)}]_{2t} - [e^{(i)}]_{2t}$, where $[e^{(i)}]_{2t}$ is the sharing when using the shares computed by $\mathcal{S}$ for corrupted parties, and $[e]_{2t}$ is the sharing when using the shares received from corrupted parties. Therefore, the difference $d^{(i)}$ computed by $\mathcal{S}$ is identical to that in $\mathbf{Hybrid}_1$.
- When P_j is corrupted, $\mathcal{S}$ can learn the value $\tilde{e}^{(i)}$ reconstructed by P_j from the shares of $[e^{(i)}]_t$ held by honest parties. $\mathcal{S}$ can also compute the correct $e^{(i)}$. Therefore, the difference $d^{(i)}$ computed by $\mathcal{S}$ is identical to that in $\mathbf{Hybrid}_1$.

Therefore, the distribution of $\mathbf{Hybrid}_2$ is identical to the distribution of $\mathbf{Hybrid}_1$.

$\mathbf{Hybrid}_3$: In this hybrid, $\mathcal{S}$ uses random elements as shares of $[e^{(i)}]_{2t}$ of honest parties when P_j is corrupted. Recall that we have shown that for $([r]_t, [r]_{2t}, P_j)$ where P_j is corrupted, the shares of $[r]_{2t}$ held by honest parties are uniformly random. Therefore, the shares of $[e^{(i)}]_{2t}$ are also uniformly random in this case. The distribution of $\mathbf{Hybrid}_3$ is identical to the distribution of $\mathbf{Hybrid}_2$.

$\mathbf{Hybrid}_4$: In this hybrid, $\mathcal{S}$ emulates $\mathcal{F}_{\mathrm{doubleRand}}$ and does not generate the shares of honest parties. Note that these shares are not used in $\mathbf{Hybrid}_3$. Therefore, the distribution of $\mathbf{Hybrid}_4$ is identical to the distribution of $\mathbf{Hybrid}_3$.

Note that $\mathbf{Hybrid}_4$ is the execution in the ideal world, and the distribution of $\mathbf{Hybrid}_4$ is identical to the distribution of $\mathbf{Hybrid}_0$, the execution in the real world. □

Using $\mathcal{F}'_{\mathbf{mult}}$ in the MPC protocol in [GSZ20]. In the secure-with-abort MPC protocol in [GSZ20], all parties invoke $\mathcal{F}_{\mathrm{mult}}$ for each multiplication gate. Note that $\mathcal{F}'_{\mathrm{mult}}$ invoke $\mathcal{F}_{\mathrm{mult}}$ for each multiplication. Therefore, we view $\mathcal{F}'_{\mathrm{mult}}$ as an interface of $\mathcal{F}_{\mathrm{mult}}$. All parties initialize $\mathcal{F}'_{\mathrm{mult}}$ in the beginning of the protocol with the number of multiplications they need to compute (which is determined by the circuit). Then we replace each invocation of $\mathcal{F}_{\mathrm{mult}}$ by $\mathcal{F}'_{\mathrm{mult}}$.

Note that every t pairs of random double sharings generated by $\mathcal{F}_{\text{doubleRand}}$ are expanded to n pairs of double sharings. Therefore, the communication cost per pair of double sharings is $4 \cdot t/n \approx 2$ elements per party. The overall cost per multiplication gate is 4 elements per party. Therefore, when using ATLAS-Mult to instantiate $\mathcal{F}'_{\text{mult}}$, we obtain a secure-with-abort MPC protocol with communication complexity of $O(Cn\phi + n^2 \cdot \log C \cdot \kappa)$ bits. In particular, the concrete efficiency per multiplication gate is 4 elements per party.

Remark 1. *It has been observed in many previous works (e.g., [CGH+18, GSZ20]) that the DN multiplication protocol can be extended to compute an inner-product operation* with the same communication complexity as a multiplication operation. *An inner-product operation is to compute the summation of the coordinate-wise multiplications between two vectors. At a high-level, given two vectors of input sharings* $([x^{(1)}]_t, [x^{(2)}]_t, \ldots, [x^{(\ell)}]_t), ([y^{(1)}]_t, [y^{(2)}]_t, \ldots, [y^{(\ell)}]_t)$*, the goal is to compute a degree-t Shamir sharing of* $z = \sum_{i=1}^{\ell} x^{(i)} \cdot y^{(i)}$*. Since all parties can locally compute a degree-2t Shamir sharing* $[z]_{2t} = \sum_{i=1}^{\ell} [x^{(i)}]_t \cdot [y^{(i)}]_t$*, all parties can use the same technique as the DN multiplication protocol to do degree reduction.*

We note that our technique of using t-wise independent double sharings also works in this extension. As a result, we obtain an inner-product protocol with communication complexity of 4 *elements per party, which is secure up to an additive attack (see Functionality 7 in Section 4 of [GS20] for the description of the corresponding functionality).*

4.3 Reducing the Number of Rounds via Beaver Triples

For the secure-with-abort MPC protocol in [GSZ20], multiplication gates in the same layer of the circuit are evaluated in parallel. Therefore, the number of rounds is linear in the depth of the circuit. To further improve the concrete efficiency, we pay our attention to the round complexity. In this part, we show that multiplication gates in a two-layer circuit can be evaluated in parallel. It allows us to reduce the number of rounds by a factor of 2. The amortized communication cost per multiplication gate is 4.5 elements per party.

4.3.1 An Overview of Our Approach.

We first start with a two-layer circuit. At a high-level, we use Beaver triples to evaluate multiplications in the second layer. Recall that a Beaver triple consists of three degree-t Shamir sharings $([a]_t, [b]_t, [c]_t)$ such that $c = a \cdot b$. Usually, a Beaver triple is used to transform one multiplication to two reconstructions. Concretely, given two sharings $[x]_t, [y]_t$, suppose we want to compute $[z]_t$ such that $z = x \cdot y$. Since

$$\begin{aligned} z &= x \cdot y = (x + a - a) \cdot (y + b - b) \\ &= (x + a) \cdot (y + b) - (x + a) \cdot b - (y + b) \cdot a + a \cdot b, \end{aligned}$$

we can compute

$$[z]_t := (x + a) \cdot (y + b) - (x + a) \cdot [b]_t - (y + b) \cdot [a]_t + [c]_t.$$

Therefore, the task of computing $[z]_t$ becomes to reconstruct two degree-t Shamir sharings $[x]_t + [a]_t$ and $[y]_t + [b]_t$. Observe that, if we set $u = x + a$ and $v = y + b$, the above equation allows us to locally compute a degree-t Shamir sharing of $z := (u - a) \cdot (v - b)$ using a Beaver triple $([a]_t, [b]_t, [c]_t)$. In particular, the values u, v *can be learnt after* preparing the Beaver triple. For multiplications in the second layer, our idea is to transform each input sharing to the form of $u - [a]_t$, where u is a public element and $[a]_t$ is a degree-t Shamir sharing. We refer to this form as the *Beaver-triple friendly form.* Moreover, the sharing $[a]_t$ is known to all parties *before evaluating the first layer.* In this way, for an multiplication gate in the second layer with input sharings $u - [a]_t$ and $v - [b]_t$, we can prepare the Beaver triple $([a]_t, [b]_t, [c]_t)$ *in parallel with* the multiplications in the first layer.

We note that an input sharing of a multiplication gate in the second layer may come from three places:

- This sharing is an input sharing of the circuit.

- This sharing is an output sharing of an addition gate in the first layer.
- This sharing is an output sharing of a multiplication gate in the first layer.

Note that an addition gate can be evaluated without interaction. For the first two cases, all parties can locally compute this sharing. Let $[x]_t$ denote such a sharing. Note that $[x]_t = 0 - (-[x]_t)$ is already in the Beaver-triple friendly form, and $(-[x]_t)$ is known before evaluating the first layer. For the third case, we want the output sharing of a multiplication gate in the first layer to have the Beaver-triple friendly form $u - [a]_t$, and $[a]_t$ is known before evaluating this gate. We note that the original multiplication protocol in [DN07] satisfies our requirement. Recall that in the original multiplication protocol in [DN07]:

1. $P_{\texttt{king}}$ reconstructs a degree-$2t$ Shamir sharing $[e]_{2t} := [x]_t \cdot [y]_t + [r]_{2t}$ and sends e to other parties.
2. All parties locally compute $[z]_t := e - [r]_t$.

