Fast Fully Secure Multi-Party Computation over Any Ring with Two-Thirds Honest Majority

Anders Dalskov,[1] Daniel Escudero[2] and Ariel Nof[3]

[1] Partisia, Denmark
[2] J.P. Morgan AI Research, U.S.A.
[3] Technion, Israel

Abstract. We introduce a new MPC protocol to securely compute any functionality over an arbitrary black-box finite ring (which may not be commutative), tolerating $t < n/3$ active corruptions while *guaranteeing output delivery* (G.O.D.). Our protocol is based on replicated secret-sharing, whose share size is known to grow exponentially with the number of parties n. However, even though the internal storage and computation in our protocol remains exponential, the communication complexity of our protocol is *constant*, except for a light constant-round check that is performed at the end before revealing the output.

Furthermore, the amortized communication complexity of our protocol is not only constant, but very small: only $1 + \frac{t-1}{n} < 1\frac{1}{3}$ ring elements per party, per multiplication gate over two rounds of interaction. This improves over the state-of-the art protocol in the same setting by Furukawa and Lindell (CCS 2019), which has a communication complexity of $2\frac{2}{3}$ *field* elements per party, per multiplication gate and while achieving fairness only. As an alternative, we also describe a variant of our protocol which has only one round of interaction per multiplication gate on average, and amortized communication cost of $\leq 1\frac{1}{2}$ ring elements per party on average for any natural circuit.

Motivated by the fact that efficiency of distributed protocols are much more penalized by high communication complexity than local computation/storage, we perform a detailed analysis together with experiments in order to explore how large the number of parties can be, before the storage and computation overhead becomes prohibitive. Our results show that our techniques are viable even for a moderate number of parties (e.g., $n > 10$).

1 Introduction

Secure Multiparty computation (MPC) is a set of techniques that enables a group of mutually distrustful parties to securely compute a given function on private data, while revealing only the output of the function. MPC protocols provide a general-purpose tool for computing on sensitive data while eliminating single points of failure.

Due to the strong guarantees that MPC protocols provide, together with the wide range of applications that they enable, several real-world problems where computation on sensitive data is required have been solved using MPC techniques. Practical applications have been found so far in key management, financial oversight [1], MPC secured database [11], market design [12], biomedical computations [22,19] and even satellite collision detection [41].

Since feasibility results for MPC were established in the 80s [49,36,9,18], the problem of constructing efficient protocols for secure computation has gained significant interest. Applications we see today are enabled thanks to a long line of works that have aimed at improving the efficiency of MPC protocols, as well as extending the theory of the field.

It is well known that the efficiency, or even the feasibility of certain MPC protocols, depends heavily on the type of security that is desired. This can be described in different and orthogonal categories:

Who can be corrupted? We assume that the adversary corrupts at most t out of the n participants.[4] Ideally, no requirement besides $t < n$ would be imposed. However, it turns out that, if we require the stronger condition of $t < n/2$, or even $t < n/3$, more efficient protocols with better security guarantees can be devised.

How do the corrupt parties behave? If the corrupt parties follow the protocol specification then the corruption is said to be *passive* (or semi-honest). In contrast, under an *active* (or malicious) corruption, the parties are not required to adhere to the protocol, and they may deviate arbitrarily.

[4] The so-called *generalized adversarial structures* allow for a more fine-grained description of the subsets of parties, but in this work we restrict to *threshold adversarial structures*, which are defined by a threshold t.

What is the adversary's computational power? Perfect security, which ensures that even a computationally unbounded adversary cannot learn anything about the honest parties' inputs, is only achievable if $t < n/3$. Statistical security allows a negligible probability of leakage, and is only achievable if $t < n/2$. Finally, if $t < n$ then the adversary has to be computationally bounded, as it must not be able to break certain hard computational problems.

Is the adversary allowed to cause an abort? Finally, we can choose if the computation must be guaranteed to terminate, which is called *full security*, or if the adversary can cause an *abort*, perhaps learning the result of the computation before the honest parties do or causing a denial-of-service. The former is only possible if $t < n/2$ given secure point-to-point channels and a broadcast primitive [48] (where the latter can be realized from a public-key infrastructure using digital signatures [32], or alternatively using only secure channels assuming $t < n/3$ [9,18]).

Different combinations of the categories are either impossible, or lead to protocols with different level of efficiencies. For example, it is well known that working in the dishonest majority setting, where $t < n$, adds a considerable overhead, and requires use of relatively expensive public-key cryptography primitives such as oblivious transfer. Furthermore, full security, or even fairness, is generally not achievable in this setting. In contrast, protocols for honest majority ($t < n/2$) or two-thirds honest majority ($t < n/3$) do not need to make use of computationally expensive cryptography, and they can achieve the strongest notion of full security; this, of course, at the expense of tolerating a weaker adversary corrupting less parties.

A good set of experimental results for different combinations of the categories above can be found in [28]. The main take-away lesson is that the efficiency of MPC protocols, plus the type of guarantees they can provide, depends heavily on different factors. For a practical deployment of MPC, it is necessary to look at the most realistic combination of the "parameters" above, in such a way that the resulting protocol is reasonably efficient and still secure for the application at hand. For example, although it may be reasonable to assume a passive adversary in a restricted set of settings, it is natural to desire security against active adversaries, given that there is no way to audit that a given participant of an MPC protocol is sending messages correctly, so this constitutes an easy way to cheat if there is enough motivation and profit from doing so.

On the other hand, the bound t on the number of corrupted parties is a less clear parameter to set. As mentioned before, the setting $t < n$ is ideal since it guarantees security for every single individual as long as it behaves honestly, but this might be too strong in some scenarios, especially when there is enough diversity among the participants and there is no strong reason to expect a large adversarial coalition to be formed. This, coupled with the inefficiency that protocols in this setting typically suffer from, leads us to naturally consider more lenient adversarial thresholds such as $t < n/2$ or $t < n/3$ that enable much more efficient protocols. Furthermore, many applications benefit from—or outright require—full security, which is only achievable in these threshold regimes. For example applications related to Machine Learning, where the computation is very large and thus very costly to run; or applications related to voting where rerunning the computation is simply not possible. The observations above set the motivation for our work.

1.1 Our Contribution

In this work, we consider the setting of $t < n/3$ with active and full security. Although this adversarial threshold is lower than the weak honest-majority and dishonest-majority settings, extremely efficient protocols can be designed in this regime, having very simple design and a thin layer that adds full security. In this direction, we present a simple new MPC protocol to compute any arithmetic circuit with the following characteristics:

- Full security against an active adversary corrupting at most $t < n/3$;[5]
- Computation over *any* finite ring (even non-commutative ones);
- Communication complexity of $n + t - 1$ ring elements in total per multiplication gate, plus a term which is independent of the number of multiplication gates. This is, the amortized communication cost is $\leq 1\frac{1}{3}$ ring elements per multiplication gate per party.
- Statistical security, except for the use of a PRG to boost efficiency.

Our protocol works by computing the circuit using a passive protocol which guarantees only privacy, and then verifying the correctness of the computation using a novel sub-protocol which incurs only constant communication cost (independent of the size of the computed circuit) and constant number of rounds.

[5] Our protocol can be easily generalized to arbitrary Q3 adversarial structures.

The only drawback of our protocol is that, due to the use of replicated secret sharing as the underlying secret sharing scheme, local storage and computation grows exponentially with the number of parties. However, since this does not affect communication of the underlying passive protocol, it is only for larger values of n that this weakness starts to kick in. In Section 7 we assess experimentally the feasibility of our protocol for a reasonably large number of parties. We show that replicated secret sharing-based protocols are not restricted in practice to only a very small number of parties (such as 3 or 4), as traditionally believed. We remark that if one is willing to settle for security with abort only, then this restriction can be removed, as we use the properties of replicated secret sharing only to identify cheaters. When considering security-with-abort only our protocol can therefore work with Shamir's secret sharing and achieve the same efficiency as the state-of-the-art protocol in this setting of Furukawa and Lindell [34]. However, unlike their work, we are able to augment our protocol to full security which is much stronger.

Our basic protocol requires 2 rounds of interaction per multiplication gate. As an alternative, we present a variant of our protocol with only a *single* round per multiplication, at the expense of increasing the communication cost slightly. In particular, for "natural" circuits, where the gate can be divided into groups, where output wires from gates in one group only enters gates from the second group, the communication cost is ≤ 1.5 sent ring elements per multiplication gate for each party. This variant is described in Section C in the Appendix.

Moreover, for $n = 4$ and $t = 1$, which is the base case of this setting, our protocol incurs communication of 1 ring element per party, distributed across two rounds. When using the variant with only one round of interaction, the cost increases to just $\frac{9}{8}$ elements (for natural circuits as defined above). This improves upon the recent protocols that were designed only for this setting [37,29,44], which requires communication of 1.5 ring elements and a single online round of interaction, or the more recent work of [45], which uses $1\frac{1}{4}$ elements per party, also in one online round. Furthermore, our protocol enjoys a very simple design that generalizes to any number of parties beyond $n = 4$.

Finally, our protocol works over any ring (even non-commutative ones) in a *black-box* way. This is in stark contrast with essentially all prior work (we elaborate on this point in Section 1.2), which rely on rings that are commutative, or have "high invertibility", like finite fields. As a result, our protocol can operate *natively* over relevant non-commutative rings such as matrix rings, which are widely used in settings like machine learning (e.g. neural networks, support vector machines, linear regression, etc.).[6] In addition, commutative rings such as integers modulo 2^k, which have received quite some attention recently [25,2,3,20,30,47], are also encompassed by our protocol.

1.2 Related Work

The goal of achieving linear communication complexity (in the number of parties) and with prefect security when $t < n/3$ was obtained in [6,39]. The protocol of [6] was used in a more practical setting in [5] by settling for security with abort only. Later, this was improved by [34], leading to a computationally-secure protocol with fairness in which each party sends, on average, $2\frac{2}{3}$ field elements. As explained above, we improve over [34] by achieving full security.

Several works have focused on achieving full security in the setting of $n = 4$ and $t = 1$ [37,44,29]. The state-of-the-art protocol by Koti *et al.* [45] requires sending 1.25 ring elements per multiplication per party in one online round, which we improve for any natural circuit. It should be noted that [45] also provide a protocol for 3-input multiplication gates with 3 ring element sent per party in one online round, which we did not consider in this work.

In the setting of $t < n/2$ recent breakthrough results have shown how to achieve full security with low communication. The protocol of [40] requires each party to send 5.5 field elements per multiplication with information-theoretic security, while the protocol of [16] reduces communication to 1.5 ring elements by allowing use of any PRG. For $n = 3$ and $t = 1$, communication can be further reduced to 1 ring elements as shown in [15]. While the later protocols achieve similar amortized communication as ours with a more powerful adversary, our protocol has several advantages over theirs. First, the additive overhead to achieve active security in these protocols is *logarithmic* in the size of the circuit while ours is *constant*. The same applies to the number of calls to an expensive broadcast channel which is logarithmic in the circuit's size in these protocols and constant in ours. Furthermore,for the case of the ring $\mathbb{Z}_{2^k}$, their protocol requires arithmetic over Galois ring extensions of very large degree (> 50), whose concrete efficiency is unclear (see

[6] Furthermore, for the particular case of matrix rings, our secret-sharing scheme enables local conversions between shared matrices and "entry-wise" sharings, which is essential for many applications like the ones described above, as they typically manipulate individual entries along with the matrix arithmetic. This is not possible for example with the work of [33].

for example [29]). The above is due to the fact that they rely on distributed zero-knowledge proofs [13], which we are able to avoid in our setting. Finally, achieving robustness in the setting of $t < n/2$ requires using authentication tags which makes these protocols much more complicated and computationally expensive compared to our protocol which avoids this completely. Without full security, for example with security with abort, efficient protocols exist in the honest majority setting (e.g. [21] and [3]).

We also point out that replicated secret-sharing for an arbitrary number of parties has been already used in works like [4]. However, in their protocol, communication per gate grows exponentially with the number of parties, whereas in our protocol the communication cost per gate is constant.

Finally, the feasibility of computation over general rings was shown in [27]. The protocol from [6] was generalized to the ring of integers modulo 2^k in [2]. Furthermore, the 4-party protocols of [37,29,44] also work over this ring. In contrast, the work of [34] focus on multiparty computation over finite fields. We are thus not aware of a concretely-efficient work[7] in this setting for more than $n = 4$ that applies to general rings. More recently, [33] considers MPC for arbitrary circuits over black-box finite rings, which could be potentially non-commutative. However, their results are mostly of theoretical interest since, due to the lack of commutativity, the offline phase in their protocols results in a large overhead. Nevertheless, we remark that their local computation, unlike ours, is polynomial in the number of parties.

1.3 Organization

We begin by presenting a detailed and self-contained overview of our construction in Section 2. Then, in Section 3 we present basic preliminaries like notation and some elementary results in ring theory. In Section 4 we introduce replicated secret-sharing, the basic building block on top of which our protocol is built. Section 5 presents the main protocols needed in our construction, which are related to multiplying shared values and verifying their correctness. Then, in Section 6 we put together these protocols together with the basic properties of replicated secret-sharing to obtain our final, full-fledged protocol for G.O.D. secure computation with $t < n/3$. Finally, experimental results are discussed in Section 7.

2 Detailed Overview of Our Construction

In our construction, we use the well-known player elimination framework which was introduced by [42] to achieve full security. In this framework, the parties divide the circuit into segments, each of which are computed as a separate unit. Each segment is computed first using a protocol with *weak* security, which in our case means that the protocol only guarantees privacy and not correctness. At the end of the segment the parties verify the correctness of the computation, and if verification succeeds, then the parties proceed to the next segment. If verification fails, parties finds someone who cheats, and remove them from the protocol, after which the segment is repeated with fewer parties. In our case, the parties will locate a pair of parties with the guarantee that one of them is corrupt. Such a pair is called "semi-corrupt" in our protocol. The semi-corrupt pair is removed by converting the secret sharing of each value on the input layer of the current segment from a t-out-of-n secret sharing into a $t-1$-out-of-$n-2$ secret sharing. Observe that when an honest majority exists, such a conversion maintains the security threshold (i.e., if $n \geq 3t+1$, then $n-2 \geq 3(t-1)+1$). Note also that the number of repetitions is bounded by the number of corrupted parties t. Thus, by carefully choosing the size of the segment, we can expect that each gate is computed approximately once.

2.1 Private Computation of a segment

We denote by $[\![x]\!]_d$ a replicated secret sharing of a secret x with threshold d. Notice that, initially, $d = t$, but after several segment repetitions it might be the case that $d < t$. Recall that in replicated secret sharing, there are $\binom{n}{d}$ shares which sum-up to x, and each subset of parties of size $n-d$ is given one share. Since this scheme is linear, all linear gates can be computed without any interaction. To compute multiplications, we use an optimized version of the semi-honest DN protocol [31]. While this protocol is usually used in the literature with Shamir's secret sharing, it can also be used with replicated secret sharing. This was observed for the first time, to the best of our knowledge, in [14], for the setting of $t < \frac{n}{2}$. In the "textbook" version of this protocol, the parties multiply $[\![x]\!]_d$ and $[\![y]\!]_d$ in the following way: First, the parties locally multiply $[\![x]\!]_d \cdot [\![y]\!]_d$ to obtain $[\![x \cdot y]\!]_{2d}$, mask the result with a random sharing $[\![r]\!]_{2d}$ and send it to P_1. Then, P_1 reconstructs $x \cdot y - r$ and send it back to the parties. Finally, the parties locally

[7] By "concretely-efficient", we mean protocols with low communication that use only cheap symmetric crypto.

add $x \cdot y - r$ to $[\![r]\!]_d$ and obtain $[\![x \cdot y]\!]_d$. Privacy for semi-honest adversaries follows from the fact that the parties see only masked values from which nothing can be learned. A somewhat surprising observation from [39] is that when $t < \frac{n}{3}$ (and in fact for any strong honest majority) a *malicious* P_1 can carry-out an attack which allows him to learn private information (instead of just changing the output). This attack is caused by the fact that there exists redundancy in the masking. That is, since $2d < n - 1$, once P_1 received $2d$ shares of $x \cdot y - r$, it can compute the remaining shares. This is used to carry-out an attack over two gates, which results in learning private shares. A simple way to prevent this attack is to use a full masking, i.e., to mask the message using $[\![r]\!]_{n-1}$. In the context of replicated secret sharing, this means that, given $[\![x]\!]_d$, $[\![y]\!]_d$, the parties need to compute an additive sharing of $x \cdot y - r$ (since replicated secret sharing with threshold $n - 1$ is exactly an additive secret sharing). We optimize this by letting only $2d + 1$ parties compute an additive sharing of $x \cdot y - r$. This means that a set U of $2d + 1$ parties are required to convert their d-out-of-n shares of x and y to an additive sharing of $x \cdot y$ across the parties in U, and then mask it with a preprocessed additive sharing of a random r. We denote an additive sharing of x across parties in a set U by $\langle x \rangle^U$. The above implies that given a set of parties U, the parties need to prepare in advance a pair $([\![r]\!]_d, \langle r \rangle^U)$ for our multiplication protocol. We show how the parties can prepare this correlated randomness without any interaction (but a short setup step) building upon the PRSS method of [26] and then show how, in our setting of $d < \frac{n}{3}$, a set U of $2d + 1$ parties can locally convert $[\![x]\!]_d$, $[\![y]\!]_d$ and $\langle r \rangle^U$ to $\langle x \cdot y - r \rangle^U$. These additive shares are then sent to P_1 who reconstructs $x \cdot y - r$ by summing the shares and sending the result to the parties as before.

Another optimization for reducing communication in the protocol is obtained by observing that it suffices for P_1 to send $x \cdot y - r$ to a subset of $n - d - 1$ parties. This holds since adding a constant to a sharing is done by adding it to one share. Thus, we can let only one subset of $n - d$ parties, which includes P_1, add $x \cdot y - r$ to their joint share of r. Overall, we obtain that $2d$ elements are sent in the first round of the protocol (by letting P_1 be in U) and $n - d - 1$ are sent in the second round. Thus, the communication cost per party is $\frac{2d+n-d-1}{n} = 1 + \frac{d-1}{n} < 1\frac{1}{3}$ elements.

We remark that, up to this point, everything that has been described only requires the additive group to be abelian, with the potential exception of the local multiplication method to obtain $[\![x \cdot y]\!]_{2d}$ from $[\![x]\!]_d \cdot [\![y]\!]_d$. However, as we will see in Section 4, this does not require commutativity of the multiplication operation. In a nutshell, so far the protocol can be instantiated with *any* finite ring.

