Efficient Fully Secure Computation via Distributed Zero-Knowledge Proofs

Elette Boyle[1], Niv Gilboa[2], Yuval Ishai[3], and Ariel Nof[3]

[1] IDC Herzliya, Israel*
eboyle@alum.mit.edu
[2] Ben-Gurion Univeristy, Israel**
gilboa@bgu.ac.il
[3] Technion, Israel***
{yuvali, ariel.nof}@cs.technion.ac.il

Abstract. Secure computation protocols enable mutually distrusting parties to compute a function of their private inputs while revealing nothing but the output. Protocols with *full security* (also known as *guaranteed output delivery*) in particular protect against denial-of-service attacks, guaranteeing that honest parties receive a correct output. This feature can be realized in the presence of an honest majority, and significant research effort has gone toward attaining full security with good asymptotic and concrete efficiency.

We present an efficient protocol for *any constant* number of parties n, with full security against $t < n/2$ corrupted parties, that makes a black-box use of a pseudorandom generator. Our protocol evaluates an arithmetic circuit C over a finite ring R (either a finite field or $R = \mathbb{Z}_{2^k}$) with communication complexity of $\frac{3t}{2t+1}S + o(S)$ R-elements per party, where S is the number of multiplication gates in C (namely, < 1.5 elements per party per gate). This matches the best known protocols for the semi-honest model up to the sublinear additive term. For a small number of parties n, this improves over a recent protocol of Goyal *et al.* (Crypto 2020) by a constant factor for circuits over large fields, and by at least an $\Omega(\log n)$ factor for Boolean circuits or circuits over rings.

Our protocol provides new methods for applying the distributed zero-knowledge proofs of Boneh *et al.* (Crypto 2019), which only require logarithmic communication, for compiling semi-honest protocols into fully secure ones in the more challenging case of $t > 1$ corrupted parties. Our protocol relies on *replicated secret sharing* to minimize communication and simplify the mechanism for achieving full security. This results in computational cost that scales exponentially with n.

Our main protocol builds on a new honest-majority protocol for verifying the correctness of multiplication triples by making a *general* use of distributed zero-knowledge proofs. While the protocol only achieves the weaker notion of *security with abort*, it applies to any linear secret-sharing scheme and provides a conceptually simpler, more general, and

* Supported by ISF grant 1861/16, AFOSR Award FA9550-17-1-0069, and ERC Project HSS (852952).

** Supported by ISF grant 2951/20, ERC grant 876110, and a grant by the BGU Cyber Center.

*** Supported by ERC Project NTSC (742754), ISF grant 2774/20, NSF-BSF grant 2015782, and BSF grant 2018393.

more efficient alternative to previous protocols from the literature. In particular, it can be combined with the Fiat-Shamir heuristic to simultaneously achieve logarithmic communication complexity and constant round complexity.

1 Introduction

Protocols for secure computation [38, 19, 2, 7] enable a set of parties with private inputs to compute a joint function of their inputs while revealing nothing but the output. Secure computation protocols provide a general-purpose tool for computing on sensitive data while eliminating single points of failure.

Beyond privacy and correctness, a highly desirable feature of such protocols is *guaranteed output delivery*, also known as *full security*, where honest parties are guaranteed to receive the final output of computation. This is in contrast to weaker notions of security, such as *security with abort* or *fairness*, which leave protocols vulnerable to denial-of-service attacks.

Full security can be achieved with an honest majority, namely when there are $n \geq 3$ parties of which at most $t < n/2$ are corrupted. This holds unconditionally given secure point-to-point channels and a broadcast primitive [35] (where the latter can be realized from a public-key infrastructure using digital signatures [12]), or alternatively using only secure channels assuming $t < n/3$ [2, 7]. However, despite intensive research efforts, there is still a significant efficiency gap between the best known protocols achieving full security and those achieving weaker notions. We focus on the *communication* complexity of such protocols, which in the domain of concretely efficient protocols typically dominates overall cost. In this work, "concretely efficient" is interpreted as making only black-box use of a pseudo-random generator (PRG).[4]

A useful metric for measuring efficiency of fully secure protocols is the ratio between the communication cost of the protocol and that of the best known protocol with a "minimal" level of security, namely security against *semi-honest* parties, who act as prescribed by the protocol but try to learn additional information from messages they receive. Minimizing the overhead of full security has been the subject of a large body of work; see [26, 21, 6, 5, 23] and references therein. Here we focus on the more challenging case of a minimal honest majority ($t < n/2$). The ultimate goal is to obtain full security with the *same communication complexity* as the best known protocols that achieve semi-honest security, up to sublinear additive terms.

The most relevant state of the art toward this goal is captured by two recent works: Boyle et al. [5] in the special case of 3 parties (i.e., $n = 3, t = 1$), and Goyal et al. [23] that approaches the goal for general n.

[4] As opposed to expensive cryptographic tools such as fully homomorphic encryption [36, 17], where communication is asymptotically small but overall concrete costs are high. In the context of protecting against malicious parties, a PRG is not known to imply sublinear-communication arguments for NP in the standard setting.

For the special case of 3 parties, the fully secure protocol of Boyle et al. [5] matches the amortized cost of the best known semi-honest protocol in this setting (due to Araki et al. [1]). More specifically, the protocol from [5] evaluates an arithmetic circuit C over a finite ring R with an amortized communication cost of a *single* R-element per party per multiplication gate.[5] The protocol applies to rings R that are either finite fields or rings of the form $R = \mathbb{Z}_{2^k}$, and in particular applies to Boolean circuits with an amortized cost of just 1 bit per party per AND gate.

Very recently, Goyal et al. [23] presented a fully secure protocol for arbitrary n that applies to the case where R is a large finite field, and provides information theoretic security. In the case that parties do not deviate from the protocol, the amortized per-party communication cost is 5.5 field elements, matching that of the best known information-theoretic semi-honest protocol. However, several gaps remain to the ultimate goal. If cheating occurs, the amortized communication cost of the protocol increases to 7.5 field elements per party, above the 5.5 semi-honest baseline. Further, by allowing black-box use of PRGs in the place of information theoretic security, the semi-honest baseline can be improved. Finally, the protocol of [23] only applies to the case that R is a finite field, as opposed to more general rings, and the quoted communication complexity is achieved only when the field is large. For instance, for Boolean circuits the protocol induces an additional $\log n$ factor. Overall, removing these limitations introduces several challenges which require new techniques.

In this work, we make progress toward closing the remaining gaps, focusing our attention on the practically motivated case of a *constant* number of parties[6] n. Even in this setting, designing fully secure protocols is a challenging task. Indeed, concretely efficient protocols in an even more restricted settings of $n = 3, 4$ or 5 parties, of which only $t = 1$ may be corrupted, have been the target of several previous works (e.g., [25, 31, 33, 20, 6, 5]). However, these protocols are heavily tailored to the case $t = 1$, and there are multiple difficulties one encounters when trying to efficiently extend them to larger t.