In particular, the random double sharings $([r]_t, [r]_{2t})$ are prepared before evaluating this gate.

In summary, a two-layer circuit can be evaluated as follows:

- For each input sharing in the second layer, all parties transform it to the Beaver-triple friendly form, denoted by $u - [a]_t$, such that $[a]_t$ is known to all parties.
- For each multiplication gate in the first layer, suppose $[x]_t, [y]_t$ are the input sharings. All parties use the original multiplication protocol in [DN07] to compute $[z]_t$, where $z = x \cdot y$. For each multiplication gate in the second layer, suppose $u - [a]_t, v - [b]_t$ are the input sharings. All parties use our multiplication protocol Mult on $[a]_t, [b]_t$ to compute $[c]_t$, where $c = a \cdot b$. Note that these two kinds of multiplications can be computed in parallel.
- For each multiplication gate in the second layer, suppose $u - [a]_t, v - [b]_t$ are the input sharings. Note that we have learnt u, v when evaluating the first layer, and we have computed the Beaver triple $([a]_t, [b]_t, [c]_t)$. Therefore, all parties compute $[z]_t := u \cdot v - u \cdot [b]_t - v \cdot [a]_t + [c]_t$.

We note that the original multiplication protocol in [DN07] requires the communication of 6 elements per party. Next, we show how to reduce the communication cost to 5 elements without breaking the form of the output sharing.

4.3.2 Improving the Original Multiplication Protocol in [DN07].

Recall that in the original multiplication protocol in [DN07]:

1. $P_{\texttt{king}}$ reconstructs a degree-$2t$ Shamir sharing $[e]_{2t} := [x]_t \cdot [y]_t + [r]_{2t}$ and sends e to other parties.
2. All parties locally compute $[z]_t := e - [r]_t$.

To keep the form of the output sharing, $P_{\texttt{king}}$ cannot replace e by a degree-t Shamir sharing $[e]_t$. Furthermore, to protect the secrecy of the multiplication result $x \cdot y$, r need to be uniformly random. Our main observation is that r being uniform is not equivalent to the double sharings $([r]_t, [r]_{2t})$ being uniform. To this end, we first decouple the relation between r and $([r]_t, [r]_{2t})$. Note that a pair of double sharings $([r]_t, [r]_{2t})$ is equivalent to a pair of sharings $([r]_t, [o]_{2t})$, where the first sharing is a degree-t Shamir sharing of r and the second sharing is a degree-$2t$ Shamir sharing of $o = 0$. To see this, given $([r]_t, [r]_{2t})$, we can set $[o]_{2t} := [r]_{2t} - [r]_t$; given $([r]_t, [o]_{2t})$, we can set $[r]_{2t} := [r]_t + [o]_{2t}$. When using a pair of sharings $([r]_t, [o]_{2t})$, the multipliation protocol becomes:

1. All parties locally compute $[e]_{2t} := [x]_t \cdot [y]_t + [r]_t + [o]_{2t}$.
2. $P_{\texttt{king}}$ collects all shares of $[e]_{2t}$ and reconstructs the secret e. Then $P_{\texttt{king}}$ sends the value e to all other parties.
3. After receiving e from $P_{\texttt{king}}$, all parties locally compute $[z]_t := e - [r]_t$.

Note that $[o]_{2t}$ is only used to compute $[e]_{2t}$. When $P_{\texttt{king}}$ is an honest party, $[o]_{2t}$ does not need to be a uniformly random degree-$2t$ sharing of 0. Thus, following the same argument as that in Section 4.2, we can use t-wise independent $[o]_{2t}$'s with uniformly random degree-t sharings $[r]_t$'s. In the following, we first introduce a protocol which allows all parties to generate random degree-$2t$ Shamir sharings of 0. Then, we will expand t such sharings into n sharings with t-wise independence.

Preparing Random Degree-$2t$ Shamir Sharings of 0. The functionality $\mathcal{F}_{\text{zero}}$ appears in Functionality 10. We follow the idea of preparing random degree-t sharings in [DN07]. At a high-level, each party generates and distributes a random degree-$2t$ sharing of 0. Then, use the transpose of a Vandermonde matrix acting as a randomness extractor to obtain $t+1$ random degree-$2t$ sharings of 0 from the n sharings generated by each party. Let $\boldsymbol{M}^{\mathrm{T}}$ be a predetermined and fixed Vandermonde matrix of size $n \times (t+1)$ (therefore $\boldsymbol{M}$ is a $(t+1) \times n$ matrix). The protocol Zero appears in Protocol 11. The communication complexity of Zero is $O(n^2)$ field elements. The amortized cost per sharing is 2 elements per party.

Functionality 10: $\mathcal{F}_{\text{zero}}$

1. $\mathcal{F}_{\text{zero}}$ receives from the adversary the set of shares $\{r_i\}_{i \in \mathcal{C}}$.
2. $\mathcal{F}_{\text{zero}}$ randomly samples t elements as the shares of the first t honest parties. Based on the secret $o = 0$, the t shares of the first t honest parties, and the t shares $\{r_i\}_{i \in \mathcal{C}}$ of corrupted parties, $\mathcal{F}_{\text{zero}}$ reconstructs the whole sharing $[o]_{2t}$. $\mathcal{F}_{\text{zero}}$ distributes the shares of $[o]_{2t}$ to honest parties.

Protocol 11: Zero

1. Each party P_i randomly samples a degree-$2t$ Shamir sharing of 0, denoted by $[o^{(i)}]_{2t}$ and distributes the shares to other parties.
2. All parties agree on a Vandermonde matrix $\boldsymbol{M}^{\mathrm{T}}$ of size $n \times (t+1)$. Then $\boldsymbol{M}$ is a matrix of size $(t+1) \times n$. All parties locally compute

$$([\tilde{o}^{(1)}]_{2t}, [\tilde{o}^{(2)}]_{2t}, \ldots, [\tilde{o}^{(t+1)}]_{2t})^{\mathrm{T}} \quad = \quad \boldsymbol{M}([o^{(1)}]_{2t}, [o^{(2)}]_{2t}, \ldots, [o^{(n)}]_{2t})^{\mathrm{T}}$$

and output $[\tilde{o}^{(1)}]_{2t}, [\tilde{o}^{(2)}]_{2t}, \ldots, [\tilde{o}^{(t+1)}]_{2t}$.

Lemma 2. *The protocol* Zero *securely computes the functionality $\mathcal{F}_{zero}$ in the presence of a fully malicious adversary controlling t corrupted parties.*

Proof. Let $\mathcal{A}$ denote the adversary. We will construct a simulator $\mathcal{S}$ to simulate the behaviors of honest parties. Recall that $\mathcal{C}$ denotes the set of corrupted parties and $\mathcal{H}$ denotes the set of honest parties.

Simulation for Zero. In the first step, when an honest party P_i needs to distribute a random degree-$2t$ sharing $[o^{(i)}]_{2t}$ of $o^{(i)} = 0$, for each corrupted party P_j, $\mathcal{S}$ samples a random element as its share of $[o^{(i)}]_{2t}$ and sends it to the adversary. For each corrupted party P_i, $\mathcal{S}$ receives the shares of $[o^{(i)}]_{2t}$ held by honest parties. $\mathcal{S}$ samples a random degree-$2t$ sharing based on the secret $o^{(i)} = 0$ and the shares held by honest parties, and views this degree-$2t$ sharing as the one distributed by P_i.

In the second step, $\mathcal{S}$ computes the shares of $[\tilde{o}^{(i)}]_{2t}$ held by corrupted parties and passes these shares to $\mathcal{F}_{\text{zero}}$.

Hybrid Arguments. Now we show that $\mathcal{S}$ perfectly simulates the behaviors of honest parties. Consider the following hybrids.

Hybrid$_0$: The execution in the real world.