2.2 Verifying the computation with cheating identification

Observe that in the above protocol there are no private messages; the only reason for the communication pattern through P_1 is to reduce bandwidth, but in fact the parties could send their messages in the first round directly to all the other parties. Leveraging this, the first step in the verification protocol is to agree on a "compressed" transcript of all the executions of the multiplication protocol. This is achieved by having each party broadcast a random linear combination of the messages they sent and received. If there is an inconsistency between a compressed message published by P_1 and any party P_i, then (P_1, P_i) is a new pair of parties to eliminate from the protocol (note that since, in the protocol, each party only sends at most one message to P_1 and receives at most one message from P_1, such inconsistency can occur only between P_1 and other parties). Since inconsistency of sent and received compressed messages cannot happen between two honest parties, we are guaranteed that either P_1 or P_i is corrupted. If all published messages are consistent, then the parties hold a compressed transcript of all executions and proceed to the next part of the protocol.

In the second step, the parties verify the correctness of all messages sent in the executions of the multiplication protocol by verifying the correctness of the compressed message sent by each party. This includes the message sent by each party to P_1, and the message sent by P_1 in the second round. Recall that P_1 receives additive shares of $x \cdot y - r$ and adds them together with its own share to compute its message in the second round. We thus need to verify that each party sent its correct additive share, and then, that P_1 used its correct own share to compute $x \cdot y - r$. Note that since the parties hold now the compressed message of each party to P_1 and the compressed message P_1 sent, they can compute P_1's implicit first round compressed message. Hence, the parties first verify the first round message of each party P_i with $i \neq 1$, and then, if the check was successful, the parties verify P_1's first round message.

The verification is carried-out in the following way. Recall that in the first round a set of parties U locally compute their additive share of $x \cdot y - r$ by converting $[\![x]\!]_d$ and $[\![y]\!]_d$ to $\langle x \cdot y \rangle^U$ and then subtracting their additive share of r. The idea of our verification procedure is to let the parties compute a $2d$-out-of-n sharing of each party's message. Towards achieving this, we observe:

1. To convert its shares of x and y into an additive share of $x \cdot y$, each party takes a *linear* combination of its shares.
2. Each share of x and y is also shared across the parties via a d-out-of-n replicated secret sharing. This follows since each share x_j (or y_j) is held by $n - 2d$ parties. Thus, we can define a secret sharing of x_j by letting the shares of all other subsets of size $n - 2d$ be zero.

Leveraging this, we show that given d-out-of-n sharings of *each share*, the parties can locally compute a $2d$-out-of-n secret sharing of the linear combination of these shares. It remains to generate a $2d$-out-of-n sharing of each party's share of r. To achieve this, we go back to the preprocessing protocol which produces the correlated randomness. We present a simple way to produce the preprocessing $([\![r]\!]_d, \langle r \rangle^U)$ for each execution of the multiplication protocol, such that each party's additive share of r is also shared in a robust way, i.e., via a d-out-of-n secret sharing. This implies that the parties can take their sharing of each party's additive share and convert it locally to a $2d$-out-of-n secret sharing, and so the parties can locally compute a $2d$-out-of-n secret sharing of each party's additive share of $x \cdot y - r$. It remains to take a random linear combination of all these sharings, which is a local operation, to compute a $2d$-out-of-n sharing of each party's compressed message. Now, observe that each share of each party' message is held by $n - 2d$ parties, and so, since in our setting $3d + 1 \leq n$, it follows that $n - 2d \geq d + 1$, which means that each share is held by at least one honest party. Thus, when opening the $2d$-out-of-n sharing, the adversary cannot open it to any value but the correct value. We can thus ask the parties to open each party's compressed message and verify its correctness. If the corrupted parties send incorrect shares, then this will result with pair-wise inconsistency, which is translated to a semi-corrupt pair to eliminate. Otherwise, the parties know that they hold the correct compressed message, and can compare it to the message published in the first step.

To sum-up the discussion, assume that a corrupted party sent an incorrect message to an honest party in any of the multiplication protocol's executions. If it tries to publish the "correct" message in the first step of the verification, then this will result in an inconsistency with the received message published by the honest party. If it publishes the actual message that it sent, then this will result in an inconsistency with the correct message the parties compute and reconstruct in the second round of the verification. In both cases, a pair of parties which includes a corrupted party will be located. The only way that cheating can succeed is, if a random linear combination of incorrect messages yields the same value as a random linear combination of the correct messages. This yields a statistical error which can be made sufficiently small.

As before, all we have described so far also holds even if the ring is non-commutative. The only non-standard result is the fact that the probability that the dot product between a non-zero and a random vector can be bounded. This is indeed the case, as discussed in Section 3.1.

One last subtle issue that we need to take care of is to prevent any leakage from the shares of each party's message. This is solved by randomizing the shares, which is done by adding a random zero sharing with threshold $2d$. We show how such a random sharing can be produced non-interactively.

Observe that the above protocol to compute a $2d$-out-of-n sharing of each party's compressed message is completely local. We thus obtain a verification protocol, where the parties need only to broadcast at most one sent message and at most one received message, and to open n sharings. The communication cost is therefore independent of the size of the circuit. Moreover, the number of rounds is constant too.

2.3 Elimination and recovery

Once a semi-corrupt pair was found, the parties need to remove it. To achieve this, it suffices to only update the sharings of the inputs to each layer, which are also the outputs of the previous layer. Observe that each said value was computed by taking $[\![r]\!]_d + (x \cdot y - r)$. In order to update the sharings, we thus need to (i) convert $[\![r]\!]_d$ into $[\![r]\!]_{d-1}$ and (ii) that a subset of only $n - 2 - (d - 1) = n - 1 - d$ parties add $x \cdot y - r$ to $[\![r]\!]_{d-1}$. Now, since the correlated randomness in our protocol is produced without interaction, we can achieve task (i) by only moving PRF keys between subsets of parties, which can be done with constant cost. For task (ii), we need to take an action only if the current subset of parties which holds $x \cdot y - r$, contains both eliminated parties, since in this case the only $n - d - 2$ active parties know $x \cdot y - r$. In this case, we need $x \cdot y - r$ to be handed to a new party. This can be done by having all current $n - d$ parties who know $x \cdot y - r$ send this value to some other party. Since, given that $n - d \geq 2d + 1$, every subset of $n - d$ parties has an honest majority, that party could identify the correct $x \cdot y - r$, by taking the majority value.

2.4 Reducing the number of rounds

Our protocol requires 2 rounds of interaction per multiplication gate. Using the ideas from [38], we can reduce the number of rounds to only a single round of interaction, at the expense of a small increase in bandwidth. Reducing round complexity may be preferred over minimizing communication when the network is slow. The observation of [38] is that after computing the DN protocol, the parties hold masked value on the output wire (this requires P_1 to send $x \cdot y - r$ to *all* parties and this is why communication increases). Thus, if the parties preprocess random Beaver triples, they can compute the gates in the next layer without interaction. This technique thus trades online with offline communication, where in the latter it is possible to carry-out all interaction in parallel. In Appendix C we provide a formal description and analysis, and in particular show that we can apply our new verification method over this variant of the protocol.

2.5 The case of 4 parties and one corruption

In the base case of our two-thirds honest majority setting, there are 4 parties and one corrupted party. Note that in this case, there is no need to split the circuit into segments. The parties can verify the circuit at once, and if the verification ends with a pair of parties to eliminate, then it means that the remaining two parties are honest, and so one of them can be used as a trusted party and compute the function for the parties.

3 Preliminaries

Notation. Let κ be the security parameter and let n be the number of parties. We denote the set of involved parties by $\mathcal{P} = \{P_1, \ldots, P_n\}$. Let t be an upper bound in the number of corrupted parties, and assume that $t < n/3$. Many of our subprotocols will be presented with a threshold $d \leq t$, since, due to the player elimination framework, they may be executed with a smaller threshold than t. We use the notation $[n]$ for the set $\{1, \ldots, n\}$.

3.1 Background in Ring Theory

Let R be *any* finite ring. We only assume procedures for adding and multiplying ring elements, as well as sampling uniformly random elements. A set $A \subseteq R$ is called *exceptional* if, for all $x, y \in A$ with $x \neq y$, $x - y$ is invertible.[8]

For the rest of the paper, let $\mathbb{A}$ be the any of the largest exceptional subsets of R, and let $\omega_R = |\mathbb{A}|$. We will need the following lemma in our protocol.

Lemma 1. *Let $a, b \in R$, with $a \neq 0$. Then* $\Pr_{x \xleftarrow{\$} \mathbb{A}}[x \cdot a + b = 0] \leq 1/\omega_R$.

Proof: Let $x, y \in \mathbb{A}$ such that $x \cdot a + b = 0$ and $y \cdot a + b = 0$, then $(x - y) \cdot a = 0$, but since $x - y$ is invertible, this implies that $a = 0$, which is a contradiction. This shows that there can be at most one $x \in \mathbb{A}$ that satisfies $x \cdot a + b = 0$, and therefore the probability of this event happening for a random sample in $\mathbb{A}$ is at most $1/|\mathbb{A}| = 1/\omega_R$. ■

Observe that if R is a field then we may take $\mathbb{A} = R$, and therefore $\omega_R = |R|$. On the other hand, if R is the ring of integers modulo 2^k, it can be shown that there are no exceptional sets of size 3 or more, so we may take $\mathbb{A} = \{0, 1\}$, and hence $\omega_R = 2$.

3.2 MPC Security Definition

In this work, we consider adversaries who can follow an arbitrary strategy to carry-out their attack. We use the standard ideal/real paradigm [35] in order to define security, where an execution in the ideal world with a trusted party who computes the functionality for the parties is compared a real execution. Although our protocols can be computed with information-theoretic security, we use minimal computational assumptions to achieve better concrete efficiency. Thus, when we say that a protocol "computationally computes" an ideal functionality, this means that the output of the ideal execution with an ideal world simulator is *computationally indistinguishable* from the output of the real world execution. In some of our protocols, there is also a statistical error, which is independent of the computational security parameter. As in [34],

[8] In a finite non-commutative ring, a is invertible if there exists b such that $a \cdot b = b \cdot a = 1$.

we formalize security in this case by saying that the outputs of the two executions can be distinguished with probability of at most some negligible function in the security parameter, *plus* the statistical error.

We prove the security of our protocols in a hybrid model, where parties run a protocol with real messages and also have access to a trusted party computing a subfunctionality for them. The modular sequential composition theorem of [17] states that one can replace the trusted party computing the subfunctionality with a real secure protocol computing the subfunctionality. When the subfunctionality is g, we say that the protocol works in the g-hybrid model.

Universal Composability. Protocols that are proven secure in the universal composability framework have the property that they maintain their security when run in parallel and concurrently with other secure and insecure protocols. In [46], it was shown that any protocol that is proven secure with a black-box non-rewinding simulator and also has the property that the inputs of all parties are fixed before the execution begins (called input availability or start synchronization in [46]), is also secure under universal composability. Since the input availability property holds for all of our protocols and subprotocols, it is sufficient to prove security in the classic stand-alone setting and automatically derive universal composability from [46]. This also enables us to call the protocol and subprotocols that we use in parallel and concurrently (and not just sequentially), allowing us to achieve more efficient computation (i.e. by running many executions in parallel or running each layer of a circuit in parallel).

Broadcast and Agreement A protocol for Byzantine agreement takes a bit from all parties as an input, and let all honest parties reach a consensus in the presence of t corrupted parties. If all honest parties holds the same bit b, then the protocol guarantees that all honest parties will output b. In our protcol, we will use Byzantine agreement to let a party broadcast one bit to the other parties. This can be done by letting the party send the bit to all parties and then run Byzantine agreement. In our setting of $t < n/3$, perfect Byzantine agreement can be achieved with quadratic communication complexity [10,24]. We stress that the number of calls to broadcast in our protocol is *constant* and so any way to implement it suffices.

4 Replicated Secret Sharing and Its Operations

Replicated secret sharing. The replicated secret sharing scheme [43], with threshold $d \leq t$, is defined by the following procedures. Below, we let $\lambda = \binom{n}{d}$ and let $T_1, \ldots, T_\lambda \subset \mathcal{P}$ be all subsets of parties of size $n - d$.

- $\mathsf{share}(x, d)$: To share a secret x with threshold d, the dealer generates λ random $x_{T_1}, \ldots, x_{T_\lambda} \in R$ under the constraint that $x = x_{T_1} + \cdots + x_{T_\lambda}$. Then, the dealer hands x_{T_j} to the parties in T_j. The share $\vec{x}_i$ held by party P_i is a tuple consisting of all x_{T_j} such that $P_i \in T_j$. We say that $[\![x]\!]_d$ is the collection of all $\vec{x}_i$s.
- $\mathsf{reconstruct}([\![x]\!]_d, i)$: In this interactive procedure, the parties in each subset T where $|T| = n - d$ and $P_i \notin T$, send all their shares to P_i. For each subset of parties T holding a share x_T, if P_i received different values for x_T, then P_i takes the *majority value* to be x_T. Finally, P_i sets $x = \sum_{T \subseteq \mathcal{P}:\ |T| = n-d} x_T$.

Secrecy of this scheme follows from the fact any set of d corrupted parties miss one additive share (namely, the one indexed by their complement), and so the secret could be any value in the ring.

Note that the sharing procedure described above implies that a corrupted dealer may cheat by sending different values to different parties in the same subset T of parties. In this case, we say that the sharing is inconsistent. We formally define the notion of consistency in the following definition.

Definition 1 (Consistency). *We say that $[\![x]\!]_d$ is consistent if for each two honest parties P_i and P_k, for each $T \subset \mathcal{P}$, such that $|T| = n - d$ and $P_i, P_k \in T$, it holds that the same x_T is held by both P_i and P_k.*

Relying on the definition of consistency, we next prove that the reconstruct procedure defined above is robust, i.e., the receiving party will always obtain the correct secret.

Claim. If $[\![x]\!]_d$ is consistent and $d < \frac{n}{3}$, then $\mathsf{reconstruct}([\![x]\!]_d, i)$ ends with P_i holding x, even in the presence of malicious adversaries controlling up to d parties.

Proof: In the procedure, P_i receives from all parties in each subset T of $n - d$ parties with $P_i \notin T$, the share x_T held by this subset. Since $3d < n$, we have that $n - d \geq 2d + 1$. This implies that in each subset, there is a majority of honest parties. Since $[\![x]\!]_d$ is consistent, it means that all honest parties in each T will send the same value, and so by taking the majority value, P_i will obtain the correct share. ∎

Complexity. For n parties and threshold d, there are $\binom{n}{d}$ distributed shares. Each party holds $\binom{n-1}{d}$ shares. When reconstructing a secret towards P_i, party P_i receives $\binom{n-1}{d-1}$ missing shares, and each share is sent by $n-d$ parties. Thus, the overall communication is $\binom{n-1}{d-1} \cdot (n-d) = \binom{n-1}{d} \cdot d$ ring elements.

Pairwise consistency check. The above definition gives us an easy way to check that a sharing is consistent, by having each pair of parties comparing their joint shares. Note that each pair of parties can check pairwise consistency of an arbitrarily large number of sharings by comparing a hash of the string consisting of all their joint shares.

4.1 Local Operations

Linear operations. Let $[\![x]\!]_d$ and $[\![y]\!]_d$ be two consistent sharings and let $\alpha \in R$ be some public constant. We define the following operations:

- $[\![x]\!]_d + [\![y]\!]_d$: let $\vec{x}_i$ and $\vec{y}_i$ be the two vectors of shares held by P_i. Then, P_i performs point-wise addition between the two vectors and stores the result as its output.
- $\alpha \cdot [\![x]\!]_d$: let $\vec{x}_i$ be the vector of shares held by P_i. Then, P_i multiplies each component in $\vec{x}_i$ with α and store the result as its output.
- $\alpha + [\![x]\!]_d$: One pre-determined subset of parties T with $|T| = n-d$ which holds x_T define $x_T \leftarrow x_T + \alpha$. The other $\lambda - 1$ shares (where $\lambda = \binom{n}{d}$) remains the same.

The next claim is straight-forward given the definitions of the operations:

Claim. For every $[\![x]\!]_d$ and $[\![y]\!]_d$ and a constant $\alpha \in R$ it holds: (i) $[\![x+y]\!]_d = [\![x]\!]_d + [\![y]\!]_d$; (ii) $[\![\alpha \cdot x]\!]_d = \alpha \cdot [\![x]\!]_d$; (iii) $[\![\alpha + x]\!]_d = \alpha + [\![x]\!]_d$.

Multiplication. We next show two local operations for multiplying two shared inputs x and y, in order to generate a sharing of $x \cdot y$, but with *a higher threshold*. The first operation, which we denote by $[\![x]\!]_d \cdot [\![y]\!]_d$, aims to generate $[\![x \cdot y]\!]_{2d}$. The second operation, which we denote by $[\![x]\!]_d \odot_U [\![y]\!]_d$, aims to compute an *additive* sharing of $x \cdot y$ across a pre-determined subset of parties $U \subseteq \mathcal{P}$ of size $n-d$. We denote such a sharing by $\langle x \cdot y \rangle^U$.

Recall that $x = x_1 + \cdots + x_\lambda$ and $y = y_1 + \cdots + y_\lambda$. It follows that $x \cdot y = \sum_{j \in [\lambda]} x_j \cdot \sum_{k \in [\lambda]} y_k = \sum_{j,k \in [\lambda]} x_j \cdot y_k$. This implies that in order to locally generate a sharing of $x \cdot y$, we need each product $x_j \cdot y_k$ to be known by a set of parties of a sufficient size, where the set's size is determined by the desired threshold. In our setting of $d < \frac{n}{3}$, this indeed holds and utilized in the following two procedures:

- $[\![x]\!]_d \cdot [\![y]\!]_d$:
  - For each $T \subset \mathcal{P}$ such that $|T| = n - 2d$: The parties in T initialize $z_T := 0$.
  - For each pair x_j, y_k that are known to a set of parties $S \in \mathcal{P}$: let $T \subset S$ be the subset containing the first $n-2d$ parties in S and let $q = \binom{|S|}{n-2d}$.
    Then, the parties in T set: $z_T \leftarrow z_T + q \cdot (x_j \cdot y_k)$.
    For each $T' \subset S$ with $|T'| = n-2d$ and $T' \neq T$ set: $z_{T'} \leftarrow z_{T'} - (x_j \cdot y_k)$.
  - Each party P_i sets $\vec{z}_i$ to be the tuple of all z_T for which $P_i \in T$ with $|T| = n-2d$, and stores it as its output.
- $[\![x]\!]_d \odot_U [\![y]\!]_d$:
  - Each party $P_i \in U$ initializes $z_i := 0$.
  - For each pair of x_j and y_k, let $T \subset U$ be the set of parties that holds both x_j and y_k and let P_ℓ be the party with the smallest index in T. Then, P_ℓ sets: $z_\ell \leftarrow z_\ell + |T| \cdot (x_j \cdot y_k)$, whereas each $P_u \in T$ with $u \neq \ell$ sets: $z_u \leftarrow z_u - (x_j \cdot y_k)$.
  - Each party $P_i \in U$ stores z_i as its output.