For a constant threshold t and $n = 2t + 1$, the relevant semi-honest baseline is a protocol from [4] that optimizes a protocol of Damgård and Nielsen [11] using pseudorandom secret sharing [18, 10]. The amortized communication cost is $\frac{3t}{2t+1}$ (< 1.5) R-elements per party per multiplication gate. This sets our target communication goal for full security.

Relaxing full security to *security with abort*, this goal was recently met by Boyle et al. [4]. For the case of non-constant t, the amortized overhead of security with abort was also eliminated recently, first for $n = 3t + 1$ parties by Furukawa and Lindell [15] and then for $n = 2t + 1$ parties by Goyal and Song [22]. (A similar result, with a bigger sublinear additive term, can be obtained from the

[5] Namely, communication of $S + o(S)$ ring elements per party, where S is the number of multiplication gates in C.

[6] More generally, our main protocol incurs computation and storage costs that scale exponentially with n. However, these costs involve only symmetric cryptography and can be shifted almost entirely to an offline phase, before the inputs are known.

technique of [4].) However, in all these protocols, the parties immediately abort whenever cheating is detected. As always, the challenge of full security is in safely recovering to completion in the case corrupt parties send improper messages, withhold information, or exit the computation prematurely.

1.1 Our Contributions

Our main contribution is a secure computation protocol for *any constant* number of parties $n = 2t + 1$ that achieves full security against up to t malicious parties with the *same amortized communication* as for the best known semi-honest protocol mentioned above. Our protocol applies to both Boolean and arithmetic circuits, and even over the rings $\mathbb{Z}_{2^k}$. It uses a broadcast channel $\mathcal{F}_{\mathsf{bc}}$ (necessary to achieve full security in this setting, where broadcast is not possible without setup [34]), and makes only black-box use of a PRG. The total size of strings communicated over $\mathcal{F}_{\mathsf{bc}}$ is sublinear in the circuit size.

A basic building block in our construction is an arbitrary n-party protocol Π_{mult} for *private multiplication* based on replicated secret-sharing [27]. In such a protocol, inputs to a multiplication gate are shared by replicated secret sharing, and if all parties act honestly then in the end of the protocol the product of the inputs is also shared by the same scheme. Furthermore, even if t malicious parties act dishonestly in the protocol, they do not obtain information on the inputs of the honest parties. The usefulness of replicated secret sharing for simplifying general secure computation protocols was first pointed out by Maurer [30]. The most communication-efficient instance of a protocol of this type was given by Boyle et al. [4], combining the approach of Damgård and Nielsen [11] with the pseudorandom secret sharing technique of Cramer et al. [10] (see also [18]).

Our first result shows how to use this building block *in a generic way* to achieve full security with only sublinear additive communication overhead when no cheating occurs. When cheating does occur, there is an additional additive term that grows linearly with a circuit "width" parameter W. Intuitively, the circuit width captures the amount of space required by the computation.

At a very high level, the protocol starts by using Π_{mult} to privately compute shares of the outputs of all multiplication gates, without reconstructing them. It then ensures that these outputs are correct by applying *distributed zero-knowledge proofs*, i.e., proofs of a statement on an input that is distributed between several verifiers. Such proofs for simple languages, including the "degree-2 languages" we require, can have sublinear (in fact, logarithmic) length in the size of the statement [4], which we use to achieve low communication overhead. A major challenge that we solve is efficient recovery from failures. We achieve this by a careful combination of a player elimination approach (cf. [24]) with an authentication mechanism (cf. [35]). Our particular way of combining these techniques takes advantage of the redundancy provided by replicated secret sharing and the amortization enabled by pseudorandom secret sharing.

Using the concrete instantiation of Π_{mult} from [11, 4], we can eliminate the extra $O(W)$ additive overhead and obtain the following main result.

Theorem 1.1 (Efficient fully secure MPC for constant n). *Let R be a finite field or a ring of the form $\mathbb{Z}_{2^k}$, let $t \geq 1$ be a constant security threshold and $n = 2t + 1$. Then, assuming a black-box access to a PRG, there is a fully t-secure n-party protocol that evaluates an arithmetic circuit over R, with S multiplication gates, by communicating $\frac{3t}{2t+1}S + o(S)$ ring elements per party.*

Compared to the recent protocol from [23], this improves the worst-case amortized communication by at least a factor of 5 over big fields, and by at least a $5 \log_2 n$ factor for Boolean circuits and circuits over $\mathbb{Z}_{2^k}$. Moreover, unlike the protocol from [23], here we can match the amortized cost of the best known semi-honest protocol even when cheating occurs. However, unlike the protocol from [23], our protocol is restricted to a constant number of parties and provides computational (rather than information-theoretic) security.

The simpler case of security-with-abort. As an intermediate step in constructing fully secure protocols, we develop a protocol that is only secure-with-abort, i.e., the adversary can force the honest parties to abort without receiving an output. Unlike our main protocol, here we apply a general compilation technique that is not restricted to replicated secret sharing or a small number of parties. Instead, we give a simple protocol for verifying the correctness of secret-shared multiplication triples by making a *general* use of (sublinear-communication) distributed zero-knowledge proofs. The main difference between the triple verification task and distributed zero knowledge is that in the latter there is a prover who knows all of the (distributed) secrets, whereas in the former there is no such prover. Nevertheless, we show that triple verification can be efficiently reduced to distributed zero knowledge. The high-level idea is to view the shares held by all parties *except* P_i as a secret-sharing of the share held by P_i. This allows each party to prove to the other parties that a computation it locally performed on its shares was done correctly using distributed zero knowledge.

We stress that unlike similar verification protocols from [3, 4, 22], our approach is very general and can rely on any instantiation of the underlying distributed proofs primitives. In particular, using the distributed zero-knowledge protocols from [4, 5], the verification cost is logarithmic in the size of the circuit. This is similar to a verification procedure from [22] and better than the square-root complexity of an earlier triple verification protocol from [4]. Compared to the protocol from [22], our approach is more general, and can rely on any distributed zero-knowledge protocol for degree-2 languages, which in fact reduces to a "zero-knowledge fully-linear IOP" for such languages [4]. Another advantage of our triple verification protocol over that of [22] is that it can be combined with the Fiat-Shamir heuristic to simultaneously achieve logarithmic communication complexity and *constant* (as opposed to logarithmic) round complexity. See Section 4.3 for a detailed discussion of concrete efficiency.

As in the generic version of our main theorem, we can apply the above technique to compile any semi-honest MPC protocol that builds on a private multiplication sub-protocol into a similar protocol that achieves security-with-abort. However, in the current case the private multiplication sub-protocol Π_{mult} can

use any linear secret-sharing scheme, in particular Shamir's scheme [37]. As a result, our compiler can yield protocols that are efficient for any (super-constant) number of parties n. This is captured by the following theorem.