Hybrid$_1$: In this hybrid, $\mathcal{S}$ changes the way of preparing a random degree-$2t$ Shamir sharing $[o^{(i)}]_{2t}$ of $o^{(i)} = 0$ for each honest party P_i:

1. $\mathcal{S}$ first randomly samples the shares of $[o^{(i)}]_{2t}$ held by corrupted parties and sends them to corrupted parties.

2. Then, based on the secret $o^{(i)} = 0$ and the shares of $[o^{(i)}]_{2t}$ held by corrupted parties, $\mathcal{S}$ samples a random degree-$2t$ sharing $[o^{(i)}]_{2t}$ (in the same way as $\mathcal{F}_{\text{zero}}$ in Step 2). Then $\mathcal{S}$ distributes the shares to the rest of honest parties.

For each corrupted party P_i, $\mathcal{S}$ locally computes the shares of $[o^{(i)}]_{2t}$ held by corrupted parties.

Note that this does not change the distribution of the random sharings generated by honest parties. The distribution of **Hybrid**$_0$ is identical to the distribution of **Hybrid**$_1$.

Hybrid$_2$: In this hybrid, $\mathcal{S}$ omits the second step when preparing $[o^{(i)}]_{2t}$ for each honest party P_i in **Hybrid**$_1$. Recall that in **Hybrid**$_1$, for all $i \in [n]$, $\mathcal{S}$ has computed the shares of $[o^{(i)}]_{2t}$ held by corrupted parties. For each $[\tilde{o}^{(i)}]_{2t}$, $\mathcal{S}$ computes the shares of corrupted parties and sends them to $\mathcal{F}_{\text{zero}}$.

We show that the distribution of **Hybrid**$_2$ is identical to the distribution of **Hybrid**$_1$.

Let $\boldsymbol{M}^{\mathcal{H}}$ denote the sub-matrix of $\boldsymbol{M}$ containing the columns of $\boldsymbol{M}$ with indices in $\mathcal{H}$ and $\boldsymbol{M}^{\mathcal{C}}$ denote the sub-matrix of $\boldsymbol{M}$ containing the columns of $\boldsymbol{M}$ with indices in $\mathcal{C}$. Let $([o^{(i)}]_{2t})_{\mathcal{H}}$ denote the vector of the sharings dealt by parties in $\mathcal{H}$ and $([o^{(i)}]_{2t})_{\mathcal{C}}$ denote the vector of the sharings dealt by parties in $\mathcal{C}$. Then,

$$\begin{aligned}([\tilde{o}^{(1)}]_{2t}, [\tilde{o}^{(2)}]_{2t}, \ldots, [\tilde{o}^{(t+1)}]_{2t})^{\mathrm{T}} &= \boldsymbol{M}([o^{(1)}]_{2t}, [o^{(2)}]_{2t}, \ldots, [o^{(n)}]_{2t})^{\mathrm{T}} \\ &= \boldsymbol{M}^{\mathcal{H}}([o^{(i)}]_{2t})_{\mathcal{H}}^{\mathrm{T}} + \boldsymbol{M}^{\mathcal{C}}([o^{(i)}]_{2t})_{\mathcal{C}}^{\mathrm{T}}\end{aligned}$$

Note that $\boldsymbol{M}^{\mathcal{H}}$ is a $(t+1) \times (t+1)$ matrix. By the property of Vandermonde matrices, $\boldsymbol{M}^{\mathcal{H}}$ is invertible. Therefore, given the sharings $\{[o^{(i)}]_{2t}\}_{i \in \mathcal{C}}$ dealt by corrupted parties, there is a one-to-one map from $\{[o^{(i)}]_{2t}\}_{i \in \mathcal{H}}$ to $\{[\tilde{o}^{(i)}]_{2t}\}_{i \in [t+1]}$. Note that the only difference between **Hybrid**$_1$ and **Hybrid**$_2$ is that, in **Hybrid**$_1$, $\{[o^{(i)}]_{2t}\}_{i \in \mathcal{H}}$ are randomly generated (based on the shares which have been sent to corrupted parties) while in **Hybrid**$_2$, $\mathcal{F}_{\text{zero}}$ directly generates $\{[\tilde{o}^{(i)}]_{2t}\}_{i \in [t+1]}$ based on the shares that corrupted parties should hold. However, this does not change the distribution of the shares of $\{[\tilde{o}^{(i)}]_{2t}\}_{i \in [t+1]}$ held by honest parties. To see this, note that for any sharings $\{[\tilde{o}^{(i)}]_{2t}\}_{i \in [t+1]}$ generated by $\mathcal{F}_{\text{zero}}$, we can compute back to a set of valid sharings $\{[o^{(i)}]_{2t}\}_{i \in \mathcal{H}}$. Therefore, the distribution of **Hybrid**$_2$ is identical to the distribution of **Hybrid**$_1$.

Note that **Hybrid**$_2$ is the execution in the ideal world and the distribution of **Hybrid**$_2$ is identical to the distribution of **Hybrid**$_0$, the execution in the real world. $\square$

The Improved Multiplication Protocol. For a sequence of n multiplication gates, all parties first prepare n random degree-t Shamir sharings using $\mathcal{F}_{\text{rand}}$, denoted by

$$[r^{(1)}]_t, \ldots, [r^{(n)}]_t.$$

Recall that the amortized communication cost of the instantiation of $\mathcal{F}_{\text{rand}}$ in [DN07, GS20] is 2 elements per sharing per party. Then, all parties invoke $\mathcal{F}_{\text{zero}}$ to prepare t random degree-$2t$ Shamir sharings of 0, denoted by

$$[o^{(1)}]_t, \ldots, [o^{(t)}]_t.$$

These t sharings are expanded to n sharings with t-wise independence. As EXPAND, we will use a predetermined $n \times t$ hyper-invertible matrix $\boldsymbol{M}$. The protocol EXPANDZERO appears in Protocol 12.

For the i-th multiplication gate, we will use $([r^{(i)}]_t, [\tilde{o}^{(i)}]_{2t}, P_i)$ and P_i will act as $P_{\texttt{king}}$. The protocol MULTDN appears in Protocol 13. As for the amortized communication cost per gate:

Protocol 12: ExpandZero

1. All parties agree on an $n \times t$ hyper-intertible matrix $\boldsymbol{M}$. All parties locally compute

$$([\tilde{o}^{(1)}]_{2t}, \ldots, [\tilde{o}^{(n)}]_{2t})^{\mathrm{T}} \quad = \quad \boldsymbol{M}([o^{(1)}]_{2t}, \ldots, [o^{(t)}]_{2t})^{\mathrm{T}}$$

2. All parties output $\{([\tilde{o}^{(i)}]_{2t}, P_i)\}_{i=1}^{n}$, where $([\tilde{o}^{(i)}]_{2t}, P_i)$ will be used for a multiplication gate handled by P_i.

- Preparing one random degree-t Shamir sharing using $\mathcal{F}_{\text{rand}}$ requires to communicate 2 elements per party.
- Preparing one t-wise independent degree-$2t$ Shamir sharing of 0 using $\mathcal{F}_{\text{zero}}$ and ExpandZero requires to communicate $2 \cdot t/n$ elements per party.
- The protocol MultDN requires to communicate 2 elements per party.

In summary, the amortized communication cost per gate is 5 elements per party.

Protocol 13: MultDN

1. Let $([r]_t, [o]_{2t}, P_i)$ be the random sharings which will be used in the protocol. Let $[x]_t, [y]_t$ denote the input sharings.
2. All parties locally compute $[e]_{2t} = [x]_t \cdot [y]_t + [r]_t + [o]_{2t}$.
3. P_i collects all shares and reconstructs the secret $e = x \cdot y + r$. Then P_i sends e to other parties.
4. All parties locally compute $[z]_t = e - [r]_t$.

4.3.3 Evaluating a Two-Layer Circuit.

Given a two-layer circuit, we assume that all parties hold a degree-t Shamir sharing for each input wire in the beginning. As described above, we will use MultDN to evaluate multiplication gates in the first layer. For multiplication gates in the second layer, note that all parties only need to obtain the output sharings. Therefore, we can use Mult, which only requires 4 elements per gate per party, to evaluate multiplication gates in the second layer.