Claim. Let $d \in \mathbb{N}$ be such that $n > 3d$ and let $U \subseteq \mathcal{P}$. If $|U| \geq 2d+1$, then for every two sharings $[\![x]\!]_d$ and $[\![y]\!]_d$ it holds: (i) $[\![x \cdot y]\!]_{2d} = [\![x]\!]_d \cdot [\![y]\!]_d$; (ii) $\langle x \cdot y \rangle^U = [\![x]\!]_d \odot_U [\![y]\!]_d$.

Proof: We first show that each pair x_j and y_k is known by a set of size $\geq n-2d$. Recall that each input share is held by a set of $n-d$ parties. Let T be the subset of parties who hold x_j and let S be the subset of parties who hold y_k. The set of parties which hold both x_j and y_k is thus $S \cap T$. Recall that $|S \cap T| = |S| + |T| - |S \cup T|$ and so $|S \cap T| = 2(n-d) - |S \cup T|$. Since $|S \cup T| \leq n$, we have $|S \cap T| \geq 2(n-d) - n = n - 2d$ as required.

To prove (i), we first show that $\sum_{T\subset\mathcal{P}:\ |T|=n-2d} z_T = x\cdot y = \sum_{j,k\in[\lambda]} x_j\cdot y_k$. This follows since in the procedure, for each set $S\subset\mathcal{P}$ of parties who know both x_j and y_k, one subset of $n-2d$ adds $\binom{|S|}{n-2d}\cdot(x_j\cdot y_k)$, whereas the remaining $\binom{|S|}{n-2d}-1$ subsets of size $n-2d$ subtract $x_j\cdot y_k$. Thus, overall, $x_j\cdot y_k$ is added exactly once to the summation $\sum_{T\subset\mathcal{P}:\ |T|=n-2d} z_T$. Next, we argue that any adversary controlling d parties does not "gain" information in the procedure, meaning that it learns no additional information on shares held by honest parties only. To see this, observe that by definition, the adversary is missing one share $x_{\hat{j}}$ of x and one share $y_{\hat{k}}$ of y. It thus suffices to show that each share z_T held by a subset T of only honest parties, contains the product $x_{\hat{j}}\cdot y_{\hat{k}}$. These easily follows from the fact that in the procedure, all subsets of size $n-2d$ of honest parties (there are $\binom{n-d}{n-2d}$ such subsets) add it to their joint share.

To prove (ii), we first claim that $\sum_{i|P_i\in U} z_i = x\cdot y = \sum_{j,k\in[\lambda]} x_j\cdot y_k$. Note that each $x_j\cdot y_k$, which is known by a subset T of parties, is added $|T|$ times by one party in T and subtracted by the $|T|-1$ other parties in T. Overall, it is added once to the summation $\sum_{i=1}^{n} z_i$ as required. It remains to show that each pair x_j and y_k is known to at least one party in U. Assume in contradiction that this is not true. This implies that there exists a pair known to a subset of parties T, which is not included in U, i.e., $U\cap T=\emptyset$. However, $|U|\geq 2d+1$ and, as shown above, $|T|\geq n-2d$, and so we have $|U\cup T|>n$. Next, note that every honest party adds/subtracts $x_{\hat{j}}\cdot y_{\hat{k}}$, where $x_{\hat{j}}$ and $y_{\hat{k}}$ are the shares of x and y held by honest parties only, to its share z_i. Thus, any adversary controlling d parties learn no information on the honest parties' shares. ■

4.2 Non-Interactive Random Secret Generation

We next show how to generate correlated randomness required by our protocol, without any interaction, but a short set-up step. Let $\mathcal{F}=\{F_k \mid k\in\{0,1\}^\kappa, F_k:\{0,1\}^\kappa\to R\}$ be a family of pseudo-random functions. The parties work as follows:

From $[\![k]\!]_d$ to any number of $[\![r]\!]_d$. As shown in [26], given $[\![k]\!]_d$, the parties generate the ℓth random sharing $[\![r_\ell]\!]_d$, by having each subset T holding k_T, set its share to be $r_\ell^T = F_{k_T}(\ell)$.

The ideal functionality $\mathcal{F}_{\text{coin}}$. Let $\mathcal{F}_{\text{coin}}$ be an ideal functionality that hands the parties fresh random coins. It can be securely realized by having the parties compute a new random sharing $[\![r]\!]_d$ and then open it by running $\mathsf{reconstruct}([\![r]\!]_d, i)$ for each $i\in[n]$. To generate any numbers of random coins with a constant cost, the parties can use the above procedure to generate a new key k from which all the required randomness is derived using a pseudo-random function F_k.

$\mathcal{F}_{\text{zeroShare}}$ - Generating any number of $[\![0]\!]_{2d}$. In our protocol, we will need random sharings of 0 with threshold $2d$, which will be used to randomize given sharings. We show how the parties can generate any number of $[\![0]\!]_{2d}$ from a single sharing $[\![k]\!]_d$ without any interaction. To generate the ℓth $[\![0]\!]_{2d}$ sharing, the parties work in the following way:

1. Each subset $S\in\mathcal{P}$ of size $n-2d$ initializes $r_S=0$.
2. For every subset $T\in\mathcal{P}$ of size $n-d$, holding k_T:
   Let $\theta=\binom{n-d}{n-2d}$ and let $S^1,S^2,\ldots,S^\theta$ be all subsets of size $n-2d$ in T. Then:
   For each $j\in\{2,\ldots,\theta\}$, the parties in S^j set: $r_{S^j}\leftarrow r_{S^j}-F_{k_T}(\ell\|j)$.
   The parties in subset S^1 set: $r_{S^1}\leftarrow r_{S^1}+\sum_{j=2}^{\theta} F_{k_T}(\ell\|j)$.
3. Each party P_i outputs a vector all r_S for which $P_i\in S$.

Note that each $F_{k_T}(\ell\|j)$ is added once and subtracted once, and so overall $\sum_{S\in\mathcal{P}:|S|=n-2d} r_S=0$ as required.

To prove security, let $\mathcal{F}_{\text{zeroShare}}$ be an ideal functionality that receives from the adversary a share for each subset of parties of size $n-2d$ that contains corrupted parties, and then chooses random shares for the remaining subsets (which contain honest parties only), under the constraint that all shares will sum to 0. Then, $\mathcal{F}_{\text{zeroShare}}$ sends the honest parties their shares. We thus have the following

Lemma 2. *If F_k is a pseudo-random function, then our protocol as described in the text, computationally computes $\mathcal{F}_{\text{zeroShare}}$ in the presence of any malicious adversary controlling up to d parties, where $d < \frac{n}{3}$.*

Proof: Let $\mathcal{S}$ be the ideal world adversary and $\mathcal{A}$ be the real world adversary. Since $\mathcal{A}$ controls at most d parties and $3d+1 \leq n$, it follows that $d < n-d$, and so in each subset of size $n-d$ there is at least one honest party. This implies that $\mathcal{S}$ knows all the distributed keys held by corrupted parties (after the initial setup step, where $\mathcal{S}$ interacts with $\mathcal{A}$, while playing the role of the honest parties) and so can compute the corrupted parties' shares and send them to $\mathcal{F}_{\text{zeroShare}}$. To prove the lemma, we thus need to show that the output shares held by subsets containing honest parties only are indistinguishable in the ideal and real world execution. Observe that in the ideal execution, they are chosen randomly under the constraint that all shares sum up to 0. Note that there are $n-d$ honest parties and so $\binom{n-d}{n-2d}$ shares known to honest parties only. For an adversary controlling d parties, this implies that there are $\binom{n-d}{n-2d} - 1$ output shares that look completely random to $\mathcal{A}$. In contrast, in the real world execution, the share held by each subset S of size $n-2d$ containing only honest parties, is a linear combination of PRF invocations with keys of its supersets of size $n-d$. Note that there is one set of size $n-d$ containing only honest parties, and thus one key which is unknown to $\mathcal{A}$. In the second step of our protocol, this key is used to compute $\binom{n-d}{n-2d} - 1$ *new* shares (because the index j of each subset is taken as an input to F). Thus, if F is a pseudorandom function, a computationally bounded adversary will not be able to distinguish between the shares held by the honest parties only in the ideal execution and their shares in the real execution. This completes the proof. ■

$\mathcal{F}_{\text{corRand}}$ - Generating any number of $(\llbracket r \rrbracket_d, \langle r \rangle^U)$ for a pseudo-random $r \in R$ and $U \subset \mathcal{P}$ such that $|U| = 2d+1$ Recall that $\langle r \rangle^U$ is an additive sharing of r across the parties in U. To obtain pairs of sharings of the same r, the traditional approach is to generate two sharings with the two thresholds separately, and then check that the obtained sharings are of the same secret. We use a different approach, where each party in U first chooses its additive share, shares it to all the other parties, and then the parties use these to compute the sharing with threshold d. Besides the fact that it allows us to avoid the need to run a check, we obtain here a property that will be used later: *the additive share of each party in U is robustly shared to the other parties.* Formally, the parties work as follow:

– *Setup step:*

1. Each party $P_i \in U$ chooses a random $k_i \in R$ and shares $\llbracket k_i \rrbracket_d$ to the parties.
2. The parties run pair-wise consistency check. If party P_j finds that the shares held by him and P_k are not the same, then it broadcasts $(\mathsf{inconsistent}, j, k)$. Then, P_i broadcasts all shares that are held by subsets that contain both P_j and P_k.
   (Note that since in this case either P_i, P_k or P_j is corrupted, these shares are anyway known to the adversary, and so publishing them gives the adversary no additional information.)

– *Generating the ℓth pair $(\llbracket r_\ell \rrbracket_d, \langle r_\ell \rangle^U)$:*

1. For each i with $P_i \in U$: the parties compute $\llbracket r_{\ell,i} \rrbracket_d$ from $\llbracket k_i \rrbracket_d$ as shown above. Knowing all shares, party P_i computes $r_{\ell,i}$ and sets it as its additive share of r_ℓ.
2. The parties locally compute $\llbracket r_\ell \rrbracket_d = \sum_{P_i \in U} \llbracket r_{\ell,i} \rrbracket_d$.

Observe that $r_\ell = \sum_{P_i \in U} r_i$ and so the parties hold an additive sharing and a d-out-of-n sharing of r_ℓ as required. In addition, as promised above, the additive share r_i of each party $P_i \in U$ is shared to the other parties via a d-out-of-n secret sharing. This property will be used later in our protocol.

The $\mathcal{F}_{\text{corRand}}$ ideal functionality. In our protocol, each time the parties will need correlated randomness from the type defined above, they will call the $\mathcal{F}_{\text{corRand}}$ ideal functionality defined in Functionality 1. The functionality $\mathcal{F}_{\text{corRand}}$ lets the adversary choose the shares of the corrupted parties, and then chooses random share for honest parties, under the constraint that the same secret r is stored in $\langle r \rangle^U$ and $\llbracket r \rrbracket_d$.

FUNCTIONALITY 1 (The $\mathcal{F}_{\text{corRand}}$ ideal functionality)

The $\mathcal{F}_{\text{corRand}}$ ideal functionality works with an ideal world adversary $\mathcal{S}$ and honest parties. Let $I \subset [n]$ with $|I| \leq d$, be the set of the corrupted parties' indices, and $H = [n] \setminus I$ be the set of the honest parties' indices. Finally, let $U \in P$ be a predetermined set of parties.

1. $\mathcal{F}_{\text{corRand}}$ receives $\{r_i\}_{i \in I: P_i \in U}$ and $\{r_T\}_{T \subset \mathcal{P}:\ (|T|=n-d) \wedge (\exists i \in I: P_i \in T)}$ from $\mathcal{S}$.
2. $\mathcal{F}_{\text{corRand}}$ chooses a random $r_j \in R$ for each $j \in H$ such that $P_j \in U$, and sets $r = \sum_{k=1}^{n} r_k$. Then, it chooses a random $r_T \in R$ for each $T \subset \mathcal{P}$ with $|T| = n - d$ that contains only honest parties, under the constraint that $r = \sum_{T \subset \mathcal{P}:\ |T|=n-d} r_T$.
3. For each honest party P_j, $\mathcal{F}_{\text{corRand}}$ hands $\{r_T\}_{P_j \in T}$ to P_j and, if $P_j \in U$, hands also r_j to P_j.

Lemma 3. *If F_k is a pseudo-random function, then our protocol as described in the text, computationally computes $\mathcal{F}_{\text{corRand}}$ in the presence of any malicious adversary controlling up to d parties, where $d < \frac{n}{3}$.*

Proof: Let $\mathcal{S}$ be the ideal world simulator and let $\mathcal{A}$ be the real world adversary. In the simulation, $\mathcal{S}$ interacts with $\mathcal{A}$ by playing the role of the honest parties. $\mathcal{S}$ invokes $\mathcal{A}$ to run the setup step, where for each honest party $P_j \in U$ and each subset of parties T that contains corrupted parties, it hands $\mathcal{A}$ random keys $k_{j,T}$, and receives from $\mathcal{A}$ all the keys $k_{i,T}$ for each corrupted $P_i \in U$ (since by definition each T contains at least one honest party). If pairwise inconsistency exists, then the corresponding key is published.

Note that after the setup step, $\mathcal{S}$ can compute all the corrupted parties' shares. Specifically, it can compute $r_{\ell,i}$ for each corrupted $P_i \in U$ (since it knows all shares in $[\![k_i]\!]_d$), and can compute the corrupted parties' shares in $[\![r_\ell]\!]_d$ (since it can compute the corrupted parties' shares of each $r_{\ell,j}$ for $P_j \in U$).

Thus, to generate the ℓth pair of sharings, $\mathcal{S}$ computes the corrupted parties' shares, sends them to $\mathcal{F}_{\text{corRand}}$ and outputs whatever $\mathcal{A}$ outputs.

It is immediate that $\mathcal{A}$'s view is identical in the ideal and real executions. It remains to show that the honest parties' output is computationally indistinguishable in both executions. Observe that in the ideal execution, their shares are random elements in R, whereas in the real execution, their shares are computed using a pseudo-random function. Specifically, let $q = \binom{|H|}{n-d}$ be the number of sets of size $n-d$ that contain honest parties only. Then, in the real execution, for each $j \in H$ such that $P_j \in U$, there are q shares that are computed using a pseudo-random function with a key that is unknown to the adversary. This implies that the number of pseudo-random shares is $q \cdot |\{j \in H : P_j \in U\}|$. In contrast, in the ideal execution, additive shares are randomly chosen for each r_j with $j \in H$ and $P_j \in U$, and then additional $q-1$ random shares are chosen (once r is defined, the functionality chooses $q-1$ shares, and the remaining share is determined by the constraint $r = \sum_{T \subset \mathcal{P}:\ |T|=n-d} r_T$).

To prove that the output of the honest parties in both executions is computationally indistinguishable, we define the following hybrids:

Hybrid 0. The ideal execution as defined above.

Hybrid 1. Let $\hat{\mathcal{S}}$ be a simulator that plays the role of $\mathcal{F}_{\text{corRand}}$ and interacts with $\mathcal{A}$ as in Hybrid 0, but with the following differences: for each $j \in H$ such that $P_j \in U$, it chooses random shares for each $r_{j,T}$ with T being a subset of only honest parties. Then, it sets: $r_j = \sum_{T \in \mathcal{P}: |T|=n-d} = r_{j,T}$ for each $j \in H$ such that $P_j \in U$, and $r_T = \sum_{k \in [n]: P_k \in U} r_{k,T}$ for each subset T of parties of size $n-d$. Finally, it sets $r = \sum_{k \in [n]: P_k \in U} r_k$. Then, it defines the output of each honest party P_j to be $\{r_T\}_{T \in \mathcal{P}:\ |T|=n-d \wedge P_j \in T}$ and, if $P_j \in U$, also r_j.

It is easy to see that Hybrid 0 $\equiv$ Hybrid 1, since in both executions the additive shares r_j held by the honest parties and the d-out-of-n shares r_T of r held by subsets of honest parties only are completely random. Note that in Hybrid 1, there are $q \cdot |\{j \in H : P_j \in U\}|$ shares that are chosen randomly.

Hybrid 2. Next, consider an execution, where all the $q \cdot |\{j \in H : P_j \in U\}|$ shares that $\hat{\mathcal{S}}$ have chosen randomly in *Hybrid* 1, are now computed using a pseudo-random function, with a different key $k_{j,T}$ for each $j \in H$ such that $P_j \in U$, and for each subset T of honest parties only and $|T| = n - d$.

Observe that the output of the honest parties in Hybrid 2 is identical to their outputs in a real execution. It thus remains to show that Hybrid 1 $\stackrel{c}{\equiv}$ Hybrid 2. This can be easily proven via a straight-forward reduction to a computationally-bounded distinguisher $\mathcal{D}$ which is given an access to $q \cdot |H|$ oracles that are either random or pseudo-random functions. The distinguisher $\mathcal{D}$ follows the instructions of simulator $\hat{\mathcal{S}}$, while for each share chosen in the simulation, $\mathcal{D}$ uses one of its oracles to choose the share. It is easy to see that when

the oracles are random functions, then $\mathcal{D}$ generates outputs that are distributed as in Hybrid 1, whereas if the oracle are pseudo-random functions, then it generates outputs that are distributed as in Hybrid 2. Thus, if it is possible to distinguish between the two hybrids with more than a negligible probability (in the security parameter κ), then it is possible to distinguish between random and pseudo-random functions with the same probability. This completes the proof. ∎

5 Building Blocks

In this section, we outline three sub-protocols that are used in our main protocol: a protocol for multiplying shared inputs which achieves only privacy, a protocol to verify the correctness of many multiplication triples and a protocol for eliminating corrupted parties from the computation.

5.1 Multiplying Two Shared Values - The DN Protocol [31]

The Damgård-Nielsen protocol [31] is the fastest multiplication protocol in the honest majority setting, known to this date. The "text-book" version of this protocol, requires the parties to prepare in advance a pair of random sharings $[\![r]\!]_d, [\![r]\!]_{2d}$. Then, in order to multiply $[\![x]\!]_d$ and $[\![x]\!]_d$, the parties locally compute $[\![x]\!]_d \cdot [\![y]\!]_d - [\![r]\!]_{2d}$ and send their shares to P_1 (or any other designated party). Party P_1 reconstructs $xy - r$ and sends it back to the parties. Then, the parties locally compute $[\![x \cdot y]\!]_d = [\![r]\!]_d + (x \cdot y - r)$.