Theorem 1.2 (Security-with-abort compiler for any n, informal). *Let R be either a finite field or a ring of the form $\mathbb{Z}_{2^k}$, let $t \geq 1$ be a security threshold, and $n = 2t + 1$. Then, assuming a black-box access to any n-party t-private protocol Π_{mult} for multiplying linearly shared secrets over R, there is an n-party protocol Π for arithmetic circuits over R with the following security and efficiency properties. The protocol Π is t-secure-with-abort, with the same type of security (information-theoretic or computational) as Π_{mult}. It evaluates an arithmetic circuit with S multiplication gates using communication complexity of $|\Pi_{\text{mult}}| \cdot S + o_n(S)$ elements of R, where $|\Pi_{\text{mult}}|$ is the communication complexity of Π_{mult}, and o_n hides polynomial terms in n.*

Theorem 1.2 can be viewed as a more general alternative to the recent protocol from [22], which is tailored to a special kind of semi-honest protocol. Our approach is more general both in its treatment of the underlying multiplication sub-protocol and in the use of general distributed zero-knowledge proofs.

2 Preliminaries

Notation. Let $P_1, \ldots, P_n$ be the set of parties and let t be such that $n = 2t + 1$. In this work, we assume that there exists an honest majority and so the number of corrupted parties is at most t. We use $[n]$ to denote the set $\{1, \ldots, n\}$. We denote by $\mathbb{F}$ a finite field and by $\mathbb{Z}_{2^k}$ the ring of integers modulo 2^k. We use the notation R to denote a ring that can either be a finite field or the ring $\mathbb{Z}_{2^k}$. We use $[\![x]\!]$ to denote a secret sharing of x with threshold t (as defined below) and $\langle x \rangle$ to denote an additive sharing of x.

2.1 Computation Model

In this work, we model the computation that represent the functionality the parties wish to compute, as a straight-line program, with addition and multiplication instructions [9]. The advantage of this representation is that it captures the notion of *width*, which is defined to be the maximal numbers of registers required to store memory during the computation.

Definition 2.1 (Straight-line programs). *A* straight-line program over a ring *R consists of an arbitrary sequence of the four following instructions, each with a unique identifier id:*

- *Load an input into memory:* $(id, \hat{R}_j \leftarrow x_i)$.
- *Add values in memory:* $(id, \hat{R}_k \leftarrow \hat{R}_i + \hat{R}_j)$.
- *Multiply two values in memory:* $(id, \hat{R}_k \leftarrow \hat{R}_i \cdot \hat{R}_j)$.
- *Output value from memory, as element of R:* $(id, O_i \leftarrow \hat{R}_j)$.

where $x_1, \ldots, x_n$ are the inputs, $O_1, \ldots, O_n$ are the outputs and $\hat{R}_1, \ldots, \hat{R}_W$ are registers holding memory. We define the size *of a program P as the number of multiplication instructions and denote it by S. We define the* width *of P as the number of registers W.*

Every arithmetic circuit with S multiplication gates can be converted into a straight-line program of size S by sorting its gates in an arbitrary topological order. We will assume for simplicity that each party has a single input and receives a single output. Our constructions can be easily adapted to the setting of multiple inputs and outputs per party.

2.2 Threshold Linear Secret Sharing Schemes

Definition 2.2. *A t-out-of-n secret sharing scheme is a protocol for a* dealer *holding a secret value v and n parties $P_1, \ldots, P_n$. The scheme consists of two interactive algorithms:* $\mathsf{share}(v)$*, which outputs shares $[\![v]\!] = (v_1, \ldots, v_n)$ and* $\mathsf{reconstruct}([\![v]\!]_T, i)$*, which given the shares $v_j, j \in T \subseteq \{1, \ldots, n\}$ outputs v or $\perp$. The dealer runs* $\mathsf{share}(v)$ *and provides P_i with a share of the secret v_i. A subset of users T run* $\mathsf{reconstruct}([\![v]\!]_T, i)$ *to reveal the secret to party P_i by sending their shares to P_i. The scheme must ensure that no subset of t shares provide any information on v, but that $v = \mathsf{reconstruct}([\![v]\!]_T, i)$ for any T, $|T| \geq t + 1$. We say that a sharing is* $\mathsf{consistent}$ *if* $\mathsf{reconstruct}([\![v]\!]_T, i) = \mathsf{reconstruct}([\![v]\!]_{T'}, i)$ *for any two sets of honest parties $T, T' \subseteq \{1, \ldots, n\}$, and $|T|, |T'| \geq t + 1$.*

Verifiable Secret Sharing (VSS). We say that $\mathsf{share}(v)$ is verifiable if at the end of $\mathsf{share}(v)$, either the parties hold a consistent sharing of the secret or the honest parties abort. This is achieved by adding a consistency check after each party receives its shares from the dealer. We will describe consistency checks for the secret sharing schemes used in our work below.

Authenticated Secret Sharing. We say that a secret sharing scheme is *authenticated* if, assuming that the sharing phase was correctly executed, malicious parties cannot prevent the correct reconstruction of the secret by tampering with their shares. (Authenticated secret sharing is sometimes also referred to as *robust* secret sharing.) We remark that it is not straightforward to achieve this when $t \geq n/3$, as standard error-correcting techniques do not suffice. In fact, perfect reconstruction is provably impossible to achieve in this setting, and one must settle for statistically small error probability. There is a recent line of work on optimizing the efficiency of authenticated secret sharing; see [13] and references therein. However, the asymptotically good constructions are quite complex and are not attractive when the number of parties are small. In this work, we only need to make minimal use of this primitive which is independent of the size of the circuit. Thus, any implementation will suffice. An example for such a simple implementation is the well-known construction of Rabin and Ben-Or [35] based on pairwise authentication of shares.

Local linear operations and multiplication. In this work, we require that linear operations over a ring for a given secret sharing scheme can be carried out locally. In particular, given $[\![x]\!], [\![y]\!]$ and some public constant c, the parties can compute: (1) $[\![x + y]\!]$ (2) $[\![c \cdot x]\!]$ and (3) $[\![c + x]\!]$. We use the notation $[\![x]\!] + [\![y]\!]$, $c \cdot [\![x]\!]$ and $c + [\![x]\!]$ to denote the three local procedures respectively that achieve this. Thus, we have $[\![x + y]\!] = [\![x]\!] + [\![y]\!]$, $[\![c \cdot x]\!] = c \cdot [\![x]\!]$ and $[\![c + x]\!] = c + [\![x]\!]$.

While a linear secret sharing scheme does not allow multiplication of shares without interaction, we assume that given $[\![x]\!]$ and $[\![y]\!]$, the parties can locally compute $\langle x \cdot y \rangle$ (thus the interaction is required for reducing the threshold). We denote the operation of computing the product's additive sharing by $[\![x]\!] \cdot [\![y]\!]$.

Local conversion from $[\![x]\!]$ to $[\![x^i]\!]$. Given a consistent sharing $[\![x]\!]$, we require that the parties are able to locally generate a consistent sharing $[\![x^i]\!]$, where x^i is the share of x held by party P_i.