Suppose there are N_1 multiplication gates in the first layer, and N_2 multiplication gates in the second layer. We assume that all parties have prepared the correlated randomness associated with these multiplication gates, i.e., N_1 pairs of sharings in the form of $([r]_t, [o]_t, P_i)$, and N_2 pairs of sharings in the form of $([r]_t, [r]_{2t}, P_i)$. In the main protocol, these sharings are prepared together at the beginning of the protocol. Then all parties execute Evaluate (Protocol 14) to compute the output sharings of this circuit.

4.3.4 Main Protocol.

Now we are ready to present the main protocol. Recall that we are in the client-server model. In particular, all the inputs belong to the clients, and only the clients receive the outputs. The functionality $\mathcal{F}_{\text{main}}$ appears in Functionality 15.

As [GSZ20], our protocol includes 4 phases:

Protocol 14: EVALUATE

1. All parties start with holding a degree-t Shamir sharing for each input wire of this circuit. For each multiplication gate in the second layer, we will transform the input sharings to the Beaver-triple friendly form $u - [a]_t$. Consider the following three cases.
    - If this sharing is an input sharing of the circuit, denoted by $[x]_t$, all parties set $u := 0$ and $[a]_t := -[x]_t$.
    - If this sharing is an output sharing of an addition gate in the first layer, all parties first locally compute this sharing, denoted by $[x]_t$, and then set $u := 0$ and $[a]_t := -[x]_t$.
    - If this sharing is an output sharing of a multiplication gate in the first layer, suppose $([r]_t, [o]_{2t}, P_i)$ are associated with this gate. All parties set $[a]_t := [r]_t$. The value u, which corresponds to e in MULTDN, will be computed when this multiplication gate is evaluated.
2. For each multiplication gate with input sharings $[x]_t, [y]_t$ in the first layer, all parties invoke MULTDN to compute $[z]_t$ where $z := x \cdot y$. For each multiplication gate with input sharings $(u - [a]_t), (v - [b]_t)$ in the second layer, where all parties have learnt the sharings $[a]_t, [b]_t$, all parties invoke MULT to compute $[c]_t$ where $c := a \cdot b$.
3. For each multiplication gate in the first layer, let e be the reconstruction result distributed by $P_{\texttt{king}}$ in MULTDN. If the output sharing of this gate is used as an input sharing of a multiplication gate in the second layer, all parties set $u := e$ for this input sharing.
4. Finally, for each multiplication gate with input sharings $(u - [a]_t), (v - [b]_t)$ in the second layer, all parties locally compute
$$[z]_t := u \cdot v - u \cdot [b]_t - v \cdot [a]_t + [c]_t$$
as the output sharing of this gate.

Functionality 15: $\mathcal{F}_{\text{main}}$

1. $\mathcal{F}_{\text{main}}$ receives from all clients their inputs.
2. $\mathcal{F}_{\text{main}}$ evaluates the circuit and computes the output. $\mathcal{F}_{\text{main}}$ first sends the output of corrupted clients to the adversary.
    - If the adversary replies `continue`, $\mathcal{F}_{\text{main}}$ distributes the output to honest clients.
    - If the adversary replies `abort`, $\mathcal{F}_{\text{main}}$ sends `abort` to honest clients.

- Input Phase: The clients will share their inputs to the parties.
- Computation Phase: The whole circuit will be partitioned into a sequence of two-layer sub-circuits. We will evaluate each sub-circuit using EVALUATE.
- Verification Phase: To check the correctness of the computation, we will check that
    - All parties receive the same values when using MULTDN to evaluate multiplication gates in the first layer of each sub-circuit.
    - Multiplication tuples computed by MULTDN and MULT are correct.

- Output Phase: All parties reconstruct the outputs to the clients.

To check that all parties receive the same values when using MultDN, all parties will compute a random linear combination of the values they received in MultDN and exchange their results. If a party receives different values, this party will abort. We will use the functionality $\mathcal{F}_{\text{coin}}$ introduced in Section 3.3 to generate a random element. The protocol CheckConsistency appears in Protocol 16. Recall that the communication complexity of the instaniation of $\mathcal{F}_{\text{coin}}$ in [GSZ20] is $O(n^2\kappa)$ bits. The communication complexity of CheckConsistency is $O(n^2\kappa)$ bits.

Protocol 16: CheckConsistency$(N, \{x^{(1)}, \ldots, x^{(N)}\})$

1. All parties invoke $\mathcal{F}_{\text{coin}}$ to generate a random element $r \in \mathbb{K}$. All parties locally compute
$$x := x^{(1)} + x^{(2)} \cdot r + \ldots + x^{(N)} \cdot r^{N-1}.$$
2. All parties exchange their results x's and check whether they are the same. If a party P_i receives different x's, P_i aborts.

Lemma 3. *If there exists two honest parties who receive different set of values $\{x^{(1)}, \ldots, x^{(N)}\}$, then with overwhelming probability, at least one honest party will abort in the protocol* CheckConsistency.

Proof. Suppose P_i, P_j are two honest parties and they receive $\{x^{(1)}, \ldots, x^{(N)}\}$ and $\{\tilde{x}^{(1)}, \ldots, \tilde{x}^{(N)}\}$ respectively. Suppose that there exists $i \in [N]$ such that $x^{(i)} \neq \tilde{x}^{(i)}$. Consider the following two polynomials in $\mathbb{K}$:

$$\begin{aligned} f(r) &= x^{(1)} + x^{(2)} \cdot r + \ldots + x^{(N)} \cdot r^{N-1} \\ \tilde{f}(r) &= \tilde{x}^{(1)} + \tilde{x}^{(2)} \cdot r + \ldots + \tilde{x}^{(N)} \cdot r^{N-1} \end{aligned}$$

Since there exists $i \in [N]$ such that $x^{(i)} \neq \tilde{x}^{(i)}$, $f(\cdot)$ and $\tilde{f}(\cdot)$ are two different polynomials. The number of r such that $f(r) = \tilde{f}(r)$ is bounded by the degree of $f(\cdot) - \tilde{f}(\cdot)$, which is $N-1$. Since r is uniformly chosen from $\mathbb{K}$, the probability that $f(r) = \tilde{f}(r)$ is at most $\frac{N-1}{|\mathbb{K}|} \leq \frac{N-1}{2^\kappa}$. Therefore, with overwhelming probability, $f(r) \neq \tilde{f}(r)$, which means that P_i, P_j will receive different values from each other and abort in the protocol CheckConsistency. □

To check that multiplication tuples computed by MultDN and Mult are correct, we will use $\mathcal{F}_{\text{multVerify}}$ from [GSZ20]. The protocol Main appears in Protocol 17.

Theorem 2. *Let c be the number of clients and $n = 2t + 1$ be the number of parties. The protocol* Main *securely computes $\mathcal{F}_{main}$ with abort in the $\{\mathcal{F}_{rand}, \mathcal{F}_{zero}, \mathcal{F}_{doubleRand}, \mathcal{F}_{coin}, \mathcal{F}_{multVerify}\}$-hybrid model in the presence of a fully malicious adversary controlling up to c clients and t parties.*

Proof. According to Theorem 1, we assume that the adversary controls exactly t parties.

Let $\mathcal{A}$ denote the adversary. We will construct a simulator $\mathcal{S}$ to simulate the behaviors of honest parties and honest clients. Recall that $\mathcal{C}$ denotes the set of corrupted parties and $\mathcal{H}$ denotes the set of honest parties.

Simulation for Main. We describe the strategy of $\mathcal{S}$ phase by phase.

- Simulation for Input Phase:

  For an input x belongs to an honest client, $\mathcal{S}$ randomly samples t elements as the shares of $[x]_t$ of corrupted parties. Then $\mathcal{S}$ sends these shares to corrupted parties.

Protocol 17: MAIN

1. **Input Phase**:

   For each client input x, client randomly samples a degree-t sharing $[x]_t$ and distributes the shares to all parties.