While the protocol was constructed and used throughout the years for Shamir's secret sharing scheme, a simple observation made by [14] is that all operations required by this protocol can be carried-out also when using replicated secret sharing.

Achieving privacy in the presence of malicious adversaries. The text-book version of the DN protocol described above is semi-honest secure. A somewhat surprising finding by [39] is that it actually *does not* achieve even privacy in the presence of malicious adversaries when $d < n/3$. In particular, a malicious P_1 can learn intermediate values. The attack is carried-out over two gates in two preceding layers. Assume that the parties need to multiply $[\![x]\!]_d$ and $[\![y]\!]_d$. Then, upon receiving the messages from the parties and computing $x \cdot y - r$, party P_1 sends the correct value to all parties, except for one honest party, say P_n, to whom it sends $x \cdot y - r + 1$. This implies that P_n now holds an incorrect share of the output $x \cdot y$. Next, assume that in the next layer $x \cdot y$ is being multiplied with w. The crux of the attack is that any $n - 2d$ shares of $xy \cdot w - r'$ (where r' is the random secret mask used in this gate) determine deterministically the remaining shares. Thus, upon holding $n - 2d$ shares of $xy \cdot w - r'$, which do not include shares held by P_n, party P_1 can compute the correct shares of P_n. However, P_n will send shares of $(xy + \epsilon) \cdot w - r'$. Thus, by taking the difference between the actual share received from P_n and the share that should have been sent, P_1 can learn private information about P_n's shares of w.

As can be seen from the above description, the main reason behind the attack is the fact that the random masking sharing $[\![r]\!]_{2d}$ has redundancy, allowing P_1 to use $n - 2d$ shares to compute the remaining $2d$ shares. Thus, a simple way to prevent this attack is to use a mask that is additively shared between the parties. To reduce communication, it suffices to use an additive sharing across $2d + 1$ parties (including P_1) only. This means that the parties need to prepare a pair of sharings $[\![r]\!]_d, \langle r \rangle^U$ for each multiplication, where U is a set of $2d + 1$ parties which includes P_1, and locally compute a sharing $\langle x \cdot y \rangle^U$ which is opened towards P_1. Fortunately, as shown in Section 4.2, we are able to generate this type of correlated randomness. In addition, replicated secret sharing allows computing $\langle x \cdot y - r \rangle^U$ given $[\![x]\!]_d, [\![y]\!]_d$ and $\langle r \rangle^U$, by taking $[\![x]\!]_d \odot_U [\![y]\!]_d$ (see the definition of this operation in Section 4.1) which yields $\langle x \cdot y \rangle^U$, and then subtracting $\langle r \rangle^U$.

Reducing communication. A simple optimization to the DN protocol that we can apply when using replicated secret sharing, is letting P_1 send $x \cdot y - r$ to only *one* subset of parties of size $n - d$. This suffices since adding $x \cdot y - r$ to $[\![r]\!]_d$, is by definition (see Section 4.1) carried-out by having one subset adding $xy - r$ to their share of r. If this subset includes P_1, then it follows that P_1 needs to send $n - d - 1$ ring elements in the second round of the protocol. Overall, the number of elements sent in the protocol is $2d + n - d - 1 = n + d - 1$, and so per party the cost is $1 + \frac{d-1}{n}$ sent ring elements. For $d = 1$, this yields 1 ring element per party, and in general, for $d < n/3$ this is bounded by $1\frac{1}{3}$ elements per party.

PROTOCOL 2 (The Optimized DN Multiplication Protocol)
Let U be the set $\{P_1, \ldots, P_{2d+1}\}$.

- **Inputs:** The parties hold $[\![x]\!]_d$, $[\![y]\!]_d$
- **Set up:** The parties call $\mathcal{F}_{\text{corRand}}$ to obtain $([\![r]\!]_d, \langle r \rangle^U)$.
- **The protocol:**
  1. The parties in U locally compute $\langle x \cdot y - r \rangle^U = [\![x]\!]_d \odot_U [\![y]\!]_d - \langle r \rangle^U$ and send the result to P_1.
  2. P_1 reconstructs $xy - r$. Let $T \in \mathcal{P}$ be a predetermined subset of size $n - d$ such that $P_1 \in T$. Then, P_1 sends $xy - r$ to the parties in T.
  3. The parties compute $[\![z]\!]_d = [\![r]\!]_d + xy - r$ and store $[\![z]\!]_d$ as the output.

Formal description. We present a formal description of the protocol in Protocol 2.

The fact that this protocol is private in the presence of malicious adversaries controlling up to d parties, follows easily from the fact that each message sent by an honest party is masked by a new independent random additive mask. Formally, this means that the view of the adversary during the execution have the same distribution, regardless of the honest parties' inputs. Let Π_{priv} be a protocol to compute a circuit C, where each party shares its input to the other parties, and then the parties traverse over the circuit with topological order, computing multiplication gates using Protocol 2. Let $\mathsf{view}^f_{\mathcal{A},\Pi_{\text{priv}},I}(\vec{v})$ be the view of an adversary $\mathcal{A}$ (i.e., its randomness, inputs, incoming and outgoing messages during the execution) controlling a subset I, when computing a functionality f using Π_{priv}, over a vector of inputs I, *without the output revealing step*. We thus have the following:

Proposition 1. *Let f be a n-aty functionality represented by an arithmetic circuit C over a ring R. Then, for every adversary $\mathcal{A}$ controlling a subset of parties $I \in \mathcal{P}$ with $|I| \leq d < \frac{n}{3}$, and for every two vector of inputs $\vec{v}_1, \vec{v}_2$ it holds that $\mathsf{view}^f_{\mathcal{A},\Pi_{\text{priv}},I}(\vec{v}_1) = \mathsf{view}^f_{\mathcal{A},\Pi_{\text{priv}},I}(\vec{v}_2)$*

Remark 1. If the parties hold two vectors of shares $[\![x_1]\!]_d, \ldots, [\![x_m]\!]_d$ and $[\![y_1]\!]_d, \ldots, [\![y_m]\!]_d$ and wish to compute $[\![\sum_{j=1}^m x_j \cdot y_j]\!]_d$, they can do so without calling the multiplication protocol m times. Instead, this can be done at the cost of one *single* multiplication, as follows. The parties can locally compute $\langle \sum_{j=1}^m x_j \cdot y_j - r \rangle^U$ by taking $\sum_{j=1}^m ([\![x_j]\!] \odot_U [\![y_j]\!]) - \langle r \rangle^U$, and send the result to P_1, who reconstructs $\sum_{j=1}^m x_j \cdot y_j - r$, send it to the parties, which locally compute $[\![r]\!]_d + \sum_{j=1}^m x_j \cdot y_j - r$. The verification protocol below can be easily adapted to incorporate this operation.

5.2 Verifying Correctness of Multiplications with Cheating Identification

In this section, we show how the parties can verify that all multiplications were carried out correctly. Our protocol has the property that if someone has cheated, then the parties will detect it with high probability and, in this case, output semi-corrupt pair to eliminate. A pair of parties is called "semi-corrupt" if it contains at least one corrupted party.

The idea behind our protocol is that the parties "compress" the transcript of all multiplication protocols into one single transcript and then verify its correctness. Observe that in our multiplication protocol, all messages are public; the only reason for communication through P_1 is to save communication, and in fact each of the messages could have been sent to all parties. Thus, the first step of our protocol is to agree on the transcript. In this step, the parties sample random coefficients and broadcast a linear combination of the messages they sent and received in all multiplications. If there is conflict between the view of two parties, then a semi-corrupt pair has been found. If all views are consistent, then the parties proceed to the next step, where they verify the correctness of each compressed message. In more details:

Step 1: Agree on the transcript. Let m be the number of multiplications in the circuit. The parties first call $\mathcal{F}_{\text{coin}}$ to receive m random elements $\delta_1, \ldots, \delta_m \in \mathbb{A}$ (this can be done with small constant cost by calling $\mathcal{F}_{\text{coin}}$ to receive a random seed from which all randomness is derived). Then, each party broadcasts a random linear combination of the messages it sent and a random linear combination of the messages it received. Note that each P_i with $i \neq 1$, needs to broadcast one sent message (a linear combination of the messages sent to P_1 in the first round) if it is included in U, and, if it is included in the subset T of parties that receive the message in the second round, broadcast one received message (a linear combination of the messages received from P_1). At the same time, P_1 broadcasts a random linear combination of all messages received from each of the other parties and a random linear combination of the messages it sent in the

second round. If there is an inconsistency between a "compressed" message P_1 claims to send/receive to/from P_i and the "compressed" message P_i claims to receive/send from/to P_1, then (P_1, P_i) is the new semi-corrupt pair. The fact that either P_1 or P_i is corrupted holds, since a contradiction cannot occur between two honest parties. If all messages are consistent, then the parties proceed to the next step with an agreed-upon compressed transcript.

Step 2: Verify each party's message. Next, the parties verify the correctness of each party's message. Observe that P_1's message is computed by summing all messages received from the other parties and his own first round's message (since P_1 sees an additive sharing of the masked output). Thus, given the message sent from P_1 in the second round, the parties can compute implicitly the message P_1 would send in the first round. This implies that the verification in this step is reduced to checking that the local computation of $[\![x]\!]_d \odot_U [\![y]\!]_d - \langle r \rangle^U$, performed by each of the parties in U, is correct. Recall that in this computation, each party performs a local computation over its shares of x and y, and then subtracts its additive share of r. Looking at the definition of the operation $[\![x]\!]_d \odot_U [\![y]\!]_d$ from Section 4.1, we observe that in this computation, the parties compute a *linear combination* of their shares. Let $\gamma = \binom{n-1}{n-d-1}$ be the number of shares held by each party. Denote the shares held by P_i by $x^i_1, \ldots, x^i_\gamma$ and $y^i_1, \ldots, y^i_\gamma$ and r^i. Then, party P_i carries-out the computation $\sum_{k=1}^{\gamma} \sum_{j=1}^{\gamma} (\alpha_{k,j} \cdot x^i_k \cdot y^i_j) - r^i$, where $\alpha_{k,j}$ is a public known coefficient (see Section 4.1).

Next, recalling that in our secret sharing scheme, each share is held by $n-d$ parties, we define $[\![x^i_k]\!]_d$ (and likewise $[\![y^i_j]\!]_d$) to be a consistent d-out-of-n secret sharing of x^i_k in the following way: The subset T of parties which know x^i_k set $x^i_{k,T} = x^i_k$, whereas the other subsets define their share to be 0. In addition, recall that in Section 4.2, we generated the correlated randomness in a special way, such that each party's additive share r^i is also secret shared across the parties in a robust way. We thus have the following:

Fact 3 *In our multiplication protocol (Protocol 2), the message each party sends in the first round, is a result of a degree-2 computation over inputs that are shared across the parties via a consistent d-out-of-n secret sharing.*

Relying on Fact 3, we can thus ask the parties to jointly compute a $2d$-out-of-n secret sharing of P_i's additive share of $x \cdot y - r$, and then compute a $2d$-out-of-n secret sharing of the "compressed message" obtained in the previous step. That is, for each $i \in U$, the parties will compute

$$[\![\mathsf{msg}^i]\!]_{2d} = \sum_{\ell=1}^{m} \delta_\ell \cdot [\![(x_\ell \cdot y_\ell - r_\ell)^i]\!]_{2d} \tag{1}$$
$$= \sum_{\ell=1}^{m} \delta_\ell \cdot \left(\sum_{k=1}^{\gamma} \sum_{j=1}^{\gamma} (\alpha_{k,j} \cdot [\![x^i_{\ell,k}]\!]_d \cdot [\![y^i_{\ell,j}]\!]_d) - [\![r^i_\ell]\!]_d \right) + [\![0]\!]_{2d}$$

where $[\![0]\!]_{2d}$ is a secret sharing of 0 which is added to randomize the parties' shares (and is handed to the parties by $\mathcal{F}_{\text{zeroShare}}$). Note that this computation is completely non-interactive: the parties can locally compute $[\![x^i_k \cdot y^i_j]\!]_{2d} = [\![x^i_k]\!]_d \cdot [\![y^i_j]\!]_d$ and $[\![r^i]\!]_{2d} = [\![r^i]\!]_d \cdot [\![1]\!]_d$ (where $[\![1]\!]_d$ is some known sharing of 1) as defined in Section 4.1, and then locally perform addition and multiplication with the public constants. It remains to open the shared secret and check whether it equals to the additive share P_i sent. Since $3d + 1 \leq n$, then $n - 2d \geq d + 1$, which implies that in each subset of $n - 2d$ there is at least one honest party. This means that by sending incorrect shares, the corrupted parties can only cause pair-wise inconsistency, which will result with a semi-corrupt pair. Thus, if all shares are consistent, the parties will hold the correct msg^i. Then, the parties can compare it to the value obtained in the first step and see whether P_i have cheated or not. We thus obtain the following protocol for Step 2:

1. For each $i \in U$:
   (a) The parties locally compute $[\![\mathsf{msg}^i]\!]_{2d}$ via Eq. (1).
   (b) For each $j \in [n]$, the parties in each subset T where $|T| = n - 2 \cdot d$ and $P_j \notin T$ send their shares of msg^i to P_j.[9]
2. If party P_j received contradicting shares in any of the n executions in the previous step, then it sets $\mathsf{cons}^j = 1$. Otherwise, it sets $\mathsf{cons}^j = 0$. Then, it broadcasts cons^j to the other parties.
3. Upon receiving cons^j from all the parties:

[9] This can be optimized by asking only one party in each set T to send the share to P_j and the rest send hashes of their shares for all $i \in [n]$.

– If $\forall j : \mathsf{cons}^j = 0$:
The parties reconstruct msg^i for each $i \in [n]$. Let msg'^i be the compressed message of P_i agreed upon in the first step.
  - If $\forall i : \mathsf{msg}'^i = \mathsf{msg}^i$: the parties output accept.
  - If $\exists i : \mathsf{msg}'^i \neq \mathsf{msg}^i$: Let i be the *largest* such that $\mathsf{msg}'^i \neq \mathsf{msg}^i$, and let j the smallest index of party such that $i \neq j$. Then, the parties output $\mathsf{reject}, (i, j)$[10]

– If $\exists j : \mathsf{cons}^j = 1$:
Let j be the smallest index for which $\mathsf{cons}^j = 1$. Let P_u and P_w be the first pair of parties who sent contradicting shares v_T^u and v_T^w to P_j, with T being the first subset for which $P_u, P_w \in T$ and $v_T^u \neq v_T^w$, and let $i \in [n]$ be the index of the execution in Step 1 where the inconsistency occurred. Then:
  (a) Party P_j broadcasts $(T, i, u, w, v_T^u, v_T^w)$.
  (b) Party P_u broadcasts $\tilde{v}_T^u$ and party P_w broadcasts $\tilde{v}_T^w$.
  (c) If $\tilde{v}_T^u \neq \tilde{v}_T^w$, then the parties output $\mathsf{reject}, (u, w)$.
  Otherwise, if $v_T^u \neq \tilde{v}_T^u$, then the parties output $\mathsf{reject}, (j, u)$.
  Otherwise, the parties output $\mathsf{reject}, (j, w)$.

Cheating probability. The only way that the protocol can end with the parties outputting accept even though a corrupted party has cheated in the multiplication protocol, is if a random linear combination of incorrect messages yields the same value as a random linear combination of the correct messages. Using Lemma 1, we see that this probability is bounded by $\frac{1}{\omega_R}$. Thus, to obtain a statistical security of s bits, the parties will repeat the above protocol $\lceil \frac{s}{\log \omega_R} \rceil$ times.

Security. The only security concern in the above protocol, is that something can be learned from the additive shares of $(x \cdot y - r)^i$. This is prevented by randomizing the sharing when adding $[\![0]\!]_{2d}$. Formally, we define the ideal functionality $\mathcal{F}_{\mathsf{checkTrans}}$ in Functionality 4, which receives the sent/received messages from all parties, and the inputs and randomness of the honest parties. The latter suffices, in our setting of two-thirds honest majority, to compute all the messages that corrupted parties should have sent in the protocol. Thus, $\mathcal{F}_{\mathsf{checkTrans}}$ can find whether any party have cheated and sent incorrect messages. In case of cheating, it asks the real-world adversary to provide a pair of parties, such that at least one of them is corrupted, which is then output to the honest parties. Note that also in the case that no one cheated in the multiplication protocol, $\mathcal{S}$ is allowed to change the output to reject, but then it must provide also a semi-corrupt pair to eliminate. This captures the case when the corrupted parties send incorrect shares in the second step of our protocol, causing the verification to fail.

FUNCTIONALITY 4 ($\mathcal{F}_{\mathsf{checkTrans}}$- Verification of messages with Cheating Identification)
Let $\mathcal{S}$ be the ideal-world adversary controlling a subset $< n/3$ of corrupted parties.

1. $\mathcal{F}_{\mathsf{checkTrans}}$ receives from the honest parties their inputs, randomness and sent/received messages. These are used to compute the inputs and randomness of the corrupted parties.
2. $\mathcal{F}_{\mathsf{checkTrans}}$ sends $\mathcal{S}$ the corrupted parties' inputs and randomness and all messages the honest parties claimed to send/receive.
3. Upon receiving from $\mathcal{S}$ all messages the corrupted parties claim to send/receive to/from honest parties:
   (a) If there is a contradiction between the message a corrupted party P_i claim to send/receive to/from an honest P_j, then $\mathcal{F}_{\mathsf{checkTrans}}$ sends $\mathsf{reject}, (i, j)$ to the honest parties.
   (b) Otherwise, $\mathcal{F}_{\mathsf{checkTrans}}$ checks that all messages sent from corrupted parties are correct given their inputs and randomness. If it holds, then $\mathcal{F}_{\mathsf{checkTrans}}$ sends accept to $\mathcal{S}$. Otherwise, $\mathcal{F}_{\mathsf{checkTrans}}$ sends reject to $\mathcal{S}$.
   – In the former case, $\mathcal{S}$ send back either accept or $\mathsf{reject}, (i, j)$ to the $\mathcal{F}_{\mathsf{checkTrans}}$, such that either P_i or P_j (or both) are corrupt. This is then handed to the honest parties.
   – In the latter case, $\mathcal{S}$ must send back a pair (i, j) such that either P_i or P_j (or both) are corrupt. Then, $\mathcal{F}_{\mathsf{checkTrans}}$ sends $\mathsf{reject}, (i, j)$ to the honest parties.