Instantiation 1: Replicated Secret Sharing [27] To share a secret $x \in R$, for each subset T of t parties the dealer hands a random share x_T to the parties in $\overline{T} = \{P_1, \ldots, P_n\} \setminus T$, under the constraint that $x = \sum_{T \subset \{P_1,\ldots,P_n\}:\ |T|=t} x_T$. The share held by each party P_i is the tuple consisting of all x_T such that $P_i \in T$. Thus, the number of shares is $\binom{n}{t}$ and each party holds $\binom{n-1}{t}$ shares.

It is easy to see that replicated secret sharing scheme is linear over R and allows local multiplication to obtain an additive sharing of the product when $t < n/2$. Local conversion from $[\![x]\!]$ to $[\![x^i]\!]$ can be done by sharing each component x_T that P_i holds separately. For each T for which $P_i \in T$, every party $P_j \in T$ will hold x_T, while parties not it T will set their share to be 0.

Pairwise consistency. Observe that since $n = 2t + 1$ in our setting, each share is held by a subset of $t + 1$ parties. Thus, a sharing is inconsistent if a cheating dealer hands different values to honest parties in the same subset. In order to verify that a sharing is consistent, it suffices that every pair of parties verify that they hold the same share for each subset T, which includes both parties. This can be done with low communication by having these parties compare a hash of their joint shares. Observe that if pairwise inconsistency is detected then this pair can ask the dealer to publish the conflicted share, as in this case, this share is already known to the adversary.

Instantiation 2: Shamir's Secret Sharing [37] In this well-known scheme, the dealer defines a random polynomial $p(x)$ of degree t over a finite field $\mathbb{F}$ such that the constant term is the secret. Each party is associated with a distinct non-zero field element $\alpha \in \mathbb{F}$ and receives $p(\alpha)$ as its share of the secret. Linear operations on secrets can be computed locally on the shares, since polynomial interpolation is a linear operation. In addition, given shares of x and y, the parties can locally multiply their shares to obtain a sharing of degree $2t$ of $x \cdot y$.

Finally, observe that since each share is a point on a polynomial, then a consistent sharing $[\![x]\!]$ is also a consistent sharing of P_i's share x^i, written as $[\![x^i]\!]$ (the only difference is that now the secret is not stored at the point 0 but at the point α_i).

Polynomial consistency. A Shamir secret sharing is consistent if all shares $(p(\alpha_1) = \beta_1, \ldots, p(\alpha_n) = \beta_n)$ lie on the same degree-t polynomial. A simple way to check the consistency of m sharings: $(\beta_{1,1}, \ldots, \beta_{1,n})$,...,$(\beta_{m,1}, \ldots, \beta_{m,n})$ together in a batch is to generate n random coefficients $c_1, \ldots, c_n \in \mathbb{F}$ and a random degree-t

polynomial $q(x)$, compute $(\sum_{i=1}^{m} c_i\beta_{i,1} + q(\alpha_1), \ldots, \sum_{i=1}^{m} c_i\beta_{i,n} + q(\alpha_n))$, open the shares, and check that they lie on a degree t polynomial.

We stress that Shamir's scheme can be used only in our base secure-with-abort construction. The fully secure construction relies on properties that hold only for replicated secret sharing.

2.3 Π_{mult} – Private Multiplication Protocol

In our main protocol, the parties first compute each multiplication instruction using a protocol Π_{mult} that satisfies only the following a weak notion of security and then run a verification protocol to detect and recover from cheating.

Definition 2.3. *Let Π_{mult} be an n-party protocol that takes as inputs $[\![x]\!]$ and $[\![y]\!]$ and outputs $[\![z]\!]$. We say that Π_{mult} is a* private *multiplication protocol in the presence of a malicious adversary controlling up to t parties if it satisfies two properties.*

Correctness *If $[\![x]\!]$ and $[\![y]\!]$ are consistent sharings and all the parties follow the protocol's instructions, then $[\![z]\!]$ is a consistent sharing of $z = x \cdot y$.*

Privacy *Denote the set of honest parties by J and denote the vector of all input shares held by the honest parties by $\boldsymbol{u}_J$. Then, for every adversary $\mathcal{A}$ controlling up to t parties, and for every two vectors of shares $\boldsymbol{u}_J, \boldsymbol{u}'_J$ the view that $\mathcal{A}$ has in the protocol when the honest parties hold $\boldsymbol{u}_J$ is computationally indistinguishable from its view when the honest parties hold $\boldsymbol{u}'_J$.*

We say that Π_{mult} is a replicated *and private multiplication protocol if in addition to the correctness and privacy properties it holds that if $[\![x]\!]$ and $[\![y]\!]$ are consistent sharings of x and y in a replicated secret sharing scheme for threshold t, and all the parties follow the protocol's instructions, then $[\![z]\!]$ is a consistent sharing of $z = x \cdot y$ in the same replicated secret sharing scheme for threshold t.*

The latter property in the above definition will be used in our fully secure construction.

Instantiation: The DN [11] multiplication protocol In the DN protocol, the parties prepare in advance two random sharings $[\![r]\!], \langle r \rangle$ which are used in the following way. First, the parties locally compute $\langle x \cdot y - r \rangle = [\![x]\!] \cdot [\![y]\!] - \langle r \rangle$ and send the result to P_1. Then, P_1 reconstructs $x \cdot y - r$ and sends it back to the parties. The parties then locally compute $[\![x \cdot y]\!] = x \cdot y - r + [\![r]\!]$. A simple optimization to the second step is having P_1 share $x \cdot y - r$ to the parties instead of sending it in the clear. Then, we can let the shares of t parties be 0 and let the shares of the remaining parties be computed given the value of $xy - r$ and the t zero shares (for replicated secret sharing this translates into having the share given to one subset of $t + 1$ parties being $x \cdot y - r$ and the remaining shares being 0). Thus, we can have P_1 send $xy - r$ to t parties, and then P_1 and these t parties can locally compute their shares of $xy - r$ and add them to their shares of r, while the remaining parties set their output to be their shares of r. Thus, the overall communication in the online step is $n - 1 + t$ elements, and

so $\frac{2t+1-1+t}{2t+1} \leq 1.5$ elements per party. The masking of all sent messages in this protocol with random value guarantees that the protocol satisfies the privacy requirement. For the offline step, it is possible to produce $[\![r]\!], \langle r \rangle$ without any interaction [10] or using interaction but with reduced computational overhead for large number of parties [11] (using hyper-invertible matrices). We refer the reader to [8, 29] for exact analysis.

In the full version of this paper, we describe other instantiations for Π_{mult} that can be usefull in some settings.

2.4 Other Basic Ideal Functionalities

Let $\mathcal{F}_{\text{rand}}(t)$ be an ideal functionality that hands the parties a sharing of a random secret value with threshold t, while allowing the adversary to choose the corrupted parties' shares. This functionality can be realized for both Shamir and the replicated secret sharing scheme [10, 11]. We remark that for replicated secret sharing, the functionality can be realized without any interaction (except for a setup step) [10], which makes the protocol fully secure. This is of high importance for our fully secure construction.