2. **Computation Phase – Preparing Correlated Randomness**:

   All parties start with holding a degree-t sharing for each input gate. The circuit is partitioned into a sequence of two-layer sub-circuits. Let N_1 denote the number of multiplications in the first layer of all sub-circuits, and N_2 denote the number of multiplications in the second layer of all sub-circuits. All parties prepare the correlated randomness as follows:

   - All parties invoke $\mathcal{F}_{\text{rand}}$ to prepare N_1 random degree-t Shamir sharings. Then all parties invoke $\mathcal{F}_{\text{zero}}$ to prepare $N_1 \cdot t/n$ random degree-$2t$ Shamir sharings of 0, and invoke EXPANDZERO to obtain N_1 degree-$2t$ Shamir sharings of 0. These sharings are transformed to N_1 pairs of sharings in the form of $([r]_t, [o]_{2t}, P_i)$.
   - All parties invoke $\mathcal{F}_{\text{doubleRand}}$ to prepare $N_2 \cdot t/n$ pairs of random double sharings. Then all parties invoke EXPAND to obtain N_2 pairs of double sharings in the form of $([r]_t, [r]_{2t}, P_i)$.

3. **Computation Phase – Evaluating Two-Layer Circuits**:

   All sub-circuits are evaluated in a predetermined topological order. For each sub-circuit with all the input sharings prepared, all parties invoke EVALUATE to compute the output sharings.

4. **Verification Phase**:

   - Suppose $e^{(1)}, \ldots, e^{(N_1)}$ are the values all parties received in MULTDN invoked in EVALUATE. All parties invoke CHECKCONSISTENCY to check that they receive the same values.
   - Suppose $\{([x^{(i)}]_t, [y^{(i)}]_t, [z^{(i)}]_t)\}_{i=1}^{N_1}$ denote the multiplication tuples computed by MULTDN invoked in EVALUATE, and $\{([a^{(i)}]_t, [b^{(i)}]_t, [c^{(i)}]_t)\}_{i=1}^{N_2}$ denote the multiplication tuples computed by MULT invoked in EVALUATE. All parties invoke $\mathcal{F}_{\text{multVerify}}$ to check the correctness of these $N_1 + N_2$ multiplication tuples.

5. **Output Phase**:

   For each output gate, suppose $[x]_t$ is the sharing associated with this gate and client is the client who should receive this output. All parties send their shares of $[x]_t$ to client. client checks whether the shares of $[x]_t$ is consistent. If not, client aborts. Otherwise, client reconstructs the result x.

For an input x belongs to a corrupted client, $\mathcal{S}$ receives from the adversary the shares held by honest parties. Note that, $\mathcal{S}$ learns $t+1$ shares of $[x]_t$. $\mathcal{S}$ reconstructs the whole sharing $[x]_t$ and sends x as the input of this corrupted client to $\mathcal{F}_{\text{main}}$.

Note that for each input sharing, $\mathcal{S}$ learns the shares held by corrupted parties.

- Simulation for Computation Phase – Preparing Correlated Randomness:

  In this part, $\mathcal{S}$ emulates $\mathcal{F}_{\text{rand}}, \mathcal{F}_{\text{zero}}, \mathcal{F}_{\text{doubleRand}}$ and receives the shares of corrupted parties. $\mathcal{S}$ follows the protocol EXPANDZERO and EXPAND to compute the shares of corrupted parties.

- Simulation for Computation Phase – Evaluating Two-Layer Circuits:

  In this part, $\mathcal{S}$ simulates the behaviors of honest parties in EVALUATE for each sub-circuit. For each sub-circuit with all the input sharings prepared, $\mathcal{S}$ will know the shares held by corrupted parties. Note that this is true for the first sub-circuit. We describe the strategy of $\mathcal{S}$ step by step.

- In the first step, $\mathcal{S}$ follows the protocol to compute the shares of corrupted parties for the input sharings of each multiplication gate in the second layer.
- In the second step, $\mathcal{S}$ simulates MULTDN and MULT as follows. For MULTDN:
  - Let $[x]_t, [y]_t$ denote the input sharings. $\mathcal{S}$ has computed the shares of $[x]_t, [y]_t$ held by corrupted parties. Let $([r]_t, [o]_{2t}, P_i)$ denote the sharings associated with this gate. $\mathcal{S}$ learns the shares of $[r]_t, [o]_{2t}$ held by corrupted parties when emulating $\mathcal{F}_{\text{rand}}, \mathcal{F}_{\text{zero}}$ and simulating EXPANDZERO.
  - $\mathcal{S}$ computes the shares of $[e]_{2t} := [x]_t \cdot [y]_t + [r]_t + [o]_{2t}$ held by corrupted parties. If P_i is corrupted, $\mathcal{S}$ samples random elements as shares of $[e]_{2t}$ of honest parties.
  - Depending on whether P_i is honest, there are two cases:
    - If P_i is honest, $\mathcal{S}$ receives the shares from corrupted parties. Let $[\tilde{e}]_{2t}$ denote the sharing when using the shares received from corrupted parties. This is to distinguish from $[e]_{2t}$ which uses the shares computed by $\mathcal{S}$ for corrupted parties. Then $\mathcal{S}$ computes the shares of $[d]_{2t} := [\tilde{e}]_{2t} - [e]_{2t}$ held by corrupted parties. Note that the shares of $[d]_{2t}$ held by honest parties are 0. $\mathcal{S}$ reconstructs the secret d. Finally, $\mathcal{S}$ samples a random element as e and sends e to corrupted parties.
    - If P_i is corrupted, $\mathcal{S}$ sends the shares of $[e]_{2t}$ of honest parties to P_i. Note that for $[e]_{2t}$, $\mathcal{S}$ knows all the shares. $\mathcal{S}$ reconstructs the secret e. Then $\mathcal{S}$ receives the value $\tilde{e}$ from P_i. If P_i sends different values to different honest parties, $\mathcal{S}$ marks this execution as `fail`, and uses the value of the first honest party. Finally, $\mathcal{S}$ computes $d := \tilde{e} - e$.
  - In the last step, $\mathcal{S}$ computes the shares of $[z]_t := \tilde{e} - [r]_t$ held by corrupted parties.

  For MULT:
  - Let $[x]_t, [y]_t$ denote the input sharings. $\mathcal{S}$ has computed the shares of $[x]_t, [y]_t$ held by corrupted parties. Let $([r]_t, [r]_{2t}, P_i)$ denote the sharings associated with this gate. $\mathcal{S}$ learns the shares of $[r]_t, [r]_{2t}$ held by corrupted parties when emulating $\mathcal{F}_{\text{doubleRand}}$ and simulating EXPAND.
  - $\mathcal{S}$ computes the shares of $[e]_{2t} := [x]_t \cdot [y]_t + [r]_{2t}$ held by corrupted parties. If P_i is corrupted, $\mathcal{S}$ samples random elements as shares of $[e]_{2t}$ of honest parties.
  - Depending on whether P_i is honest, there are two cases:
    - If P_i is honest, $\mathcal{S}$ receives the shares from corrupted parties. Let $[\tilde{e}]_{2t}$ denote the sharing when using the shares received from corrupted parties. This is to distinguish from $[e]_{2t}$ which uses the shares computed by $\mathcal{S}$ for corrupted parties. Then $\mathcal{S}$ computes the shares of $[d]_{2t} := [\tilde{e}]_{2t} - [e]_{2t}$ held by corrupted parties. Note that the shares of $[d]_{2t}$ held by honest parties are 0. $\mathcal{S}$ reconstructs the secret d. Finally, for $[e]_t$, $\mathcal{S}$ samples random elements as the shares of corrupted parties.
    - If P_i is corrupted, $\mathcal{S}$ sends the shares of $[e]_{2t}$ of honest parties to P_i. Note that for $[e]_{2t}$, $\mathcal{S}$ knows all the shares. $\mathcal{S}$ reconstructs the secret e. Then $\mathcal{S}$ receives the shares of $[e]_t$ of honest parties from P_i. $\mathcal{S}$ reconstructs the whole sharing and computes the secret of $[e]_t$, denoted by $\tilde{e}$. Finally, $\mathcal{S}$ computes $d := \tilde{e} - e$.
  - In the last step, $\mathcal{S}$ computes the shares of $[z]_t := [e]_t - [r]_t$ held by corrupted parties.
- In the rest of two steps (which only contain local computation), $\mathcal{S}$ follows the protocol and computes the shares of corrupted parties for each degree-t Shamir sharing.