To simulate the protocol, note that by the definition of $\mathcal{F}_{\mathsf{checkTrans}}$, the simulator $\mathcal{S}$ receives from the trusted party computing $\mathcal{F}_{\mathsf{checkTrans}}$ all the corrupted parties' inputs and randomness, as well as all the

[10] Note that here we know that P_i is corrupted and we could essentially remove only him. Nevertheless, since removing a pair of parties (with one of them being corrupt) maintains the threshold while reducing the overall communication, this is preferable.

messages sent/received in the protocol by the honest parties. Thus, $\mathcal{S}$ can perfectly simulate the first step. In the second step, it can perfectly simulate the messages sent when verifying msg^i for all i such that P_i is corrupted. This holds since all messages are a function of P_i's inputs and randomness which are known to $\mathcal{S}$, and shares distributed by $\mathcal{F}_{\text{zeroShare}}$, which is played by $\mathcal{S}$. Finally, when simulating the opening of msg^i for an honest P_i, the simulator $\mathcal{S}$ chooses random shares for the subsets containing honest parties only, under the constraint that all shares will open to msg^i and given the corrupted parties' shares which are known to $\mathcal{S}$. Since for honest parties it holds that $\mathsf{msg}'^i = \mathsf{msg}^i$, and since all shares are randomized before opening, it holds that the honest parties' shares in the simulation are indistinguishable from their shares in the real execution. The only difference between the simulation and real execution is the case when $\mathcal{F}_{\mathsf{checkTrans}}$ decides to reject, and the honest parties in the simulation accept. This happens when there is an incorrect message which is not detected since the random linear combination yields the same result as if correct messages were sent. As we have seen above, this can happen with probability 2^{-s} when repeating the process sufficient number of times.

We thus obtain the following:

Lemma 4. *Our protocol, as described in the text above, computationally computes $\mathcal{F}_{\mathsf{checkTrans}}$ in the $(\mathcal{F}_{\text{coin}}, \mathcal{F}_{\text{zeroShare}})$-hybrid model, in the presence of malicious adversaries controlling d parties, where $n > 3d$, with statistical error 2^{-s}.*

Communication cost. Assume that the elements in R are represented using ℓ bits. In the first step, each party broadcasts, on average, 2 messages, and so 2ℓ bits. In the second step, each party sends $(2d+1) \cdot \binom{n-1}{2d} \cdot 2d$ elements to the other parties and broadcasts one bit, but if only one party in each set sends the shares and the other parties send a single hash, the factor $2d+1$ can be removed, asymptotically. With this in mind, overall, each party sends approximately $\binom{n-1}{2d} \cdot 2d \cdot \ell + 2\ell \cdot |BC|$ bits, where $|BC|$ is the cost of broadcasting one bit per party. When repeating the protocol s times, the above is multiplied by s. Note that the cost is completely independent from the number of verified multiplications.

5.3 Parties' Elimination and Recovery

In the previous section, we showed how to find a semi-corrupt pair. Once such a pair is found, the parties need to remove this pair from the computation in a secure way. After removing two parties where at least one is corrupt the new set of parties contains $n' = n - 2$ participants, with $d' = d - 1$ being corrupt. This new set of parameters preserves the required bound $d' < n'/3$, since $d < n/3$. Removing a semi-corrupt pair is fairly standard technique in Shamir secret-sharing-based protocols (e.g. [7]), and it can be extended to replicated secret-sharing in a reasonably straightforward manner, which we discuss in detail below.

Removing two parties, from which one is guaranteed to be corrupt, requires the parties to move from a d-out-of-n secret sharing to a $(d-1)$-out-of-$(n-2)$ secret sharing of each value on the output layer of the previous segment (which is the last state that was verified and approved by the parties).

For simplicity let us assume that all the values on the output layer of the previous segment are outputs of multiplication gates (otherwise, they are a linear function of multiplication's outputs). Recall that in our DN-style multiplication protocol, the output is computed by taking $[\![r]\!]_d + (xy - r)$. Recall also that this step is carried-out by having one subset of $n - d$ parties receive $xy - r$ from P_1 and add it to their share of r. To successfully eliminate a semi-corrupt pair, the parties thus need to prepare $[\![r]\!]_{d-1}$ and then let one subset of $n - 2 - (d-1) = n - d - 1$ parties, that do not contain the two eliminated parties, add $xy - r$ to their share of r. To achieve this, we leverage the fact that $xy - r$, as well as each share of r, is held by a subset of $n - d \geq 2d + 1$ parties, which means that among the $n - d$ parties there exists an honest majority.

Assume that P_i and P_j are the parties to eliminate. The parties work as follows:

1. **Random keys update:** For each $i \in [n]$: for each subset $T \in \mathcal{P}$ such that $|T| = n - d$ and $P_i, P_j \in T$, the parties in T send the key $k_{i,T}$ to some party $P_u \notin T$. If P_u received different value for each key, then it chooses the majority value.
   Then, Party P_u uses these keys to compute r_T and adds it to its shares of r.
2. Let $\mathcal{P}' = \mathcal{P} \setminus \{P_i, P_j\}$, $n' = n - 2$ and $d' = d - 1$. Note that after the previous step, each share of r is known to a set of active parties of size at least $n' - d' = n - 2 - (d-1) = n - d - 1$.
3. **From $[\![r]\!]_d$ to $[\![r]\!]_{d-1}$:**
   - For each subset $S \in \mathcal{P}'$ such that $|S| = n' - d'$, the parties in S initialize $r_S := 0$.
   - For each subset $T \in \mathcal{P}$ of size $n - d$:

- *Case 1*: $P_i, P_j \notin T$.
  Note that there are $n-d$ subsets of size $n'-d'$ in each T (since $n'-d' = n-d-1$ and $\binom{n-d}{n-d-1} = n-d$). Then:
  The subset S containing the first $n'-d'$ parties in T sets $r_S \leftarrow r_S + (n-d) \cdot r_T$. The other subsets S of size $n'-d'$ parties set their share to be $r_S \leftarrow r_S - r_T$.
- *Case 2*: $P_i \in T \wedge P_j \notin T$ or $P_j \in T \wedge P_i \notin T$.
  The subset $S = T \setminus \{P_i, P_j\}$ of size $n'-d'$ sets it share to be $r_S \leftarrow r_S + r_T$.
- *Case 3*: $P_i, P_j \in T$.
  Let $S = T \setminus \{P_i, P_j\} \cup \{P_u\}$, where P_u is the party who learned r_T in the first step. Then, the parties in S set: $r_S \leftarrow r_S + r_T$.

4. **Updating a multiplication's output: from $[\![z]\!]_d$ to $[\![z]\!]_{d-1}$.**
   Let $T \in P$ be the set of parties holding $x \cdot y - r$. Then:
   - *Case 1*: $P_i, P_j \notin T$.
     Let S be the set of the first $n'-d'$ parties in T. Then, the parties in S set: $z_S = r_S + xy - r$. For each subset $S' \in \mathcal{P}'$ of size $n'-d'$ with $S' \neq S$, the parties in S set $z_S = r_S$.
   - *Case 2*: $P_i \in T \wedge P_j \notin T$ or $P_j \in T \wedge P_i \notin T$.
     Note that $S = T \setminus \{P_i, P_j\}$ is a set of size $n'-d'$. Thus, the parties in S set $z_S = r_S + xy - r$, whereas for each subset $S' \in \mathcal{P}'$ of size $n'-d'$ with $S' \neq S$, the parties in S' set $z_S = r_S$.
   - *Case 3*: $P_i, P_j \in T$.
     The parties in T send $xy - r$ to some party $P_u \notin T$. If P_u receives different values, then it chooses the majority value. Then, the parties in $S = T \setminus \{P_i, P_j\} \cup \{P_u\}$ set $z_S = r_S + xy - r$, whereas for each subset $S' \in \mathcal{P}'$ of size $n'-d'$ with $S' \neq S$, the parties in S' set $z_S = r_S$.

Communication cost. The above protocol requires interaction in two steps. First, the parties need to send all keys that are known by both P_i and P_j. Note that this cost is constant and does not depend on the size of the circuit. A second source of interaction is the case where both P_i and P_j are in the set of the parties who hold $xy - r$ for an output wire on the output layer of the last segment. Here we have $n-d$ elements that are sent for each output wire. Per party, the communication cost is thus bounded by $\frac{n-d}{n} \cdot W < W$, where W denotes the "width" of the circuit, i.e., the maximal number of multiplication gates that are on the same layer of the circuit. Note that W is *always* smaller than the size of the circuit, and for any "natural" circuit is of sublinear size.

6 Securely Computing Any Functionality Over Rings

We are now ready to present our main protocol to compute arithmetic circuits over the finite ring R. As explained before, the circuit is divided into segments, and each segment is computed separately. That is, the parties compute the segment using our private multiplication protocol, and then call $\mathcal{F}_{\mathsf{checkTrans}}$ to verify the correctness of the computation. If the parties receive accept from $\mathcal{F}_{\mathsf{checkTrans}}$, then they know that the secrets shared on the output layer of the current segment are correct, and so they can proceed to the next segment. Otherwise, they receive a semi-corrupt pair from $\mathcal{F}_{\mathsf{checkTrans}}$ which is removed from the computation. This is carried out by updating the secret sharing of the inputs to the current segment using our elimination and recovery subprotocol. The segment is then recomputed with less parties. More formally:

Input sharing step. At the end of this step, the parties will hold a consistent d-out-of-n secret sharing of each input.

1. The parties set $n' = n$ and $d = t$, where $n = 3t + 1$.
2. For each $i \in [n]$: party P_i distributes $[\![k_i]\!]_d$ to the other parties (the parties run a pairwise inconsistency check for $[\![k_i]\!]_d$. For each share that is inconsistent, P_i broadcasts the share to the parties). Note that this step is carried-out once, and $[\![k_i]\!]_d$ can be used to many computations.
3. For the jth input x_j held by P_i: the parties locally derive $[\![r^i_j]\!]_d$ from $[\![k_i]\!]_d$ as shown in Section 4.2. Then, P_i broadcasts $x_j - r^i_j$ to the parties. Finally, the parties locally compute $[\![x_j]\!]_d = [\![r^i_j]\!] + x_j - r^i_j$.

Computing the next segment. This step begins with the parties holding a consistent secret sharing of the values on the input wires of the segment.

4. The parties compute the segment gate after gate in some predetermined topological order. Linear gates are computed locally and multiplication gates are computed using Protocol 2.

5. The parties send their inputs, randomness and sent/received messages in the execution of all multiplication protocols in the previous step to $\mathcal{F}_{\mathsf{checkTrans}}$. If $\mathcal{F}_{\mathsf{checkTrans}}$ sent accept, then they proceed to the next segment. Otherwise, they receive from $\mathcal{F}_{\mathsf{checkTrans}}$ a pair of parties (P_i, P_j) to eliminate. The parties then run the elimination and recovery subprotocol, set $n' = n' - 2$, $d = d - 1$ and go back to Step 4.

Output reconstruction. At the beginning of this step, the parties hold a d-out-of-n' secret sharing of each output. Then, for each output o_k intended to party P_i, the parties run $\mathsf{reconstruct}([\![o_k]\!]_d, i)$.

Theorem 5. *Let R be a finite ring, let f be a n-party functionality represented by an arithmetic circuit over R, and let $t \in \mathbb{N}$ be a security threshold parameter such that $n = 3t + 1$. Then, our main protocol, as described in the text, computationally computes f in the $(\mathcal{F}_{\mathrm{corRand}}, \mathcal{F}_{\mathsf{checkTrans}})$-hybrid model, in the presence of malicious adversaries controlling up to t parties.*

Proof: Let $\mathcal{S}$ be the ideal world simulator. In the simulation, $\mathcal{S}$ plays the role of the honest parties, and the ideal functionalities $\mathcal{F}_{\mathrm{corRand}}$ and $\mathcal{F}_{\mathsf{checkTrans}}$. The simulation in each of the steps works as follows:

- Input sharing step: In this step, $\mathcal{S}$ sets the input of each honest party to be 0. Then, it follows the instructions of the protocol. Observe that since an honest majority exists, $\mathcal{S}$ receives all the keys that each corrupted party distributes, and thus it can compute the mask r^i_j for the jth input of the corrupted party P_i. This enables $\mathcal{S}$ to extract the inputs of each corrupted party P_i by computing $x^i_j = x^i_j - r^i_j + r^i_j$.
- Segment computation: In this step, $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathrm{corRand}}$ in the multiplication protocol, and so it receives all the corrupted parties' random shares for $[\![r]\!]_d, \langle r \rangle^U$. Then, $\mathcal{S}$ follows the instructions of the protocol playing the role of the honest parties. Since $\mathcal{S}$ knows the inputs' shares and randomness to each gate, it knows whether cheating took place. Finally, $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{checkTrans}}$. If no cheating took place, then it sends accept to $\mathcal{A}$, and otherwise, it sends reject. In the former case, it waits for $\mathcal{A}$ to send back accept which means that the execution proceeds to the next computation, or to repeat the computation by handing $\mathcal{S}$ a semi-corrupt pair to remove. In the latter case, it waits for $\mathcal{A}$ to send him a semi-corrupt pair.
  In the case that a semi-corrupt pair was located and the segment is recomputed, the elimination and recovery subprotocol is executed. In the simulation, $\mathcal{S}$ simply follows the instructions of the protocol while playing the role of the honest parties.
  Observe that since $\mathcal{S}$ knows the corrupted parties' random shares of r and $x \cdot y - r$ for each multiplication gate, then it knows the shares of the corrupted parties on each wire of the circuit.
- Output reconstruction: The simulator $\mathcal{S}$ sends the corrupted parties' inputs to the trusted party computing f, to receive back their outputs. Then, $\mathcal{S}$ replaces the shares held by subsets containing honest parties only, with new random shares that, together with the corrupted parties' shares (known to $\mathcal{S}$), would reconstruct to the output received from the trusted party. Then, $\mathcal{S}$ plays the role of the honest parties sending $\mathcal{A}$ their shares. Finally, $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs.

Observe that the only difference between the simulated and the real execution, is the honest parties' inputs. However, by the secrecy of the secret sharing scheme, this makes no difference in the input sharing step. In the circuit's computation step, by Proposition 1, the view of the corrupted parties is the same regardless of the used input. Thus, the view of $\mathcal{A}$ up to the last step, is distributed the same in both executions. Finally, in the output reconstruction step, by the secrecy of the secret sharing scheme and since $\mathcal{A}$'s view till this step is the same in both executions, it follows that the honest parties' shares of the outputs are identically distributed in both executions. This concludes the proof. ■

7 Performance Study

In this section we study the concrete performance of our protocol. Our goal is to illustrate that, by means of our novel techniques, replicated secret-sharing can be used with reasonable efficiency for more than 3 or 4 parties, which are the traditional settings in which this scheme has been used. To this end, we provide an assessment of the communication and storage requirements of our protocol, for different parameters, in Section 7.1. In addition, we completely[11] implemented our protocol in C++, and in Section 7.2 we discuss in detail the experimental results we have obtained. The source code of our implementation can be found in https://github.com/anderspkd/ccs-DEN22.git.

n	**Share size**	**Communication** **Mult. gate (opt/worst)**	**Checks (opt/worst)**
4	24 B	8 B/16 B	48 B/48 B
7	120 B	9.14 B/27.43 B	480 B/528 B
10	672 B	9.6 B/38.4 B	3.94 KiB/4.45 KiB
13	3.87 KiB	9.85 B/49.23 B	30.94 KiB/35.39 KiB
16	23.46 KiB	10 B/60 B	234.61 KiB/270 KiB
19	145.03 KiB	10.11 B/70.74 B	1.7 MiB/1.96 MiB
22	908.44 KiB	10.18 B/81.45 B	12.42 MiB/14.38 MiB
25	5.61 MiB	10.24 B/92.16 B	89.78 MiB/104.16 MiB
28	35.76 MiB	10.29 B/102.86 B	643.64 MiB/747.8 MiB
31	229.23 MiB	10.32 B/113.55 B	4.48 GiB/5.21 GiB
n	$\binom{n-1}{t} \cdot \ell$	$\ell \cdot (1 + \frac{t-1}{n}) \Big/ \times(1+t)$	$\mathsf{Ch}_n \Big/ + \sum_{\ell=1}^{t} \mathsf{Ch}_{n-3\ell}$

Table 1. Storage and communication complexity (per party) of our protocol for different number of parties $n = 3t + 1$. The bit-size ℓ of each ring element is assumed to be 64 bits. $|BC|$, the cost of broadcasting one bit per party, is taken to be 0 as its contribution is minimal, and the number of repetitions of the check to achieve negligible soundness is assumed to be 1. We consider the case in which the whole circuit is one single segment, so only one check is performed at the end of the execution. We report complexities for the **optimistic** case where no cheating occurs, and for the **worst** case where the adversary repeats the circuit t times (each time with three parties less). This causes a multiplicative overhead of $(1 + t)$ in the complexity per multiplication gate, and an additive overhead of $\sum_{\ell=1}^{t} \mathsf{Ch}_{n-3\ell}$ in the complexity regarding the checks. Here $\mathsf{Ch}_n = \binom{n-1}{2t} \cdot 2t \cdot \ell + 2\ell \cdot |BC|$.

7.1 Storage and Communication Costs

Communication costs. We begin by deriving an expression for the communication complexity of our protocol from Section 6. Let $|C|$ be the size of circuit (measured by the number of multiplication gates). Let $|S|$ be the size of a segment, and let $m = |C|/|S|$ be the number of segments. Recall that the cost of our multiplication protocol is $1 + \frac{d-1}{n}$ ring elements per party, which we upper-bound by $4/3$. Furthermore, let Ch_n be the cost of the multiplication check with n parties, which equals $\binom{n-1}{2t} \cdot 2t$ ring elements, plus the cost of broadcasting two ring elements (recall this overall cost is exponential in n, but independent of the size of the segment). The communication cost of *one segment* in our protocol is $\frac{4}{3} \cdot |S| + \mathsf{Ch}_n$ ring elements per party. In the optimistic case, where all parties act honestly, there are $m = |C|/|S|$ segments executed, which leads to a communication complexity of $\frac{4}{3}|C| + m \cdot \mathsf{Ch}_n$ ring elements per party.

In the case of active cheating, several segments might be executed multiple times. The exact communication complexity in this case depends heavily on where the adversary cheats, and how many times he does so. However, in terms of the worst case it is easy to see that the scenario that leads to the most expensive communication complexity is when the adversary behaves honestly for all segments except for the last one, point in which the adversary misbehaves, making this segment be executed t more times, reducing the number of parties by three in each repetition.[12] As a result, the worst case communication complexity in the event of active cheating by adding, to the optimistic case above, the cost of the t extra repetitions, which is given by $\frac{4}{3}|S|t + \sum_{\ell=1}^{t} \mathsf{Ch}_{n-3\ell}$.