Let $\mathcal{F}_{\text{coin}}$ be an ideal functionality that hands the parties fresh random coins. In the security with abort model, it can be realized by calling $\mathcal{F}_{\text{rand}}$ and opening the result. To achieve full security, heavier machinery is required. Nevertheless, we can reduce the number of calls to this functionality to the size of the security parameter (as it is possible to call it only to generate a seed r from which all the required randomness is derived.

Finally, Let $\mathcal{F}_{\mathsf{bc}}$ be a secure broadcast functionality which allows the parties to broadcast a message to all the other parties. We remark that use of a broadcast channel is necessary to achieve full security within this setting, where broadcast is not possible without setup [34]. Full security of $\mathcal{F}_{\mathsf{bc}}$ is achievable given PKI setup [35]. The number of times this functionality is called will be sublinear in the size of the circuit and so any reasonable implementation will suffice.

3 Prove Correctness of Degree-2 Relations Over Shared Data

In this section, we present the main building block for our constructions: a protocol that allows the parties to prove that a degree-2 computation over their shares was carried-out correctly. Specifically, in our protocol, we have a party P_i who wishes to prove that the following equation holds:

$$c - \sum_{k=1}^{L} (a_k \cdot b_k) = 0 \tag{1}$$

where c, $\{a_k\}_{k=1}^{L}$ and $\{b_k\}_{k=1}^{L}$ are known to P_i and are secret shared among the parties via a consistent t-out-of-n linear secret sharing scheme (see Definition 2.2). We note that the above task can be seen as an application of the distributed zero-knowledge proof system defined in [4]. In the setting of distributed zero-knowledge proofs there is a prover who wishes to prove a statement in zero-knowledge, where the statement is held in a distributed manner across multiple

verifiers. An example for a statement that is distributed across verifiers, is our setting in which the statement is secret shared among the verifiers. As in any zero-knowledge proof system, the definition of distributed zero-knowledge interactive proofs requires that three properties will be satisfied: *completeness* (if the statement is correct and the parties follow the protocol, then the verifiers will output accept with probability 1), *soundness* (if the statement is incorrect, then the honest verifiers will output accept only with a small probability) and *zero-knowledge* (no information about the inputs is leaked during the execution). However, in distributed zero-knowledge proof protocols, the above requirements should be met even if the prover colludes with a subset of verifiers. As shown in [4], for low-degree relations it is possible to construct zero-knowledge proof protocols with sub-linear communication complexity. In Section 3.1, we rely on one of their ideas to design a highly-efficient protocol to prove that Eq. (1) holds. In Section 3.2 we take a step further and provide a protocol where an honest prover can also identify a cheating verifier in case the proof is rejected.

3.1 The Functionality $\mathcal{F}^{\text{abort}}_{\text{proveDeg2Rel}}$ - Prove Correctness with Abort

We begin with a protocol that is secure with abort, i.e., it allows a malicious verifier to cause honest parties to reject even when the statement is correct. In this section, we assume that the prover knows also $[\![c]\!]$ (i.e., the shares of all parties of c)[7]. In contrast, for the a_ks and b_ks, P_i does not need to know the other parties' shares, and in fact, in this case, P_i's share is the secret itself. We compute the ideal functionality $\mathcal{F}^{\text{abort}}_{\text{proveDeg2Rel}}$. The functionality checks that Eq. (1) holds using the honest parties' shares. This is sufficient since in the honest majority setting, the honest parties' shares fully determine both the secret and the corrupted parties' shares. Observe that in case the equation holds, $\mathcal{F}^{\text{abort}}_{\text{proveDeg2Rel}}$ lets the adversary determine the output (i.e., accept or reject) for each party, whereas if the equation does not hold, the output is always reject. Note also that in case the prover is corrupt, $\mathcal{F}^{\text{abort}}_{\text{proveDeg2Rel}}$ hands the adversary $\mathcal{S}$ also the inputs, and all shares of c (since these are known anyway to the real world adversary).

Computing $\mathcal{F}^{\text{abort}}_{\text{proveDeg2Rel}}$ using distributed zero-knowledge proofs. While the definition of $\mathcal{F}^{\text{abort}}_{\text{proveDeg2Rel}}$ yields a setting which is similar to the setting of distributed zero-knowledge proofs defined in [4], there is still one difference. The zero-knowledge property in the definition of [4] considers only privacy in the presence of a subset of verifiers. Here however we assume that the prover does not know the verifiers' shares of the a_ks and b_ks. Thus, the proof protocol must also prevent the prover from learning any information on these shares. Thus, any distributed zero-knowledge proof used to realize $\mathcal{F}^{\text{abort}}_{\text{proveDeg2Rel}}$ must provide this stronger requirement.

A concrete protocol to compute $\mathcal{F}^{\text{abort}}_{\text{proveDeg2Rel}}$. We next show how to compute this functionality using the fully linear interactive oracle proof from [4] with low communication. The idea works as follows. First, the parties define a g-gate $g(\nu_1, \ldots, \nu_L) = \sum_{\ell=1}^{L/2} \nu_{2\ell-1} \cdot \nu_{2\ell}$. We now can write Eq. (1) as

[7] It is possible to avoid this assumption, but it nevertheless holds for our verification protocol that uses this proof as a building block.

FUNCTIONALITY 3.1 ($\mathcal{F}^{\text{abort}}_{\text{proveDeg2Rel}}$- Prove Correctness of a Shared Secret)
Let $\mathcal{S}$ be the ideal world adversary controlling a subset $< n/2$ of corrupted parties.
The functionality $\mathcal{F}^{\text{abort}}_{\text{proveDeg2Rel}}$ works with $\mathcal{S}$ and honest parties holding consistent t-out-of-n secret sharings $[\![c]\!], \{[\![a_k]\!]\}_{k=1}^{L}, \{[\![b_k]\!]\}_{k=1}^{L}$.
$\mathcal{F}^{\text{abort}}_{\text{proveDeg2Rel}}$ is invoked by an index i sent from the honest parties and works as follows:

1. $\mathcal{F}^{\text{abort}}_{\text{proveDeg2Rel}}$ receives from the honest parties their shares of c, $\{a_k\}_{k=1}^{L}$ and $\{b_k\}_{k=1}^{L}$.
2. $\mathcal{F}^{\text{abort}}_{\text{proveDeg2Rel}}$ computes c, $\{a_k\}_{k=1}^{L}$ and $\{b_k\}_{k=1}^{L}$. Then, it computes the corrupted parties' shares of these values and sends them to $\mathcal{S}$. If P_i is corrupted, then it sends also $[\![c]\!]$, $\{a_k\}_{k=1}^{L}$ and $\{b_k\}_{k=1}^{L}$ to $\mathcal{S}$.
3. $\mathcal{F}^{\text{abort}}_{\text{proveDeg2Rel}}$ checks that Eq. (1) holds.
   If it holds, then it sends accept to $\mathcal{S}$ to receive back $\mathsf{out}_j \in \{\mathsf{accept}, \mathsf{reject}\}$ for each honest party P_j, which is handed to party P_j.
   Otherwise, it sends reject to $\mathcal{S}$ and the honest parties.