• Simulation for Verification Phase:

- $\mathcal{S}$ first simulates CHECKCONSISTENCY. Note that $e^{(1)}, \ldots, e^{(N_1)}$ are either received from corrupted $P_{\texttt{king}}$'s or explicitly generated by $\mathcal{S}$. $\mathcal{S}$ follows the protocol honestly. If any party aborts *or $\mathcal{S}$ has marked this execution as* `fail`, $\mathcal{S}$ sends `abort` to $\mathcal{F}_{\text{main}}$ and aborts.

- $\mathcal{S}$ then emulates the functionality $\mathcal{F}_{\text{multVerify}}$. Recall that in the simulation of the computation phase, $\mathcal{S}$ has computed the shares of each multiplication tuple held by corrupted parties and the difference. $\mathcal{S}$ directly sends these shares and differences to the adversary as in $\mathcal{F}_{\text{multVerify}}$. If there exists a non-zero difference, $\mathcal{S}$ sets $b = \texttt{abort}$. Otherwise, $\mathcal{S}$ sets $b = \texttt{accept}$. Then $\mathcal{S}$ sends b to the adversary.
    - If $b = \texttt{accept}$ and the adversary replies `continue`, $\mathcal{S}$ moves to the next phase.
    - Otherwise, $\mathcal{S}$ sends `abort` to $\mathcal{F}_{\text{main}}$ and aborts.

- Simulation for Output Phase:

  For each output gate with $[x]_t$ associated with it, if the receiver is an honest client, $\mathcal{S}$ receives from the adversary the shares held by corrupted parties. Then $\mathcal{S}$ checks whether the shares are the same as the ones computed by $\mathcal{S}$. If true, $\mathcal{S}$ accepts the output. Otherwise, $\mathcal{S}$ rejects the output.

  If the receiver is a corrupted client, $\mathcal{S}$ receives the result x from $\mathcal{F}_{\text{main}}$. Then, based on the shares of $[x]_t$ held by corrupted parties and the secret x, $\mathcal{S}$ reconstructs the shares held by honest parties, and sends these shares to the adversary.

  Finally, if $\mathcal{S}$ rejects any output of honest clients, $\mathcal{S}$ sends `abort` to $\mathcal{F}_{\text{main}}$. Otherwise, $\mathcal{S}$ sends `continue` to $\mathcal{F}_{\text{main}}$.

Hybrid Arguments. Now we show that $\mathcal{S}$ perfectly simulates the behaviors of honest parties with overwhelming probability. Consider the following hybrids.

Hybrid$_0$: The execution in the real world.

Hybrid$_1$: In this hybrid, $\mathcal{S}$ computes the input of corrupted clients and sends them to $\mathcal{F}_{\text{main}}$. The distribution of **Hybrid$_1$** is identical to **Hybrid$_0$**.

Hybrid$_2$: In this hybrid, $\mathcal{S}$ simulates CheckConsistency. Concretely, $\mathcal{S}$ checks whether all honest parties receive the same values from $P_{\texttt{king}}$ in MultDN. If not, $\mathcal{S}$ sends `abort` to $\mathcal{F}_{\text{main}}$ and aborts. According to Lemma 3, the probability that an honest party aborts in this case is overwhelming. Therefore, the distribution of **Hybrid$_2$** is statistically close to the distribution of **Hybrid$_1$**.

Hybrid$_3$: In this hybrid, $\mathcal{S}$ simulates the output phase as described above. Note that the output phase is executed after the verification phase, which ensures the correctness of MultDN and Mult. Therefore, the output phase is executed only when the computation is correct. For an output gate with $[x]_t$ associated with it, if the receiver is an honest client, the shares that corrupted parties should hold are determined by the shares of honest parties. Therefore, if corrupted parties send different shares from the ones computed by $\mathcal{S}$, the shares of $[x]_t$ will be inconsistent and the client will reject the output. If the receiver is a corrupted client, the shares of honest parties are determined by the output x and the shares held by corrupted parties. Therefore, the shares prepared by $\mathcal{S}$ are identical to the real shares held by honest parties.

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

Hybrid$_4$: In this hybrid, $\mathcal{S}$ computes the difference of each multiplication tuple in the computation phase. Then $\mathcal{S}$ simulates the verification phase. Note that $\mathcal{F}_{\text{multVerify}}$ simply checks whether there is an incorrect multiplication tuple, which is equivalent to check whether there is a non-zero difference.

Therefore, the distribution of **Hybrid$_4$** is identical to the distribution of **Hybrid$_3$**.

Hybrid$_5$: In this hybrid, $\mathcal{S}$ simulates Evaluate. Since the parts which require interaction are MultDN and Mult, we only focus on the simulation of these two protocols. We argue the following two points:

- When a corrupted party P_i behaves as $P_{\texttt{king}}$, the shares of $[e]_{2t}$ of honest parties are uniformly random in both MultDN and Mult.

- When $P_{\texttt{king}}$'s send the same values to all honest parties in MultDN, $\mathcal{S}$ extracts the correct difference for each multiplication tuple computed by MultDN and Mult.

For the first point, following from a similar argument in Lemma 1, the random sharing $[r]_t + [o]_{2t}$ in MultDN and the random sharing $[r]_{2t}$ in Mult satisfy that the shares of honest parties are uniformly random.

Therefore, the shares of $[e]_{2t}$ of honest parties are uniformly random. For the second point, when $P_{\texttt{king}}$'s send the same values to all honest parties, all degree-t Shamir sharings are consistent in the sense that the shares of corrupted parties computed by $\mathcal{S}$ are consistent with the shares of honest parties. In this case, following from a similar argument in Lemma 1, $\mathcal{S}$ extracts the correct difference for each multiplication tuple computed by MULTDN and MULT.

Note that $\mathcal{F}_{\text{multVerify}}$ is only executed when all parties receive the same values from $P_{\texttt{king}}$'s in MULTDN. Therefore, the above two points show that the distribution of $\mathbf{Hybrid}_5$ is identical to the distribution of $\mathbf{Hybrid}_4$.

$\mathbf{Hybrid}_6$: In this hybrid, $\mathcal{S}$ emulates $\mathcal{F}_{\text{rand}}, \mathcal{F}_{\text{doubleRand}}, \mathcal{F}_{\text{zero}}$ and simulates EXPANDZERO and EXPAND as described above. Note that only the shares of corrupted parties are used in the rest of steps in $\mathbf{Hybrid}_5$. Therefore, the distribution of $\mathbf{Hybrid}_6$ is identical to the distribution of $\mathbf{Hybrid}_5$.

$\mathbf{Hybrid}_7$: In this hybrid, $\mathcal{S}$ simulates the input phase. The only difference between $\mathbf{Hybrid}_6$ and $\mathbf{Hybrid}_7$ is that, in $\mathbf{Hybrid}_6$, $\mathcal{S}$ uses the real input of honest clients to generate the input sharings, while in $\mathbf{Hybrid}_7$, $\mathcal{S}$ simply samples random elements as the shares of corrupted parties. Note that the distributions of the shares of corrupted parties in both hybrids are the same. Therefore, the distribution of $\mathbf{Hybrid}_7$ is the same as $\mathbf{Hybrid}_6$.

Note that $\mathbf{Hybrid}_7$ is the execution in the ideal world, and the distribution of $\mathbf{Hybrid}_7$ is identical to the distribution of $\mathbf{Hybrid}_0$, the execution in the real world. $\square$

Analysis of the Concrete Efficiency. In MAIN, all multiplication gates in the first layer of all sub-circuits are evaluated by MULTDN, which requires 5 elements per party per gate. All multiplication gates in the second layer of all sub-circuits are evaluated by MULT, which requires 4 elements per party per gate. Assuming that the number of multiplication gates in the first layer is roughly the same as the number of multiplication gates in the second layer, the concrete efficiency of MAIN is 4.5 elements per party per gate. Note that each sub-circuit is evaluated within one round of multiplication. Therefore, we reduce the number of rounds by a factor of 2. The overall communication complexity is the same as that in [GSZ20], i.e., $O(Cn\phi + n^2 \cdot \log C \cdot \kappa)$ bits.