This way, we can write the worst case complexity per party as

$$\frac{4}{3}|C|(1 + \frac{t}{m}) + m \cdot \mathsf{Ch}_n + \sum_{\ell=1}^{t} \mathsf{Ch}_{n-3\ell}.$$

We discuss possible choices of m below.

Concrete complexity for certain parameters. As we have mentioned, certain metrics of our protocol increase exponentially with the number of parties. For instance, the communication complexity of the

[11] Except for some minor steps that are not expected to affect runtimes drastically.

[12] The protocol from Section 6 is described as removing *two* parties in each repetition, but it is easy to see that the bound $t < n/3$ can be preserved while removing *three*, which helps efficiency and results in the four-party case being the base case.

final check, although is independent of the number of multiplications being checked, is exponential in n. In addition, storage complexity, which is directly related to the size of each share, is exponential in n, and this also affects computation involving shares, like locally adding secret-shared values or reconstructing a secret from a given set of shares. However, we recall that a crucial aspect of our protocol is that its communication complexity, apart from the final check, does not grow exponentially with the number of parties, and in fact it is kept constant (per-party); this is in contrast to many existing work that makes use of replicated secret-sharing.

In Table 1 we see the share size, together with the communication cost of each multiplication gate and the final check, per party, for a 64-bit ring and increasing number of parties. We consider one single segment corresponding to the whole circuit, meaning that $m = 1$, and there is only one single check at the end of the execution. We report complexities for both the optimistic case (when there is no cheating) and the worst-case (when the circuit is re-run t more times, each with 3 parties less).

In the optimistic case, the communication cost per multiplication gate (regardless of the number of parties) is kept under 11 bytes, while in the worst case when the circuit is evaluate t more times it goes up by a factor of $(1 + t)\times$. This multiplicative overhead takes a toll when n is moderately large, like for $n = 31$, where it increases the cost per gate from around 10 to 110 bytes, an overhead of $10\times$. Regarding the communication arising from the different checks, in the optimistic case only one check is executed, but in the worst case t more checks must be performed, each with three parties less. Fortunately, from Table 1 we see that this overhead, being *additive*, is quite small with respect to the check in the optimistic case, increasing from around 4.5 to 5.2 gigabytes for $n = 31$, for example.

We see that the communication complexity of the final check grows very fast, even when compared to the share size. However, we stress that this is only executed *once* at the end of the protocol. Depending on the application at hand, this overhead could be considered acceptable with respect to the rest of the computation. To illustrate this we study, for different number of parties, the number of multiplications needed so that the communication complexity involved in their computation *matches* the communication complexity of checking them, which means that the overhead of the final check at this point is 2x, and it approaches 1x as the number of multiplications grow.[13] For moderately large values of n such as $n = 10$, the check costs the same as less than one thousand multiplications, and for larger values like $n = 22$, the check costs the same as a bit over one million multiplication. This grows up to roughly one billion if $n = 31$. A detailed analysis for more values of n appears in Section D.1 in the Appendix. Furthemore, other additional aspects of the communication complexity of our protocol appear in Section D in the Appendix.

Finally, the share size, which grows exponentially with n, is kept in the order of bytes for $n = 4, 7, 10$, kilobytes for $n = 13, 16, 19, 22$, and tens of megabytes for $n = 25, 28$. For $n = 31$, this size reaches around 200 megabytes, which is large when considering that this corresponds to *each* shared value. However, the following optimization can prove to be crucial for reducing the impact of this overhead in practice. As currently described, our protocol requires the parties to store *all* the secret-shared values computed in a given segment, to be able to check them at the verification step. Instead, the parties can sample the random coefficients δ_i used to compress the values to be checked via a linear combination *on the fly*. In a bit more detail, after performing the computation of a given multiplication layer of the circuit (in particular, after the adversary committed to its errors) the parties sample the necessary random values for the given layer, and aggregate these into a small amount of secret-shared values that correspond to these computed in the final verification step. This way, the parties can discard the shares obtained in a given layer (unless they are required for a subsequent step in the computation). A similar approach was also used in the context of MPC with dynamic participants in [23], in order to reduce the number of shared values needed from one round to the next.

On the choice of m. For the results in Table 1 we have chosen $m = 1$, so we regarded the whole circuit as one single segment, and only one check is performed at the end. If this check fails then the entire circuit is re-run, with three parties less. Choosing $m = 1$ leads to the best possible total communication complexity in the optimistic case (which is arguably the scenario more relevant in practice), but the gap between the optimistic and worst-case scenarios is very large. If, instead, it is the goal to minimize this gap, we could take larger values of m. For example, $m = t$ leads to an overhead in the amortized communication

[13] We remark that this only measures the amount of messages sent. We must take into account that the computation of the multiplication gates happens in several rounds, while the check only uses a constant number of rounds, which in practice makes it more efficient to compute even if its communication is the same (or even more) than evaluating several multiplications.

complexity per multiplication gate in the worst-case of only $2\times$ with respect to the optimistic case, but now the latter is more expensive as the check is performed t times, instead of just one.

Different choices of m lead to different performance results, and which one is optimal depends on the expected "amount of cheating". We defer to Section D.2 in the Appendix a more detailed discussion on some concrete values of m and their effect on the communication complexity.

7.2 Experimental Results

We created a proof of concept implementation of the core parts of our contribution, namely the multiplication and check protocols. Our intention is to investigate, in concrete terms, the overhead of our techniques with a varying number of parties and computation sizes. Our implementation can be found alongside this submission, as can all the experimental data we generated and analyze in this section. To the best of our knowledge, our implementation and evaluation constitute the first set of experimental results regarding replicated secret-sharing for an increasing number of parties.

Our implementation was written in C++ and all experiments where run on a single `c5.4xlarge` AWS instance, with each party being executed in a different procedure. We set our experiments in a study a WAN by setting a delay of 100ms and a bandwidth of 100 Mpb/s. We believe that this creates an experimental setup that is easier to replicate. We choose a prime field of approximately 64 bits (so in particular, the final check only needs to be repeated once for a statistical security of $\approx 2^{-64}$).

Experiments Recall that, when the number of parties increases, storage, local computation and the communication of the final check increase *exponentially*, but the communication complexity per multiplication gate remains *constant*. As a result, one might expect our protocol to *not* be very competitive in scenarios such as the following:

- The number of parties is large (so storage and local computation becomes very expensive), *and*
- The network is reasonably fast[14] (so the fact that the communication per multiplication is small does not provide a benefit with respect to the first item), *and*
- The circuit is relatively small (so the benefit of the complexity of the final check being independent of the circuit size matters less).

However, our protocol can potentially become competitive if any of these conditions does not hold. The goal of our experiments is to study precisely this hypothesis, that is, in which settings the exponential nature of our protocol represents an insurmountable overhead, and which cases our protocol can prove beneficial.

We run several experiments to investigate the computation and communication complexity. While the communication complexity has already been analyzed in the previous section, the computation complexity has not. In particular, the use of replicated secret-sharing imposes a non-trivial computationally overhead due to all of the combinatorics involved. Thus the goal is to shed some light on how this complexity grows with the number of inputs and parties, respectively.

For both multiplication and check protocols, we run several experiments where we vary the number of inputs and parties, respectively. We expect to see that *local* computation matters more, the higher the number of parties, and that this complexity outweighs the communication complexity.

Multiplication Our first experiment perform a number of multiplications, where we measure the time to compute the product $\odot$ between the shares; the time it takes to send around shares, and the time it takes to receive and adjust them. Results can be seen in Figure. 1. What is clearly, visible, is that local computation quickly becomes dominant, as the number of multiplication grows and parties grows.

Check Our second experiment, executes the check protocol on a variable number of multiplications. The results can be found in Figure. 2. As expected, we see that the time is essentially constant, regardless of the number of multiplcations. (The outlier for $n = 4, 7$ can be explained by variability in the experimental data, and we believe it should dissapear by increasing the number of trials.)

However, we also see that local computation (represented by the blue bar) increases in significance as both the number of parties and the number of things to check, increases.

[14] This is partly our motivation for choosing a (simulation of a) WAN network for our experiments.

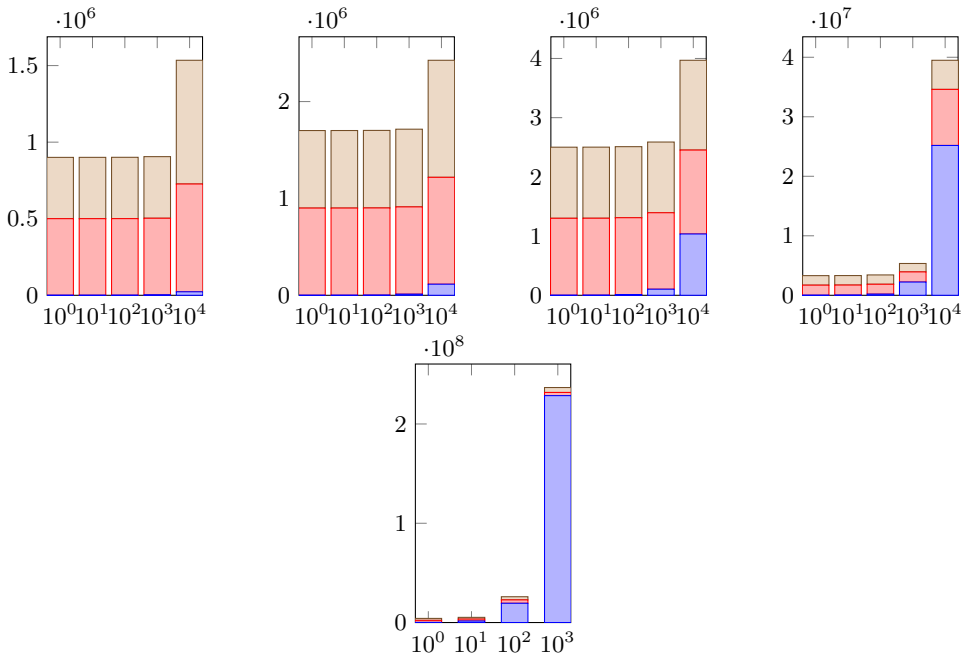

Fig. 1. Multiplication protocol. Each graph represents a different choice for the number of parties $n \in \{4, 7, 10, 13, 16\}$ while each bar in each graph represents the number of multiplications performed (x-axis). The y-axis represents time in microseconds. In each graph the blue section corresponds to the local product each party performs, the red section is the time it takes to send these messages to P_1, and the beige section is the time it takes for P_1 to send the reconstructions back.

n	Input phase (s)	Mult. phase (s)	Check phase (s)
4	0.6	1.5	0.6
7	1.2	2.4	0.6
10	1.8	3.9	0.8
13	2.5	39.5	25.6

Table 2. End-to-end runtimes (in seconds) for a circuit with 100 input gates and 10000 multiplication gates distributed across one layer, for a varying number of parties.

End-to-End Finally, for the sake of completeness we include some end-to-end results. As we have already argued, our main goal is to explore the concrete practicality of replicated secret-sharing-based protocols for increasing number of parties, for which the experiments presented in previous section are more useful since they show the relative performance of the different parts of our protocol, and they allow us to see how the exponential blow-up of our protocol manifests itself in practice. However, we believe it is fruitful to consider end-to-end runtimes as a *rough* estimate of how our protocol would fare in certain tasks. We warn, however, that these numbers are highly volatile as they strongly depend on the experimental setting, quality of implementation, etc., and they should only be used as a coarse guideline.

We consider a circuit with 100 input gates, and 10000 multiplication gates distributed across one layer. Our results are presented in Table. 2.

Discussion The results of the experiments we performed seem to be in accordance with our hypothesis. Specifically, local computation quickly becomes dominant, in particular in the multiplication protocol, and so our protocol would, in those cases, benefit more from a slower network.

This is in particular relevant, taking into consideration low communication cost of the check, as pointed out earlier.

Acknowledgments

A. Nof supported by ERC Project NTSC (742754).

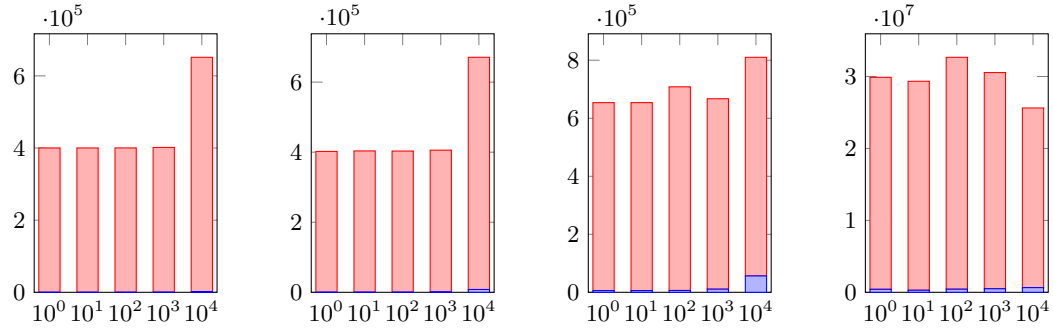

Fig. 2. Check phase. Blue represents preparing $[\![\mathsf{msg}^i]\!]_{2d}$, while the red represents reconstruction. Each graph corresponds to $n \in \{4, 7, 10, 13\}$, the x-axis is number of multiplications that is checked, while the y-axis is time in microseconds.

References

1. E. A. Abbe, A. E. Khandani, and A. W. Lo. Privacy-preserving methods for sharing financial risk exposures. *American Economic Review*, 102(3):65–70, 2012.
2. M. Abspoel, R. Cramer, I. Damgård, D. Escudero, and C. Yuan. Efficient information-theoretic secure multiparty computation over $\mathbb{Z}/p^k\mathbb{Z}$ via galois rings. In *Theory of Cryptography Conference*, pages 471–501. Springer, 2019.
3. M. Abspoel, A. Dalskov, D. Escudero, and A. Nof. An efficient passive-to-active compiler for honest-majority mpc over rings. In *International Conference on Applied Cryptography and Network Security*, pages 122–152. Springer, 2021.
4. A. Baccarini, M. Blanton, and C. Yuan. Multi-party replicated secret sharing over a ring with applications to privacy-preserving machine learning. *Cryptology ePrint Archive*, 2020.
5. A. Barak, M. Hirt, L. Koskas, and Y. Lindell. An end-to-end system for large scale P2P mpc-as-a-service and low-bandwidth MPC for weak participants. In *ACM CCS*, 2018.
6. Z. Beerliová-Trubíniová and M. Hirt. Perfectly-secure MPC with linear communication complexity. In *TCC*, 2008.
7. Z. Beerliová-Trubíniová and M. Hirt. Perfectly-secure MPC with linear communication complexity. In R. Canetti, editor, *Theory of Cryptography, Fifth Theory of Cryptography Conference, TCC 2008, New York, USA, March 19-21, 2008*, volume 4948 of *Lecture Notes in Computer Science*, pages 213–230. Springer, 2008.
8. A. Ben-Efraim, M. Nielsen, and E. Omri. Turbospeedz: Double your online spdz! improving spdz using function dependent preprocessing. In R. H. Deng, V. Gauthier-Umaña, M. Ochoa, and M. Yung, editors, *Applied Cryptography and Network Security*, pages 530–549, Cham, 2019. Springer International Publishing.
9. M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *ACM Symposium on Theory of Computing*, 1988.
10. P. Berman, J. A. Garay, and K. J. Perry. Bit optimal distributed consensus. In *Computer science*, pages 313–321. Springer, 1992.
11. D. Bogdanov, R. Talviste, and J. Willemson. Deploying secure multi-party computation for financial data analysis. In *International Conference on Financial Cryptography and Data Security*, pages 57–64. Springer, 2012.
12. P. Bogetoft, D. L. Christensen, I. Damgård, M. Geisler, T. Jakobsen, M. Krøigaard, J. D. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter, et al. Secure multiparty computation goes live. In *International Conference on Financial Cryptography and Data Security*, pages 325–343. Springer, 2009.
13. D. Boneh, E. Boyle, H. Corrigan-Gibbs, N. Gilboa, and Y. Ishai. Zero-knowledge proofs on secret-shared data via fully linear pcps. In *Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part III*, pages 67–97, 2019.