$$c - g\left(a_1, b_1, \ldots, a_{L/2}, b_{L/2}\right) - g\left(a_{L/2+1}, b_{L/2+1}, \ldots, a_L, b_L\right) = 0.$$

Next, the prover P_i, who knows all inputs, computes the output of the two g gates and verifiably secret shares them to the parties. Let $g_1 = g\left(a_1, b_1, \ldots, a_{L/2}, b_{L/2}\right)$ and $g_2 = g\left(a_{L/2+1}, b_{L/2+1}, \ldots, a_L, b_L\right)$. Thus, the parties hold now a t-out-of-n secret sharing of c, g_1 and g_2. Hence, the parties can locally compute $[\![b]\!] = [\![c]\!] - [\![g_1]\!] - [\![g_2]\!]$ and check that $b = 0$ by revealing their shares of b. Since an honest majority exists, the adversary cannot do any harm in the opening beyond causing the parties to abort. However, this is not enough; a corrupted P_i may cheat when sharing g_1 and g_2. To prevent this, the parties carry-out an additional test. Let $f_1, \ldots f_L$ be polynomials defined in the following way: for each $e \in [L]$, $f_e(1)$ is the eth input to the first g-gate, and $f_e(2)$ is the eth input to the second g-gate. It follows that f_e is a linear function (i.e., polynomial of degree-1). Next, define the polynomial $q(x) = g(f_1(x), \ldots, f_L(x))$. From the definition of q, it follows that: (1) $q(1)$ is the output of the first g-gate and $q(2)$ is the output of the second; (2) q is of degree-2 (since g is a circuit of of multiplicative depth-1 and the f polynomials are of degree-1). Now, to check that P_i shared the correct $q(1)$ and $q(2)$, it suffices to check that $q(r) = g(f_1(r), \ldots, f_L(r))$ for some random r in the ring/field. To carry-out the check, the parties can locally compute a t-out-of-n secret sharings of $q(r)$ and $f_1(r), \ldots, f_L(r)$ via Lagrange interpolation over their shares (note that this is a local linear operation), open these sharings and check the equality in the clear. This requires that P_i will share also $q(3)$, so that the parties have enough points on q (and so r cannot be in $\{1, 2, 3\}$)). Note however that opening L shares results with communication cost that is linear in L. To achieve communication that is logarithmic in L, instead of opening, we let P_i prove that

$$q(r) - g(f_1(r), \ldots, f_L(r)) = 0 \tag{2}$$

by repeating *the exact same process as above*. This is possible since Eq. (2) has the same form as Eq. (1) and since all parties hold a consistent sharing of all the inputs to Eq. (2). Note that this time we only have L inputs (instead of $2L$). Thus, the parties can repeat the process $\log L$ times, until there are only small constant number of inputs and then check equality to 0 by opening. One subtle security issue that arise here is that $f_e(r)$ is a linear combination of inputs. Thus, to securely open it, the parties randomize the f polynomials by adding (only in the last step) a random point to each polynomial. This is achieved by using $\mathcal{F}_{\text{rand}}$ to generate an additional shared point for each of f polynomials. Note that the degree of q is now 4 (since the degree of f was increased to 2) and so P_i needs to share 5 points on q instead of 3. As an additional optimization, we also deffer the check of equality to 0 of the b values to the end, and then perform a single check by taking a random linear combination of all b values generated in each step of the recursion. As we will argue below, the cost per step in the recursion is constant, and so since we have $\log L$ steps, the overall communication cost is logarithmic in L. The protocol is formalized in Protocol 3.2.

Cheating probability for finite fields. We now compute the probability that the parties output accept, even though Eq. (1) does not hold, when the protocol is executed over finite fields. Note that for this to hold, the prover P_i has two choices: (i) not to cheat in the protocol, hoping that the linear combination of the b values will yield 0. This will happen with probability $\frac{1}{\mathbb{F}}$; (ii) cheat when sharing the points on the polynomial q. This means that $q \neq g(f_1, \ldots, f_L)$ and so the polynomial $h(x) = q(x) - g(f_1(x), \ldots, f_L(x))$ is not the zero polynomial. Thus, by the Schwartz–Zippel lemma, the probability that $h(r) = 0$ for a randomly chosen $r \in \mathbb{F} \setminus \{1, 2, 3\}$ is bounded by $\frac{2}{|\mathbb{F}|-3}$ (since the degree of the polynomial h is 2) in the first $\log L - 1$ rounds and $\frac{4}{|\mathbb{F}|-5}$ in the last round (since then the degree of h is 4). Observe that for the prover to successfully cheat, this event should happen in *one* of the iterations of the protocol. Thus, the overall cheating probability is bounded by $\frac{2(\log L-1)}{|\mathbb{F}|-3} + \frac{4}{|\mathbb{F}|-5} < \frac{2\log L+4}{|\mathbb{F}|-5}$. Finally, note that $\frac{1}{\mathbb{F}} < \frac{2\log L}{|\mathbb{F}|-3}$ and so a malicious prover will increase its success cheating probability by cheating as in (ii). If the field is not large enough to achieve the desired level of security, the parties can repeat the protocol several times.

We prove that Protocol 3.2 securely computes $\mathcal{F}^{\text{abort}}_{\text{proveDeg2Rel}}$ in the full version of the paper.

Extending the protocol to the ring $\mathbb{Z}_{2^k}$. The main challenge in extending the verification protocol to rings, and in particular the ring $\mathbb{Z}_{2^k}$, is that we require interpolation and not all elements in a ring have an inverse. To overcome this, the solution suggested in [4, 5] is to work over the extension ring $\mathbb{Z}_{2^k}[x]/f(x)$, i.e., the ring of all polynomials with coefficients in $\mathbb{Z}_{2^k}$ working modulo a polynomial f that is of the right degree and is irreducible over $\mathbb{Z}_2$. As shown in [4, 5], this enables to define enough points on the polynomial that allow interpolation.

PROTOCOL 3.2 (Securely Computing $\mathcal{F}^{\text{abort}}_{\text{proveDeg2Rel}}$)

– **Inputs:** Prover P_i holds $2L+1$ inputs $c, \{a_k\}_{k=1}^{L}, \{b_k\}_{k=1}^{L}$. The parties hold a consistent t-out-of-n secret sharing of each of these inputs. P_i knows all shares of c.