4.4 Using PRG to Reduce Communication Complexity

We note that, relying on a pseudo-random generator (PRG), one can further reduce the communication complexity. This idea is not new and has appeared in previous works such as [LN17, NV18, BBCG+19].

The high-level idea is to let each pair of parties hold a common random seed, which is not known to other parties. For these two parties, they first evaluate the PRG on the common seed and chop the output string into several segments. When one party needs to send a random field element to the other party, both parties will transform the first unused segment into a field element. In this way, the communication cost of sending random shares can be saved.

To set up a common random seed for each pair of parties, one party simply generates a random seed and sends it to the other party. When a party P_i needs to distribute a random sharing $[r]_t$, P_i first sends random field elements to the first t parties $\{P_1, \ldots, P_t\}$ (which is achieved by using PRG without communication). Then P_i reconstructs the whole sharing $[r]_t$ using the shares of the first t parties and the secret r, and sends the shares to the rest of $t+1$ parties. In this way, the communication cost of distributing a random sharing is reduced to $t+1$ field elements.

When a party P_i needs to distribute a pair of random double shairngs $([r]_t, [r]_{2t})$, the random degree-t sharings can be distributed as above. Regarding $[r]_{2t}$, P_i sends random field elements to all other parties as their shares of $[r]_{2t}$ (which is achieved by using PRG without communication). Then, P_i can compute its share of $[r]_{2t}$ by using the shares of the other parties and the secret r. In this way, the communication cost of distributing a random sharing is reduced to $t+1$ field elements.

We briefly discuss how to use PRG in our two improvements.

- For the first improvement, recall that we transform t pairs of random double sharings generated by $\mathcal{F}_{\text{doubleRand}}$ to n pairs of double sharings which are t-wise independent. Then a multiplication gate is evaluated by Mult with one pair of t-wise independent double sharings. We can use PRG in the following two points:

1. In the instantiation of $\mathcal{F}_{\text{doubleRand}}$ in [DN07, GSZ20], each party generates a pair of random double sharings, and all parties then locally transform these n pairs to $t+1$ pairs of uniformly random double sharings. Relying on PRG, the communication cost of distributing a pair of random double sharings is reduced from $2n$ elements to $t+1 \approx 0.5n$ elements. Therefore, the communication cost of generating one pair of random double sharings is reduced from 4 elements per party to 1 elements per party.
2. In Mult, when $P_{\texttt{king}}$ needs to distribute a random degree-t sharing of the reconstruction result, it only needs to send $t+1$ elements instead of n elements. Therefore, the communication cost of Mult is reduced from 2 elements to 1.5 elements.

Originally, each multiplication gate requires to communicate $4 \cdot t/n + 2 \approx 4$ elements per party, including $4 \cdot t/n$ elements for generating a pair of t-wise independent double sharings and 2 elements for Mult. Relying on PRG, the communication cost is reduced to $1 \cdot t/n + 1.5 \approx 2$ elements per party.

- For the second improvement, note that multiplication gates in the second-layer of all sub-circuits are evaluated by Mult, which require 2 elements per gate per party. For the multiplication gates in the first layer, they are evaluated by MultDN. we can use PRG in the following two points:

1. In the instantiation of $\mathcal{F}_{\text{rand}}$ in [DN07, GSZ20], each party generates a random degree-t Shamir sharing, and all parties then locally transform these n sharings to $t+1$ uniformly random degree-t Shamir sharings. Relying on PRG, the communication cost of distributing a random degree-t Shamir sharing is reduced from n elements to $t+1 \approx 0.5n$ elements. Therefore, the communication cost of generating one random degree-t Shamir sharing is reduced from 2 elements per party to 1 elements per party.
2. In Zero, all parties need to prepare random degree-$2t$ Shamir sharings of 0. Relying on PRG, no communication is required when one party needs to distribute a random degree-$2t$ Shamir sharing of 0. Therefore, generating random degree-$2t$ sharings of 0 are free.

Originally, each multiplication gate requires to communicate $2 + 2 \cdot t/n + 2 \approx 5$ elements per party, including 2 elements for generating a random degree-t Shamir sharing, $2 \cdot t/n$ elements for generating a t-wise independent degree-$2t$ Shamir sharing of 0, and 2 elements for MultDN. Relying on PRG, the communication cost is reduced to $1 + 2 = 3$ elements per party. Therefore on average, the communication cost per multiplication is reduced to $(2+3)/2 = 2.5$ elements per party.

Regarding the security, we first start from the case where every pair of honest parties still uses truly random shares. Note that the corrupted parties learn nothing about the secret of a sharing distributed by an honest party even if the shares of corrupted parties are determined by themselves. This is because the corrupted parties only learn t shares of either a degree-t sharing or a degree-$2t$ sharing, which are independent of the secret value. Now, we can change the use of truly random shares to the shares generated by PRG for each pair of honest parties one by one. The security follows from a straightforward hybrid argument.

5 Experimental Evaluation

In this section, we evaluate and compare the concrete efficiency of our proposed improvements. As a baseline for comparison, we use the publicly available implementation of [CGH$^+$18]. We also use a setup similar to [CGH$^+$18].

5.0.1 Experiment Setup.

We run each party on an independent `C4.large` instance (2 cores with 2.9GHz and 3.75GB RAM) on Amazon AWS. The instances are all located in the same region (i.e. a *LAN* configuration). Throughout our experiments, we use the 61-bit Mersenne field, and we report the average of 5 executions as [CGH$^+$18].

Our benchmark consists of two sets of synthetic arithmetic circuits. The first set has 4 circuits of 1 million multiplication gates, ranging from 20 layers to 10,000 layers. The second set has 2 circuits of 10 million multiplication gates, each with 20 layers and 100 layers. Together, the two sets cover scenarios ranging from wide-and-shallow circuits to narrow-and-deep ones. We generate these two sets of synthetic arithmetic circuits by using the code from [CGH$^+$18]. We show running time on these circuits with 3 to 21 parties.

5.0.2 Benchmark Results.

Depth	version	3	5	7	9	11	15	21
20	[CGH$^+$18]	1126	1235	1642	1739	2029	2315	2762
20	[GSZ20]	763	857	1007	1068	1177	1301	1528
20	`round-compression`	642	709	810	858	974	989	1118
20	`t-wise`	545	622	711	752	842	917	1047
20	speedup vs [CGH$^+$18]	2.1x	2.0x	2.3x	2.3x	2.4x	2.5x	2.6x
20	speedup vs [GSZ20]	1.4x	1.4x	1.4x	1.4x	1.4x	1.4x	1.5x
100	[CGH$^+$18]	1122	1174	1591	1729	2033	2442	2915
100	[GSZ20]	696	887	1096	1122	1230	1430	1830
100	`round-compression`	655	719	839	849	914	1050	1190
100	`t-wise`	535	618	770	820	910	1038	1250
100	speedup vs [CGH$^+$18]	2.1x	1.9x	2.1x	2.1x	2.2x	2.4x	2.3x
100	speedup vs [GSZ20]	1.3x	1.4x	1.4x	1.4x	1.4x	1.4x	1.5x
1k	[CGH$^+$18]	1480	1802	2510	2793	3232	4053	5093
1k	[GSZ20]	1146	1358	1748	1920	2332	2744	3543
1k	`t-wise`	939	1136	1490	1618	1983	2389	3108
1k	`round-compression`	855	976	1195	1268	1511	1700	2100
1k	speedup vs [CGH$^+$18]	1.7x	1.8x	2.1x	2.2x	2.1x	2.4x	2.4x
1k	speedup vs [GSZ20]	1.3x	1.4x	1.5x	1.5x	1.5x	1.6x	1.7x
10k	[CGH$^+$18]	4470	6444	9641	10702	15040	18398	24693
10k	[GSZ20]	4457	5892	8747	9850	12832	18630	23026
10k	`t-wise`	4333	5641	8570	9327	12323	16580	22220
10k	`round-compression`	2477	3252	4713	5173	6633	8713	11719
10k	speedup vs [CGH$^+$18]	1.8x	2.0x	2.0x	2.1x	2.3x	2.1x	2.1x
10k	speedup vs [GSZ20]	1.8x	1.8x	1.9x	1.9x	1.9x	2.1x	2.0x

Table 1: This table shows running times (in milliseconds) for circuits with 1 million multiplication gates and of various depths. The columns show running times for different number of parties.