14. D. Boneh, E. Boyle, H. Corrigan-Gibbs, N. Gilboa, and Y. Ishai. Zero-knowledge proofs on secret-shared data via fully linear pcps. In *CRYPTO*, 2019.
15. E. Boyle, N. Gilboa, Y. Ishai, and A. Nof. Practical fully secure three-party computation via sublinear distributed zero-knowledge proofs. In *ACM CCS*, 2019.
16. E. Boyle, N. Gilboa, Y. Ishai, and A. Nof. Efficient fully secure computation via distributed zero-knowledge proofs. In *Advances in Cryptology - ASIACRYPT*, pages 244–276, 2020.
17. R. Canetti. Security and composition of multiparty cryptographic protocols. *J. Cryptology*, 13(1):143–202, 2000.
18. D. Chaum, C. Crépeau, and I. Damgard. Multiparty unconditionally secure protocols. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 11–19, 1988.
19. E. Check Hayden. Extreme cryptography paves way to personalized medicine. *Nature News*, 519(7544):400, 2015.
20. J. H. Cheon, D. Kim, and K. Lee. Mhz2k: Mpc from he over $z/2^k z$ with new packing, simpler reshare, and better zkp. In *Annual International Cryptology Conference*, pages 426–456. Springer, 2021.
21. K. Chida, D. Genkin, K. Hamada, D. Ikarashi, R. Kikuchi, Y. Lindell, and A. Nof. Fast large-scale honest-majority MPC for malicious adversaries. In *CRYPTO*, 2018.
22. H. Cho, D. J. Wu, and B. Berger. Secure genome-wide association analysis using multiparty computation. *Nature biotechnology*, 36(6):547, 2018.
23. A. R. Choudhuri, A. Goel, M. Green, A. Jain, and G. Kaptchuk. Fluid mpc: Secure multiparty computation with dynamic participants. *IACR Cryptol. ePrint Arch*, 754:2020, 2020.
24. B. A. Coan and J. L. Welch. Modular construction of nearly optimal byzantine agreement protocols. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing, Edmonton, Alberta, Canada, August 14-16, 1989*, pages 295–305, 1989.
25. R. Cramer, I. Damgård, D. Escudero, P. Scholl, and C. Xing. Spdz2k: Efficient MPC mod 2k for dishonest majority. In *CRYPTO*, 2018.
26. R. Cramer, I. Damgård, and Y. Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In *TCC*, 2005.
27. R. Cramer, S. Fehr, Y. Ishai, and E. Kushilevitz. Efficient multi-party computation over rings. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 596–613. Springer, 2003.
28. A. Dalskov, D. Escudero, and M. Keller. Secure evaluation of quantized neural networks. *Proceedings on Privacy Enhancing Technologies*, 4:355–375, 2020.
29. A. Dalskov, D. Escudero, and M. Keller. Fantastic four: Honest-majority four-party secure computation with malicious security. In *USENIX 2021*, 2021.
30. I. Damgård, D. Escudero, T. Frederiksen, M. Keller, P. Scholl, and N. Volgushev. New primitives for actively-secure mpc over rings with applications to private machine learning. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1102–1120. IEEE, 2019.
31. I. Damgård and J. B. Nielsen. Scalable and unconditionally secure multiparty computation. In *CRYPTO*, 2007.
32. D. Dolev and H. R. Strong. Authenticated algorithms for byzantine agreement. *SIAM J. Comput.*, 12(4):656–666, 1983.
33. D. Escudero and E. Soria-Vazquez. Efficient information-theoretic multi-party computation over non-commutative rings. In *Annual International Cryptology Conference*, pages 335–364. Springer, 2021.
34. J. Furukawa and Y. Lindell. Two-thirds honest-majority MPC for malicious adversaries at almost the cost of semi-honest. In *ACM CCS*, 2019.
35. O. Goldreich. *The Foundations of Cryptography - Volume 2, Basic Applications*. Cambridge University Press, 2004.
36. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *ACM Symposium on Theory of Computing*, 1987.
37. S. D. Gordon, S. Ranellucci, and X. Wang. Secure computation with low communication from cross-checking. In *ASIACRYPT*, 2018.
38. V. Goyal, H. Li, R. Ostrovsky, A. Polychroniadou, and Y. Song. Atlas: efficient and scalable mpc in the honest majority setting. In *Annual International Cryptology Conference*, pages 244–274. Springer, 2021.
39. V. Goyal, Y. Liu, and Y. Song. Communication-efficient unconditional MPC with guaranteed output delivery. In *CRYPTO*, 2019.
40. V. Goyal, Y. Song, and C. Zhu. Guaranteed output delivery comes free in honest majority MPC. In *CRYPTO*, 2020.
41. B. Hemenway, S. Lu, R. Ostrovsky, and W. Welser Iv. High-precision secure computation of satellite collision probabilities. In *International Conference on Security and Cryptography for Networks*, pages 169–187. Springer, 2016.
42. M. Hirt, U. M. Maurer, and B. Przydatek. Efficient secure multi-party computation. In *ASIACRYPT*, 2000.
43. M. Ito, A. Saito, and T. Nishizeki. Secret sharing scheme realizing general access structure. *Electronics and Communications in Japan*, 1989.
44. N. Koti, M. Pancholi, A. Patra, and A. Suresh. Swift: super-fast and robust privacy-preserving machine learning. In *USENIX 2021*, 2021.

45. N. Koti, A. Patra, R. Rachuri, and A. Suresh. Tetrad: Actively secure 4pc for secure training and inference. Cryptology ePrint Archive, Report 2021/755, 2021. https://ia.cr/2021/755. To appear at NDSS 2022.
46. E. Kushilevitz, Y. Lindell, and T. Rabin. Information-theoretically secure protocols and security under composition. *SIAM J. Comput.*, 39(5):2090–2112, 2010.
47. E. Orsini, N. P. Smart, and F. Vercauteren. Overdrive2k: Efficient secure mpc over $z/2^k z$ from somewhat homomorphic encryption. In *Cryptographers' Track at the RSA Conference*, pages 254–283. Springer, 2020.
48. T. Rabin and M. Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority. In *ACM Symposium on Theory of Computing*, 1989.
49. A. C. Yao. How to generate and exchange secrets (extended abstract). In *Symposium on Foundations of Computer Science*, 1986.

A Example: Four Parties and One Corruption

Our protocol works for any number of parties n and adversarial threshold $t < n/3$. However, as mentioned in Section 1.2, a big body of research has been devoted to the case of $t = 1$ and $n = 4$. In this section we give an alternative presentation of our protocol for this concrete case.

A.1 Replicated Secret-Sharing

Let P_1, P_2, P_3, P_4 be the set of parties. First we describe the main secret-sharing scheme used, which is replicated secret-sharing with threshold 1.

- $\mathsf{share}(x)$: To share a secret x with threshold 1, the dealer samples four random values $x_1, \ldots, x_4 \in R$ under the constraint that $x = x_1 + x_2 + x_3 + x_4$. P_1 gets (x_2, x_3, x_4), P_2 gets (x_1, x_3, x_4), P_3 gets (x_1, x_2, x_4) and P_4 gets (x_1, x_2, x_3). $[\![x]\!]$ denotes the collection of these shares.
- $\mathsf{reconstruct}([\![x]\!], i)$: In this interactive procedure, each party P_j for $j \neq i$ sends x_j to P_i, who takes the majority of the values received and reconstructs $x = x_1 + x_2 + x_3 + x_4$.

Observe that each share x_i is held by the parties $\{P_k\}_{k \neq i}$. However, a dishonest dealer can distribute shares in such a way that these parties do not get the same x_i. We say that the shares $[\![x]\!]$ are *consistent* if, for every $i \in [4]$, the honest parties among $\{P_k\}_{k \neq i}$ receive the same value x_i.

Checking that a dealer distributed shares $[\![x]\!]$ consistenly can be done by asking each pair of parties P_i and P_j to exchange their common shares x_k for $k \notin \{i, j\}$, verifying that they coincide. For verifying the consistency of multiple shared values, the parties can exchange hashes instead of the actual shares.

It is easy to see that if a sharing $[\![x]\!]$ is consistent, then P_i is able to learn the correct secret x after the execution of $\mathsf{reconstruct}([\![x]\!]_1, i)$. This is because P_i receives the missing share x_i from at least two honest parties, which are guaranteed to hold the same value due to the consistency of the sharing.

Shares of Shares Finally, we notice that if the parties have shares $[\![x]\!]$, with $x = x_1 + x_2 + x_3 + x_4$, then the parties also have $[\![x_i]\!]$ for $i \in [4]$. This is because we can define $x_{ij} = x_i$ for $j = i$, and $x_{ij} = 0$ otherwise, so $x_i = x_{i1} + x_{i2} + x_{i3} + x_{i4}$, and it can be easily checked that each party P_k can compute x_{ij} for $j \neq k$.

Threshold 2 We will also need replicated secret sharing with threshold 2, which is denoted by $(\!|x|\!)$. In such sharing, each party P_i holds x_{jk} with $i \neq j$ and $i \neq k$, where $x = x_{\{1,2\}} + x_{\{1,3\}} + x_{\{1,4\}} + x_{\{2,3\}} + x_{\{2,4\}} + x_{\{3,4\}}$. For simplicity we write $x_{\{i,j\}} = x_{ij}$, implicitly understanding that $x_{ij} = x_{ji}$. For example, P_1 has (x_{23}, x_{24}, x_{34}) as his share.

Observe that in the above sharing P_i is missing the shares x_{ij} for $j \neq i$. For reconstruction, P_i can receive this value from P_ℓ, where $\ell \notin \{i, j\}$. Since there are two such possible indexes ℓ_1, ℓ_2, we ask P_{ℓ_1} to send x_{ij} to P_i, while P_{ℓ_2} sends a hash of x_{ij}, where $\ell_1 < \ell_2$.[15] This procedure is denoted by $\mathsf{reconstruct}((\!|x|\!), i)$.

If P_i receives inconsistent values from P_{ℓ_1} and P_{ℓ_2}, then P_i broadcasts a complaint to the other parties, together with the two inconsistent values. Then P_{ℓ_1} and P_{ℓ_2} respond by broadcasting the actual values that they sent P_i. Based on this the parties can identify a semi-corrupt pair: if the announced messages by P_{ℓ_1} and P_{ℓ_2} do not coincide, then (P_{ℓ_1}, P_{ℓ_2}) is a semi-corrupt pair. Else, if P_i's message disagrees with the one from P_{ℓ_1}, (P_i, P_{ℓ_1}) is a semi-corrupt pair, and finally, if P_i's message disagrees with the one from P_{ℓ_2}, then (P_i, P_{ℓ_2}) is a semi-corrupt pair.

[15] This is mostly useful when reconstructing multiple values at once, as must be done in our protocol. In this case P_{ℓ_2} only sends a hash of the concatenation of all the shares x_{ij}.

Local Operations It is easy to see that the parties can locally add their shares of $[\![x]\!]$ and $[\![y]\!]$ to obtain shares $[\![x \cdot y]\!]$. Furthermore, the parties can obtain shares $[\![x + e]\!]$ from $[\![x]\!]$ if the value e is known by three parties P_i, P_j, P_k, by simply letting these parties add e to their share indexed by $\ell \in [4] \setminus \{i, j, k\}$.

Finally, the parties can locally obtain shares $(\!|x \cdot y|\!)$ from $[\![x]\!]$ and $[\![y]\!]$. This is denoted by $(\!|x \cdot y|\!) = [\![x]\!] \cdot [\![y]\!]$, and this proceeds as follows: P_i initially defines $z_{jk} = x_j \cdot y_k + x_k \cdot y_j$, for $j, k \notin \{i\}$ with $j \neq k$. Then, P_i updates $z_{i+1,i+2} \leftarrow z_{i+1,i+2} + x_{i+1} \cdot y_{i+1} + x_{i+2} \cdot y_{i+2}$.[16] It can be checked that $x \cdot y = z_{12} + z_{13} + z_{14} + z_{23} + z_{24} + z_{34}$, so the parties have shares $(\!|x \cdot y|\!)$.

A.2 Non-Interactive Correlation Generation

In our protocol the parties need to make use of certain preprocessing material that can be generated from a simple setup. We discuss this below.

Setup We assume that the parties hold $[\![k^{(i)}]\!]$ for $i \in [4]$, where P_i knows the random key $k^{(i)} \in R$. This can be done by asking P_i to sample $k^{(i)} \in R$ uniformly at random, and then followed by P_i acting as a dealer to distribute shares, together with a consistency check.

Shared random values Given $k \in R$ such that the parties hold shares $[\![k]\!]$, the parties can locally obtain $[\![F_k(\mathtt{x})]\!]$ for any input $\mathtt{x}$ by setting P_i's share to be $\{F_{k_j}(\mathtt{x})\}_{j \neq i}$. Here $k = k_1 + k_2 + k_3 + k_4$. We denote this by $[\![F_k(\mathtt{x})]\!] = F_{[\![k]\!]}(\mathtt{x})$. If some party P_i knows k, then this party knows $F_k(\mathtt{x})$.

Double sharings We use $\langle x \rangle$ to denote additive secret sharing among P_1, P_2, P_3 of x. Let $[\![k]\!] = [\![k^{(1)}]\!] + [\![k^{(2)}]\!] + [\![k^{(3)}]\!]$. The parties can locally obtain $[\![r]\!]$ where $r = F_k(\mathtt{x})$ as indicated above. Since P_i knows $k^{(i)}$, this means that (r_1, r_2, r_3), with $r_i = F_k(\mathtt{x})$, constitute sharings $\langle r \rangle$. Furthermore, the parties have $[\![k^{(i)}]\!]$, they can obtain $[\![r_i]\!]$, which means that the additive shares of r are $[\![\cdot]\!]$-shared among the parties. The pair $([\![r]\!], \langle r \rangle)$ is called a *double-sharing*, and we assume that the parties use a pre-agreed different input $\mathtt{x}$ for double-sharing generation (e.g. a counter).

Shares of zero Given $k \in R$ in secret-shared form $[\![k]\!]$, the parties can obtain randomly-looking shares of zero $(\!|0|\!)$ as follows. Let $r_i = F_{k_i}(\mathtt{x})$. Define the share $z_{i+1,i+2} = r_{i+1} - r_{i+2}$ for $i \in [4]$, and $z_{jk} = 0$ otherwise. Observe that each party P_i can locally compute z_{jk} for $j, k \notin \{i\}$, and $0 = \sum_{i \leq j} z_{ij}$.

A.3 Input Phase

Let x_i be the input of P_i. To distribute shares of this input, let $[\![k^{(i)}]\!]$ be the shares of the key $k^{(i)}$ distributed by P_i as part of the setup phase. Then P_i broadcasts $x_i - r_i$ to the parties, where $r_i = F_{k_i}(\mathtt{x})$, who locally compute $[\![x_i]\!] = [\![r_i]\!] + (x_i - r_i)$. Notice that the parties have shares $[\![r_i]\!]$, as described in the previous section.

A.4 Secure Multiplication

The protocol is presented as Protocol 6. It is a direct instantiation of Protocol 2 to the case of $n = 4$. The messages sent by the parties are made more explicit in order to showcase the communication pattern.

Communication Complexity. We see that P_2 and P_3 send one element each to P_1, who sends one element back to P_2 and P_3, resulting in a total of 4 ring elements being communicated. Observe also that P_4 is not involved in any of these interactions. As observed in Section 7, the communication pattern of our protocol enables running the computation among $2t + 1 = 3$ parties only, requiring only the involvement of the remaining party for the final check, discussed in the next section.

Observe that, if P_4 does not need to provide input, it does not even need to participate of the input protocol. Hence, we can think of our protocol as a three-party protocol that requires the aid of a fourth party that joins only during the final check phase. This fourth party only needs to share the same set-up as the other parties, which in fact can be distributed to this extra party during the final check phase (so it does not even need to share some state with the other parties prior to joining the verification step).

The above is in contrast to other existing four-party protocols such as [37,44], which require permanent participation from all the four participants (on top of requiring more communication per party). However, other four-party works like [29,45] also allow for active participation from two parties only.

[16] For the rest of this section, the subindices wrap around modulo 4 in the set [4].

PROTOCOL 6 (The Optimized DN Multiplication Protocol)

- **Inputs:** The parties hold $[\![x]\!]$, $[\![y]\!]$.
- **Setup:** The parties obtain a double-sharing $([\![r]\!], \langle r \rangle)$ non-interactively. The additive shares are denoted by (r_1, r_2, r_3).
- **The protocol:**
  1. P_2 sends $e_2 = x_1y_3 + x_1y_4 + x_2y_1 + x_3y_1 + x_4y_1 - r_2$ to P_1, and P_3 sends $e_3 = x_1y_2 + x_2y_1 - r_3$ to P_1.
  2. P_1 reconstructs $xy - r = e_1 + e_2 + e_3$, where $e_1 = -r_1 + \sum_{i,j=2}^{4} x_i y_j$.
  3. P_1 sends $e = xy - r$ to P_2 and P_3.
  4. The parties output the shares $[\![z]\!] = [\![r]\!] + e$.

A.5 Identifying a Semi-Corrupt Pair

Now we discuss how the parties can detect that some party cheated in the multiplication protocol. For simplicity we discuss how this is done for the case in which there is only one multiplication. If there are many, the transcript can be compressed into one single check, as discussed in Section 5.2.

We make use of the notation of the protocols above. Let $[\![x]\!]$ and $[\![y]\!]$ be the inputs to the multiplication protocol. As discussed in Section A.1, the parties automatically hold shares $[\![x_i]\!]$ and $[\![y_i]\!]$ for $i \in [4]$. Also, by design of the non-interactive procedure to generate the double-sharings from Section A.2, this also holds for the additive shares $[\![r_i]\!]$, $i \in [4]$.

The verification protocol is presented as Protocol 7.

PROTOCOL 7 (Semi-Corrupt Pair Detection)

- **Inputs:** $[\![x]\!], [\![y]\!]$, and the double-sharing $([\![r]\!], \langle r \rangle)$.
- **The protocol:**
  1. P_1 broadcasts e_2 and e_3, P_2 broadcasts e_2 and P_3 broadcasts e_3. The parties output a semi-corrupt pair (P_1, P_j) if the message e_j announced by P_1 and P_j are not consistent.
  2. The parties locally compute

$$(\!|e_2|\!) = [\![x_1]\!] \cdot [\![y_3]\!] + [\![x_1]\!] \cdot [\![y_4]\!] + [\![x_2]\!] \cdot [\![y_1]\!] + [\![x_3]\!] \cdot [\![y_1]\!] + [\![x_4]\!] \cdot [\![y_1]\!] - [\![r_2]\!] + (\!|0|\!)$$

     and

$$(\!|e_3|\!) = [\![x_1]\!] \cdot [\![y_2]\!] + [\![x_2]\!] \cdot [\![y_1]\!] - [\![r_3]\!] + (\!|0|\!).$$

  3. The parties call $\mathsf{reconstruct}((\!|e_2|\!), i)$ and $\mathsf{reconstruct}((\!|e_3|\!), i)$ for $i \in [4]$ to reconstruct these values towards the parties.
  4. If no semi-corrupt pair was identified in the previous step, then the parties proceed as follows
     - If the e_2 reconstructed from the previous step is different from the one announced by P_1 (or P_2, which have been checked to be the same) in the first step of the protocol, then the parties output the corrupt party P_1.
     - If the e_3 reconstructed from the previous step is different from the one announced by P_1 (or P_3, which have been checked to be the same) in the first step of the protocol, then the parties output the corrupt party P_3.

B Reducing the Number of Rounds

Our protocol, which is based on the DN protocol, requires two rounds to evaluate a single multiplication gate: one round for the parties to send shares to the "king" P_1, and another round for this party to send the reconstructed masked value to (a subset of) the other parties. In [38], a method to securely evaluate two multiplication layers using two rounds, which amounts to one round per layer overall, is presented. However, this is introduced in the context of the original DN protocol, which makes use of

Shamir secret-sharing. In this section we show that these techniques can also be used in our current setting with replicated secret-sharing to achieve one-round multiplication without substantially hurting communication complexity. Furthermore, we combine the techniques from [38] with these from [8] in order to futher optimize our protocol. We remark that such optimizations, with some tweaks, can also be applied to the original work of [38] with Shamir secret-sharing, improving over that protocol.

B.1 Applying ATLAS to our Setting

The intuition of the round compression techniques in [38] are as follows. For illustrative purposes suppose we have a two-layer circuit like this: there are four inputs x_1, x_2, y_1, y_2, the first layer multiplies $z_1 = x_1 \cdot y_1$ and $z_2 = x_2 \cdot y_2$, and the second layer multiplies $w = z_1 \cdot z_2$. The goal is to evaluate these two layers in two rounds. To do this, the parties begin as in our current protocol: to multiply $x_i \cdot y_i$, $2t$ parties send in the first round their shares of $e_i = x_i \cdot y_i - r_i$ to P_1, and in the second round, P_1 will send e_1 and e_2 to all the parties (in our original protocol from Section 5.1 it suffices for P_1 to send these values to a subset of the parties, but here P_1 needs to send these to all the parties).