– **The protocol:**

1. The parties set $\bar{L} = L$.
2. For $l = 1$ to $\log \bar{L} - 1$:
   (a) The parties define linear polynomials $f_1, f_2 \ldots, f_L$ such that for each $e \in [L]$ the polynomial f_e is defined by the two points:

$$f_e(1) = \begin{cases} a_{\lceil \frac{e}{2} \rceil} & \text{if } e \bmod 2 = 1 \\ b_{\frac{e}{2}} & \text{if } e \bmod 2 = 0 \end{cases} \qquad f_e(2) = \begin{cases} a_{\frac{L}{2}+\lceil \frac{e}{2} \rceil} & \text{if } e \bmod 2 = 1 \\ b_{\frac{L}{2}+\frac{e}{2}} & \text{if } e \bmod 2 = 0 \end{cases}$$

   (b) Let $q(x) = g(f_1(x), \ldots, f_L(x))$ be a polynomial of degree 2, where

$$g(f_1(x), \ldots, f_L(x)) = \sum_{\ell=1}^{L/2} f_{2\ell-1}(x) \cdot f_{2\ell}(x).$$

   Then, P_i locally computes $q(1), q(2), q(3)$ and verifiably secret shares (VSS) them to the other parties (If the check consistency fails for some party, then it outputs reject).
   (c) The parties locally compute $[\![b_l]\!] = [\![c]\!] - [\![q(1)]\!] - [\![q(2)]\!]$ and store the result.
   (d) The parties call $\mathcal{F}_{\text{coin}}$ to receive a random $r \in R \setminus \{1, 2, 3\}$.
   (e) The parties locally compute $[\![q(r)]\!]$ and $[\![f_1(r)]\!], \ldots, [\![f_L(r)]\!]$ via Lagrange interpolation.
   (f) The parties set $c \leftarrow q(r)$, and $\forall k \in [L/2] : a_k \leftarrow f_{2k-1}(r), b_k \leftarrow f_{2k}(r)$ and $L \leftarrow L/2$.
3. The parties exit the loop with $L = 2$ and inputs c, a_1, a_2, b_1, b_2 that are known to P_i and are secret shared among the parties. Then:
   (a) The parties call $\mathcal{F}_{\text{rand}}$ to receive $[\![w_1]\!]$ and $[\![w_2]\!]$, where $w_1, w_2 \in R$ are P_i's shares. Then, they define two polynomials f_1, f_2 of degree-2 such that: $f_1(0) = w_1, f_1(1) = a_1, f_1(2) = a_2$ and $f_2(0) = w_2, f_2(1) = b_1, f_2(2) = b_2$.
   (b) Party P_i defines a polynomial $q(x) = g(f_1(x), f_2(x))$ where $g(f_1(x), f_2(x)) = f_1(x) \cdot f_2(x)$. Thus, q is of degree-4. Then, P_i computes $q(0), q(1), \ldots, q(4)$.
   (c) Party P_i verifiably secret shares (VSS) the points $q(0), q(1), \ldots, q(4)$ to the other parties (If the check consistency fails for some party, then it outputs reject).
   (d) The parties locally compute $[\![b_{\log L}]\!] = [\![c]\!] - [\![q(1)]\!] - [\![q(2)]\!]$.
   (e) The parties call $\mathcal{F}_{\text{coin}}$ to receive random $r, \gamma_1, \ldots \gamma_{\log L} \in R$.
   (f) The parties locally compute $[\![b]\!] = \sum_{l=1}^{\log L} \gamma_l \cdot [\![b_l]\!]$.
   (g) The parties locally compute $[\![f_1(r)]\!], [\![f_2(r)]\!]$ and $[\![q(r)]\!]$ via Lagrange interpolation.
   (h) The parties run reconstruct$([\![b]\!], j)$, reconstruct$([\![q(r)]\!], j)$, reconstruct$([\![f_1(r)]\!], j)$ and reconstruct$([\![f_2(r)]\!], j)$ for each $j \in [n]$. If any party received $\perp$ in any of these executions or if $b \neq 0$ or $q(r) \neq f_1(r) \cdot f_2(r)$, then it outputs reject. Otherwise, the parties output accept.

We note that the cheating probability when working with the extension ring and hence the statistical error of the protocol is different, since the number of roots of a polynomial defined over a ring, is larger than its degree. For a program with m multiplication instructions, the error will be roughly $\frac{2\log m+4}{2^d}$, where d is the extension degree. We refer the reader to [4, 5] for more details. Nevertheless, the main observation here is that the communication when using this solution blows up only by a *constant*, and so asymptotically the complexity remains the same.

Cost Analysis In the first $\log L - 1$ iterations, the prover shares 3 ring elements in each iteration. In the last round, the prover shares 5 elements, followed by opening 4 shared elements. Using a PRG, it is possible to share a secret by sending $t \approx n/2$ ring elements, and opening a secret requires transmission of n^2 elements. To realize $\mathcal{F}_{\text{coin}}$ (with abort) it suffices to open a random sharing. Hence, in this case, the overall communication cost per party is

$$(1.5 + n - 1)\log(L-1) + 2.5 + 4(n-1) \approx n \cdot \log(L) + 4n \quad \text{field elements.}$$

The asymptotic communication complexity is thus $O(n \log L + n)$. When the verified shared triples are defined over a *ring*, then the cost is multiplied with the degree of extension d. We ignore here the cost of consistency checks (in the VSS protocol) that can typically be batched together with a small constant cost.

For the computational cost, we remark that while our protocol requires many interpolations, all polynomials used in the protocol are of small degree (up to 4). Thus, the number of operations (i.e., multiplications and additions) required for each interpolation is a small constant. The number of polynomials that we have in the protocol is $L + 1$ in the first iteration, $L/2 + 1$ in the second, $L/4 + 1$ in the third and so on. Over $\log L$ iterations, we thus have $O(L)$ polynomials and so the overall computational cost is also $O(L)$ operations.

A Constant-Round Protocol using the Fiat-Shamir Transform The number of rounds in Protocol 3.2 is logarithmic in the size of the input. In the full version of the paper, we show how to use the Fiat-Shamir transform [14] to reduce interaction and achieve constant number of rounds. This transform applies to public-coin protocols and proceeds by letting the prover generating the challenge in each round on its own, by applying a random oracle $H : \{0,1\}^* \to \{0,1\}^\kappa$ to the concatenation of the messages exchanged so far. In our protocol, the prover secret shares 3 elements in each round. This means that the random oracle should be applied on the shares sent to all the parties. This seems problematic, since the shares are private information which cannot be revealed, and so the verifiers have no way to compute the public randomness. Nevertheless, we show how to solve it by changing slightly the protocol.

Batching n Proofs Together In our protocols, we will call Protocol 3.2 n times in parallel, each time for one of the parties participating in the multi-party computation. Naively, this means that the communication cost per party will be $O(n^2 \log L + n^2)$. We now show how to batch together these n proofs, reducing the cost to $O(n \log L + n)$.

To reduce the term $O(n^2 \log L)$ to $O(n \log L)$, one simply need to call $\mathcal{F}_{\text{coin}}$ once for each round of the n proofs. The parties can jointly generate a seed from which all the randomness is derived.

To reduce the term $O(n^2)$ to $O(n)$, recall first that in our proof the parties perform two tests: (i) they check that $b = 0$ and (ii) they check that $q(r) = f_1(r) \cdot f_2(r)$. These checks are carried-out by opening the secret shared $b, f_1(r), f_2(r)$ and $q(r)$ and checking that (i) and (ii) hold in the clear.