In Table 1 and Table 2, we compare the running time of four protocols: the baseline from [CGH$^+$18], the secure-with-abort protocol from [GSZ20], our improved protocol using t-wise independence (abbreviated as `t-wise`), and the further improved version with round compression (abbreviated as `round-compression`). The orders of the protocols shown in both tables are based on the running times. Table 1 shows results for circuits of 1 million multiplication gates, and Table 2 shows results for circuits of 10 million multiplication gates. Note that in Table 2, the baseline implementation runs out of memory when running with 11, 15, or 21 parties. We put N/A in those cases.

We observe that when the circuit depth D is small relative to its size (e.g. $D = 20, 100$), the `t-wise` version achieves better speedup than the `round-compression` version. When D is large (e.g. $D = 1,000, 10,000$), the `round-compression` version achieves significant further speedup.

D	version	3	5	7	9	11	15	21
20	[CGH+18]	11312	15118	17265	18988	N/A	N/A	N/A
20	[GSZ20]	7374	8795	10487	10883	11860	13520	15298
20	`round-compression`	5959	7176	8577	8846	9454	10538	11353
20	`t-wise`	5568	6461	7309	7892	8628	9524	10450
20	speedup vs [CGH+18]	2.0x	2.3x	2.4x	2.4x	N/A	N/A	N/A
20	speedup vs [GSZ20]	1.3x	1.4x	1.4x	1.4x	1.4x	1.4x	1.5x
100	[CGH+18]	12279	15434	17797	19273	N/A	N/A	N/A
100	[GSZ20]	7502	8220	10480	10845	12467	13112	14766
100	`round-compression`	6799	7319	8333	8867	9545	10396	11309
100	`t-wise`	5503	6076	7254	7818	8849	9144	10250
100	speedup vs [CGH+18]	2.2x	2.5x	2.5x	2.5x	N/A	N/A	N/A
100	speedup vs [GSZ20]	1.4x	1.4x	1.4x	1.4x	1.4x	1.4x	1.4x

Table 2: This table shows running times (in milliseconds) for circuits with 10 million multiplication gates and of various depths. The columns show running times for different number of parties.

This is because when D is small, communication bandwidth is the bottleneck of running times. The `t-wise` version effectively reduces the number of bytes communicated in each round, hence speeds up the running time. The overhead of the `round-compression` version when D is small surpasses its improvement in running time. However, when D is large, round latency becomes the bottleneck of running times, and improvements on communication complexity become less significant. The `round-compression` version in this case achieves significant speedup by reducing the round complexity.

In practice, we can have a switch in the code to decide whether to use the `t-wise` version or the `round-compression` version according to the size and depth of each input circuit. By combining the two improvements, we achieve around 2 times speedup compared with [CGH+18] in the overall running time, which includes both communication and computation time, in all cases, and around 1.4 times speedup compared with [GSZ20].

Acknowledgements.

V. Goyal, H. Li, Y. Song—Supported in part by the NSF award 1916939, DARPA SIEVE program, a gift from Ripple, a DoE NETL award, a JP Morgan Faculty Fellowship, a PNC center for financial services innovation award, and a Cylab seed funding award.
R. Ostrovsky—Supported in part by DARPA under Cooperative Agreement HR0011-20-2-0025, NSF grant CNS-2001096, US-Israel BSF grant 2015782, Google Faculty Award, JP Morgan Faculty Award, IBM Faculty Research Award, Xerox Faculty Research Award, OKAWA Foundation Research Award, B. John Garrick Foundation Award, Teradata Research Award, Lockheed-Martin Research Award and Sunday Group. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of DARPA, the Department of Defense, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes not withstanding any copyright annotation therein.
A. Polychroniadou—This paper was prepared in part for information purposes by the Artificial Intelligence Research group of JPMorgan Chase & Co and its affiliates ("JP Morgan"), and is not a product of the Research Department of JP Morgan. JP Morgan makes no representation and warranty whatsoever and disclaims all liability, for the completeness, accuracy or reliability of the information contained herein. This document is not intended as investment research or investment advice, or a recommendation, offer or solicitation for the purchase or sale of any security, financial instrument, financial product or service, or to be used in any way for evaluating the merits of participating in any transaction, and shall not constitute a solicitation under any jurisdiction or to any person, if such solicitation under such jurisdiction or to such

References

[ABF$^+$17] Toshinori Araki, Assi Barak, Jun Furukawa, Tamar Lichter, Yehuda Lindell, Ariel Nof, Kazuma Ohara, Adi Watzman, and Or Weinstein. Optimized honest-majority MPC for malicious adversaries - breaking the 1 billion-gate per second barrier. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 843–862. IEEE, 2017.

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

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

[BOGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 1–10. ACM, 1988.

[BSFO12] Eli Ben-Sasson, Serge Fehr, and Rafail Ostrovsky. Near-linear unconditionally-secure multiparty computation with a dishonest minority. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, pages 663–680, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[BTH08] Zuzana Beerliová-Trubíniová and Martin Hirt. Perfectly-secure MPC with linear communication complexity. In Ran Canetti, editor, *Theory of Cryptography*, pages 213–230, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 11–19. ACM, 1988.

[CDVdG87] David Chaum, Ivan B Damgård, and Jeroen Van de Graaf. Multiparty computations ensuring privacy of each party's input and correctness of the result. In *Conference on the Theory and Application of Cryptographic Techniques*, pages 87–119. Springer, 1987.

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

[DIK10] Ivan Damgård, Yuval Ishai, and Mikkel Krøigaard. Perfectly secure multiparty computation and the computational overhead of cryptography. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 445–465. Springer, 2010.

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

[DNPR16] Ivan Damgård, Jesper Buus Nielsen, Antigoni Polychroniadou, and Michael Raskin. On the communication required for unconditionally secure multiplication. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology – CRYPTO 2016*, pages 459–488, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[FLNW17] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 225–255. Springer, 2017.

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

[GLS19] Vipul Goyal, Yanyi Liu, and Yifan Song. Communication-efficient unconditional MPC with guaranteed output delivery. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019*, pages 85–114, Cham, 2019. Springer International Publishing.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 218–229. ACM, 1987.

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

[GSZ20] Vipul Goyal, Yifan Song, and Chenzhi Zhu. Guaranteed output delivery comes free in honest majority MPC. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020*, pages 618–646, Cham, 2020. Springer International Publishing.

[HM01] Martin Hirt and Ueli Maurer. Robustness for free in unconditional multi-party computation. In *Annual International Cryptology Conference*, pages 101–118. Springer, 2001.

[HMP00] Martin Hirt, Ueli Maurer, and Bartosz Przydatek. Efficient secure multi-party computation. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 143–161. Springer, 2000.

[LN17] Yehuda Lindell and Ariel Nof. A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 259–276. ACM, 2017.

[LP12] Yehuda Lindell and Benny Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. *Journal of cryptology*, 25(4):680–722, 2012.

[NNOB12] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In *Advances in Cryptology–CRYPTO 2012*, pages 681–700. Springer, 2012.

[NV18] Peter Sebastian Nordholt and Meilof Veeningen. Minimising communication in honest-majority MPC by batchwise multiplication verification. In Bart Preneel and Frederik Vercauteren, editors, *Applied Cryptography and Network Security*, pages 321–339, Cham, 2018. Springer International Publishing.

[Sha79] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979.

[Yao82] Andrew C Yao. Protocols for secure computations. In *Foundations of Computer Science, 1982. SFCS'08. 23rd Annual Symposium on*, pages 160–164. IEEE, 1982.