At this point the parties can define $[\![z_i]\!]_d = [\![r_i]\!]_d + e_i$ for $i = 1, 2$ as in our original protocol, but the ultimate goal is to obtain $[\![w = z_1 \cdot z_2]\!]_d$, and no more communication is allowed given that two rounds have been already spent. However, this can be done by noticing that, if the parties had preprocessed $[\![r_1 \cdot r_2]\!]_d$, they could compute $[\![w]\!]_d = e_1 \cdot [\![r_2]\!]_d + e_2 \cdot [\![r_1]\!]_d + [\![r_1 \cdot r_2]\!]_d + e_1 \cdot e_2$. Intuitively, one could say this techniques uses the masked values e_1 and e_2 as the openings required for Beaver-based multiplication, where the corresponding multiplication triple is $([\![r_1]\!]_d, [\![r_2]\!]_d, [\![r_1 \cdot r_2]\!]_d)$.

B.2 Using the Turbospeedz Invariant

Coupled with the above, we can also make use of the invariant for secure computation introduced in [8]. Currently, in order to securely evaluate a given circuit, the parties hold shares $[\![x]\!]_d$ of each intermediate value x. Progress through the computation is guaranteed via the linearity of $[\![\cdot]\!]_d$, together with our multiplication protocol. Furthermore, privacy is achieved since an adversary controlling at most d parties cannot infer anything about x from $[\![x]\!]_d$.

An alternative invariant is proposed in [8] where, instead of the parties holding shares $[\![x]\!]_d$ of each intermediate value, the parties hold shares of a completely random value $[\![\lambda_x]\!]_d$, together with a *public* value μ_x known to all parties that perfectly hides the actual wire value x as $\mu_x = x - \lambda_x$. If x is the output of an addition gate $x = u + v$, then we assume that $\lambda_x = \lambda_u + \lambda_v$. This ensures that the invariant can be preserved through addition gates by also defining $\mu_x = \mu_u + \mu_v$. In what follows we discuss how this new invariant can be combined with the techniques sketched in the previous section to further improve our protocol.

B.3 Final Protocol with Passive Security

We begin by describing, for the sake of simplicity, the passively secure version of our optimized protocol. The fully-secure version is described below in the next section. Here, because of the adversary being passive, we can further modify the required invariant by allowing μ_x to be only known by the parties in the fixed set U.

Our goal is to maintain the invariant previously described, namely, for every intermediate wire value x, the parties hold shares of a uniformly random value $[\![\lambda_x]\!]_d$, together with a publicly known value μ_x satisfying $\mu_x = x - \lambda_x$ (with the additional restriction for output wires of multiplication gates). These sharings can be processed non-interactively as described in Section 4.2. This invariant can be achieved for input gates by simply enabling each party P_i to learn λ_x, with x being an input wire associated to this party, and asking this party to broadcast $\mu_x = x - \lambda_x$. For the other wires in the circuit, the parties proceed as we describe below.

We begin by splitting the layers (after the input layer) into groups of two layers each, and assume for the sake of recursion that the invariant already holds for all the wires originating from previous layers. First, the parties execute the following local operations:

1. For every multiplication gate in the *first layer* taking inputs x and y, the parties preprocess $[\![\lambda_x \lambda_y]\!]_d$. Let $[\![\lambda_z]\!]_d$ be the random value associated with the output $z = x \cdot y$. The parties in U, knowing $(\mu_x, [\![\lambda_x]\!]_d)$ and $(\mu_y, [\![\lambda_y]\!]_d)$, compute locally $[\![z]\!]_d = \mu_y [\![\lambda_x]\!]_d + \mu_x [\![\lambda_y]\!]_d + \mu_x \mu_y + [\![\lambda_x \lambda_y]\!]_d$ and $[\![\mu_z]\!]_d = [\![xy]\!]_d - [\![\lambda_z]\!]_d$.

2. Consider a multiplication gate in the *second layer* taking inputs u and v, and producing output $w = uv$. Observe that the parties in U have $[\![u]\!]_d$ and $[\![v]\!]_d$.[17] Let $[\![\lambda_w]\!]_d$ be the random value associated with the output $z = x \cdot y$, and assumed it is preprocessed together with $\langle \lambda_w \rangle^U$, as in Section 4.2. At this point the parties in U can locally compute $\langle \mu_w \rangle^U = [\![u]\!]_d \odot_U [\![v]\!]_d - \langle \lambda_w \rangle^U$.

Once this is done, the parties engage in the following two-round interaction:

1. For every multiplication gate in the first layer with output z, the parties in U have $[\![\mu_z]\!]_d$; In case the output wire of this gate is also used as an input to a gate of the first layer, then $d+1$ of these parties in U send their share to P_1.[18] For every multiplication gate in the second layer with output w, the parties in U have $\langle \mu_w \rangle_d$; the parties in U send their shares to P_1.
2. P_1 reconstructs μ_z and μ_w for every mutiplication gate as above, and send these values to the parties in U.

After this interaction, the invariant is now preserved for the outputs of the current pair of layers, which concludes the presentation of the protocol

Communication cost. Let us consider a given pair of subsequent layers being evaluated with the technique above, and let N_1 and N_2 be the number of multiplication gates in the first and second layers respectively.

In the optimistic scenario, which applies to many natural circuits, the cost for the first type of gates is $n + d - 1$ sent elements in the offline phase and the online phase requires no interaction. For the second type, there are $2d$ messages to P_1, and again $n - 1$ messages back. Hence, the total cost is $N_1(n + d - 1) + N_2(2d + n - 1)$ elements being communicated.

Assuming the circuit is balanced, i.e., $N_1 = N_2 =: N/2$, we have that the cost is $\frac{N}{2} \cdot (2(n-1) + 3d)$, which since $n - 1 = 3d$, is equal to $N \cdot \frac{3}{2}(n-1)$ elements. Thus, the communication cost per gate is $\leq \frac{3}{2}$ elements per party. In the 4-party setting, the cost is $\frac{9}{8}$ elements per gate sent per party.

However, in case the circuit has a "bad" structure, i.e., each gate from the first layer feeds gates from both layers, then we need to add communication cost for the first layer of d messages towards P_1 and $n-1$ messages from P_1 to the parties. In this case, the total cost is $N_1(2(n + d - 1)) + N_2(2d + n - 1)$, and so assuming $N_1 = N_2 =: N/2$ as before, this can be approximated to $\frac{N}{2} \cdot (4d + 3(n-1))$. This corresponds to an amortized communication complexity of $2d + \frac{3}{2}(n-1)$ elements per gate. Given that $n - 1 = 3d$, this is equal to $\frac{13}{6}(n-1)$ elements and so $\leq 2\frac{1}{6}$ elements per gate for each party. In the 4-party case, this amounts to $\frac{13}{8}$ elements per gate per party.

B.4 Achieving Full Security

To achieve full security, the parties need to verify that the correct messages were sent in the protocol above. Note that there are three types of interactions to check: (i) interaction to compute the second layer of gates in the above description; (ii) interaction to convert additive shares on the output wires of the first layer of gates in the above description, into public masked values (to maintain the invariant on the wires); and (iii) interaction to compute the preprocessed random triples $([\![\lambda_x]\!], [\![\lambda_y]\!], [\![\lambda_x \lambda_y]\!])$ for the gates in the first layer.

Note that in (i) and (iii) the parties simply run the DN protocol as in Section 5.1. Thus, verification can work exactly as before, namely, as we described in Section 5.2. It remains to show how verifying correctness for (ii) takes place. Recall that here the parties simply reveal μ_z by having $d+1$ parties send shares to P_1 who reconstructs μ_z and sends it back to the parties. Observe that at the begining of this interaction, the parties hold $[\![\mu_z]\!]$, which means that μ_z is shared in a robust way across the parties. This implies that in particular each *share* of μ_z is known by a majority of honest parties. Thus, we can apply the same verification mechanism as before. Specifically, the parties first agree on the compressed transcript by having each party who participated in the interaction publishing a random linear combination of the its sent and received messages. Then, the parties can verify each party's compressed message by reconstructing it. Namely, each party who holds the shares that were sent by some party P_i to P_1 compute

[17] Indeed, if u (or v) is an output from a multiplication in the previous layer, then this follows from the first item. If u (or v) is an output from a previous layer, then the invariant implies that the parties in U know μ_u (or μ_v), so in particular they can define $[\![u]\!]_d = \mu_u + [\![\lambda_u]\!]_d$ (and similarly $[\![v]\!]_d$).

[18] An important optimization is that, if the value z is only intended to be used as an input to multiplication gates in the second layer, then the parties do not need to send this to P_1. This is because the parties already used $[\![\mu_z]\!]_d$ to compute the gate in the second layer where z appears as input, and they do not need to learn μ_z anymore.

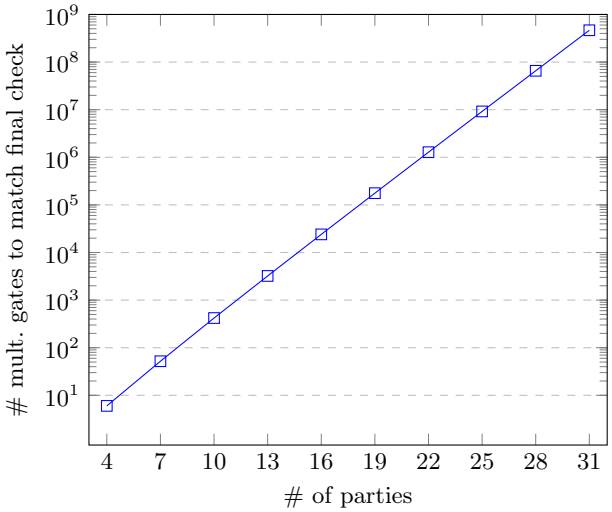

Fig. 3. Number of multiplication gates required (in terms of the number of parties) in order to match the communication cost of evaluating these gates with the cost of the final check. This is computed by dividing the communication cost of the final check by the communication cost of a single multiplication gate. When this number of gates is reached then the overhead of the final check is only 2x.

the random linear combination of these shares and publish it. Since each share is known by a majority of honest parties, the parties will obtain the correct value and identify any cheating by P_i. Finally, if all messages sent to P_1 are correct, the parties can emulate P_1's role and verify the correctness of his compressed message.

Note that the cost of the verification protocol remains constant, as in our main construction, and thus is amortized away.

C More Comments on the Communication Costs

C.1 Overhead of the Final Check

The communication complexity of the final check $\mathcal{F}_{\mathsf{checkTrans}}$, although independent of the number of multiplications being checked, grows exponentially with the number of parties. Hence, a natural question is: at what point is the number of multiplication gates large enough so that the overhead of this final check is less noticeable? To this end, in Fig. 3 we plot, in terms of the number of parties, the necessary number of multiplication gates so that the communication complexity of securely computing them equals that of the final check. In other words, when this number of gates is reached the overhead of the final check is exactly 2x, and with more gates, the overhead shrinks accordingly (e.g. with k times the number of gates the overhead becomes $1\frac{1}{k}$).

We observe that, for circuits having one million (10^6) gates, we can reach up to 22 parties with a final check that only doubles communication complexity, and if the circuit has one billion (10^9) gates then the number of parties can be increased up to 31 while keeping the overhead of the final check below 2x. This number of gates can be easily encountered, for example, in securely training/evaluating certain neural network architectures (e.g. GPT3[19] itself has 175 billion parameters, which translates to roughly the same number of multiplications).

C.2 On the Amount of Segments

Recall that the communication complexity of computing a circuit C depends on the amount of segments m, and it also differs in the optimistic case with respect to the worst case. In the optimistic scenario where there is no cheating, the communication complexity to evaluate a circuit C can be written as $|C| \cdot (\texttt{opt\_mult}) + (\texttt{opt\_check})$, while in the worst case where the last segment is executed t more times,

[19] https://arxiv.org/abs/2005.14165

		Circuit size			
	m	10^3	10^4	10^5	10^6
Opt.	1	1.71 MiB	1.8 MiB	2.66 MiB	11.34 MiB
	10	17.01 MiB	17.09 MiB	17.96 MiB	26.63 MiB
	25	42.5 MiB	42.59 MiB	43.45 MiB	52.13 MiB
	50	84.99 MiB	85.08 MiB	85.94 MiB	94.62 MiB
Worst	1	2.03 MiB	2.64 MiB	8.71 MiB	69.42 MiB
	10	17.27 MiB	17.41 MiB	18.8 MiB	32.68 MiB
	25	42.77 MiB	42.87 MiB	43.95 MiB	54.7 MiB
	50	85.25 MiB	85.35 MiB	86.32 MiB	96.04 MiB
Ratio	1	1.19	1.47	3.27	6.12
	10	1.02	1.02	1.05	1.23
	25	1.01	1.01	1.01	1.05
	50	1	1	1	1.02

Table 3. Total communication complexity in the optimistic and worst-case scenarios for different circuit sizes and different values of m, the number of segments. Here n, the number of parties, equals 19. We note that the gap between the optimistic and worst-case scenarios is larger for small m, and this is more notorious as the circuit size grows larger. This gap shrinks if m increases, but at the expense of making the complexities concretely much more expensive.

each with three less parties, we can write this as $|C| \cdot (\texttt{worst\_mult}) + (\texttt{worst\_check})$. We can find close formulas for these quantities:

$$\begin{aligned} \texttt{opt\_mult} &= \frac{4}{3} \\ \texttt{worst\_mult} &= \texttt{opt\_mult} \cdot (1 + \frac{t}{m}) \\ \texttt{opt\_check} &= m \cdot \mathsf{Ch}_n \\ \texttt{worst\_check} &= \texttt{opt\_check} + \sum_{\ell=1}^{t} \mathsf{Ch}_{n-3\ell}, \end{aligned}$$

where $\mathsf{Ch}_n = \binom{n-1}{2t} \cdot 2t$. In Section 7.1 we reported concrete costs for the case in which $m = 1$, which minimizes `opt_check`, and hence minimizes the total communication complexity of the optimistic case.

In the event of a real-world deployment the choice $m = 1$ is arguably the most sensible one in practice. Nevertheless, it is still relevant to explore the trade-offs of considering other values of m. As m increases, the *ratio* between the total communication complexity in the worst case w.r.t. the optimistic case gets smaller, which can be interpreted as reducing how much adversarial behavior can affect the communication complexity. However, this happens at the expense of increasing the concrete complexity of the optimistic case, which may not be ideal.

We explore in Table 3, for different choices of m, the total communication complexity in both the optimistic and worst-case settings, for a fixed number of parties and for a varying range of circuit sizes. The table confirms our observations that the gap between the optimistic and the worst-case settings shrinks if we choose a larger m, but this increases the *concrete* efficiency of the optimistic (and hence the worst-case) scenario.

C.3 Comparison with [34]

Let us assume that $\omega_R \approx 2^s$, so the number of repetitions needed for the final check is only 1. Furthermore, let us assume that $|S| = |C|$, that is, the whole circuit is the only segment. In this setting, the total communication of our protocol is $\mathcal{C}_1 = (1 + \frac{t-1}{n}) \cdot |C| + |\mathcal{F}_{\mathsf{checkTrans}}|$, where $|\mathcal{F}_{\mathsf{checkTrans}}| = \binom{n-1}{2t} \cdot 2t$ (ignoring the cost of broadcast). On the other hand, under these circumstances the communication complexity of [34] is $\mathcal{C}_2 = 2\frac{2}{3} \cdot |C| + 6\frac{2}{3}n$. Their communication per multiplication is higher than ours, but the additive term independent of the circuit size is smaller. It is important to take into account at this point that *our protocol achieves G.O.D. , while the protocol from [34] only ensures fairness.*

We are interested in the behavior of the quotient $\mathcal{C}_2/\mathcal{C}_1$, which represents the improvement factor of our work with respect to [34]. For fixed n, this quotient approaches 2 as $|C| \to \infty$, but our goal is to

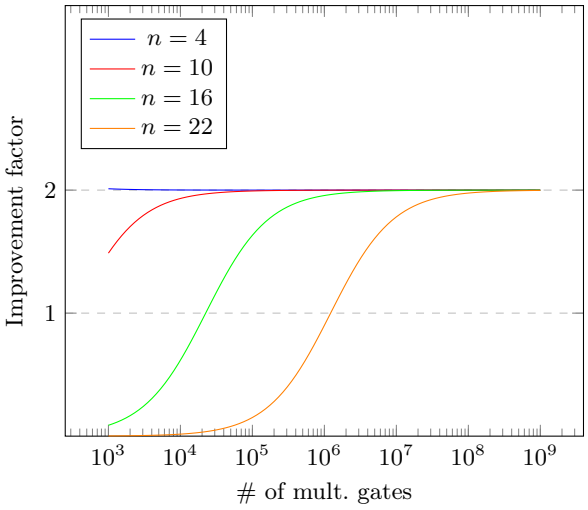

Fig. 4. Improvement factor of our protocol with respect to the one from [34] as the number of multiplication gates grow, for some values of n. Asymptotically, our protocol performs two times better than the one from [34], but as n grows the number of multiplication gates required to reach that factor increases (exponentially).

determine more concretely at which point this happens. To this end, in Figure 4 we plot this quotient for $n \in \{4, 10, 16, 22\}$. We observe that for a small number of parties, like $n = 4$ or $n = 10$, even relatively small circuits with a few thousand multiplication gates already benefit from our protocol, which has a communication complexity of roughly half with respect to [34]. For larger values of n, like $n = 16$ and $n = 22$, the threshold in which the improvement factor starts approaching 2 is above $\approx 10^5$ and $\approx 10^7$ multiplication gates, respectively.

We remark that our protocol, in contrast to the one from [34], does not make use of polynomial-based arithmetic, and instead, only uses simple integer arithmetic. In addition, the communication of our protocol is less during *most* of the protocol execution, with the heaviest messages appearing at the end at the verification stage.

C.4 Communication Pattern

Another feature of our protocol that has practical impact is that its communication pattern is such that not all parties need to interact in the execution of the protocol, except for the final verification step, where all the parties are involved. Indeed, we see that, for each multiplication gate, the communication pattern follows a star topology in which the $2t$ parties in the set $U \setminus \{P_1\}$ send messages to P_1, who then replies back to these parties. The remaining $\geq t$ parties are inactive during these executions, and only participate in the final verification phase. This is important since it reduces communication channels, and allows us to save in other resources like monetary costs of keeping parties alive.