It is immediate to see that the first check can be compressed to one single check by taking a random linear combination of the b values in n proofs and opening the result. For the second check, we observe that verifying (ii) across n proofs is equivalent to check the correctness of n multiplication triples. This can be done in $O(n)$ complexity and $O(1)$ rounds via the verification technique of [32]. We present the details in the full version, where we show that the overall communication per party for running n proofs in parallel is

$$n \log L + 8n \quad \text{field elements}$$

and the asymptotic complexity is $O(n \log L + n)$ as required.

3.2 The Ideal Functionality $\mathcal{F}^{\text{cheatIdntfy}}_{\text{proveDeg2Rel}}$- Prove Correctness with Cheating Identification

In this section, we augment our protocol to prove degree-2 relations over shared data to achieve an additional property: if the protocol ends with the parties rejecting the proof, then in addition to reject, the parties will also output a pair of parties, with the guarantee that one of these parties belongs to the set of corrupted parties. Our protocol computes the ideal functionality $\mathcal{F}^{\text{cheatIdntfy}}_{\text{proveDeg2Rel}}$ defined in Functionality 3.3. The functionality works the same as the $\mathcal{F}^{\text{abort}}_{\text{proveDeg2Rel}}$ functionality defined in the previous section, with one addition: in case the output is reject, it outputs a pair of parties' indices. These contain the index of the prover and of an additional party chosen by the ideal world adversary $\mathcal{S}$. If P_i is corrupted, then $\mathcal{S}$ is allowed to pick any party it wishes. Otherwise, it must pick an index of a corrupted party. This ensures that one of the chosen parties is corrupted: in the first case, it is the prover, whereas in the second case $\mathcal{S}$ hands a corrupted party's index. Note also that in this functionality, unlike $\mathcal{F}^{\text{abort}}_{\text{proveDeg2Rel}}$, all honest parties output the same output.

To compute functionality we use Protocol 3.2 from the previous section, with one additional step: in case that the parties reject the proof, the prover is asked to identify a party who cheated in the execution. Then, the pair of parties outputted by the protocol includes the prover and the party that was pointed at by the prover. Clearly, if the prover is corrupted, then regardless of the party it chooses, the output pair will contain a corrupted party. However, it is not clear how an honest prover will identify a party who cheated in the protocol (note that in this case, we know that the degree-2 relation holds, and so if the protocol ends with a reject, then it means that someone sent incorrect messages during the execution of the proof-of-correctness protocol). To allow an honest prover to correctly identify cheaters, we require the following additional property

FUNCTIONALITY 3.3 ($\mathcal{F}^{\text{cheatIdntfy}}_{\text{proveDeg2Rel}}$**- Prove Correctness - Identify Cheating)**
Let $\mathcal{S}$ be the ideal world adversary controlling a subset $< n/2$ of corrupted parties. The functionality $\mathcal{F}^{\text{cheatIdntfy}}_{\text{proveDeg2Rel}}$ is invoked by an index i sent from the honest parties and works exactly as $\mathcal{F}^{\text{abort}}_{\text{proveDeg2Rel}}$ with the following modification:
If Eq. (1) holds, then $\mathcal{F}^{\text{cheatIdntfy}}_{\text{proveDeg2Rel}}$ sends accept to $\mathcal{S}$, to receive back out $\in$ {accept, reject}. Then, $\mathcal{F}^{\text{cheatIdntfy}}_{\text{proveDeg2Rel}}$ sends out to the honest parties. If Eq. (1) does not hold, then $\mathcal{F}^{\text{cheatIdntfy}}_{\text{proveDeg2Rel}}$ sends reject to the honest parties.
If the output handed to the honest parties is reject:

- If P_i is corrupted, then $\mathcal{S}$ sends an index $j \in [n]$ to $\mathcal{F}^{\text{cheatIdntfy}}_{\text{proveDeg2Rel}}$.
  If P_i is honest, then $\mathcal{S}$ send an index j where P_j is corrupted.
- $\mathcal{F}^{\text{cheatIdntfy}}_{\text{proveDeg2Rel}}$ sends the pair (i, j) to the honest parties.

from our protocol: *the shares held by the parties should be known to the prover.* To leverage this property, we first observe the following fact:

Fact 3.4 *Each message sent by each verifier P_j in Protocol 3.2 is a deterministic function of (1) messages received from the prover P_i; (2) its inputs to the protocol; and (3) randomness received from $\mathcal{F}_{\text{coin}}$ and $\mathcal{F}_{\text{rand}}$.*

This implies that if the inputs of all parties and the randomness chosen during the execution are known to P_i, then it can compute by himself the messages that should be sent by the other parties, and so P_i can identify cheating parties that send incorrect messages. We stress that this fact does not mean that P_i knows *in advance* what messages should be sent in the execution, since these depend on randomness received in the execution only *after* P_i sends his messages. Thus, knowing the shares held by all parties does not break the soundness of the protocol, which rely on the randomness of the evaluated point r - randomness which P_i cannot predict.

Our protocol is described and proved in the full version of the paper. It is identical to Protocol 3.2 with the following modifications in the last steps: (i) the random sharings of $f_1(0)$ and $f_2(0)$ are now verifiably secret shared by P_i (this is allowed since P_i knows now all the inputs and essential to achieve the property of P_i knowing the messages that should be sent by all other parties); (ii) the messages to reconstruct the secrets are now broadcast (to ensure anonymous output) and (iii) if the parties reject the proof, the prover P_i identify a cheating party and broadcasts its index to the other parties.

Batching n proofs together and communication cost. In section 3.1 we showed a way to batch n proofs together when only security with abort is considered. This enabled us to reduce communication complexity of n proofs ran in parallel from $O(n^2 \log L + n^2)$ to $O(n \log L + n)$ elements sent per party. While the optimization to reduce the term $O(n^2 \log L)$ to $O(n \log L)$ can be used here as well (call $\mathcal{F}_{\text{coin}}$ once for each round for all protocols), we note that it is impossible to batch all

the checks at the end of the protocol together, since then the prover will lose the ability to identify cheaters.

Thus, the communication cost of running n proofs together per party is

$$n \log L + 4n \cdot |\mathcal{F}_{\mathsf{bc}}| \quad \text{field elements.}$$

4 Secure Computation of any Straight-Line Program with Abort

In this section we present a base construction, which is only secure with abort. Given a straight-line program P, the protocol computes $P(x)$ in two stages. It first executes a protocol which computes $P(x)$ using a private multiplication protocol, as defined in Section 2.3. It then runs a verification protocol which requires communication that is sublinear in the program's size S. If the verification protocol accepts then the value of $P(x)$ is correct, while if the verification protocol rejects then the honest parties abort the protocol.

The protocol can be based on any linear threshold secret-sharing as defined in Section 2.2 and works for both finite fields and the ring $\mathbb{Z}_{2^k}$. When instantiating the protocol with Shamir's secret sharing scheme, the obtained protocol matches the complexity achieved by the protocol of [22] for finite fields and arbitrary number of parties. When using replicated secret sharing as the underlying secret sharing scehme, the obtained protocol improves upon the result of [4] for constant number of parties over the ring $\mathbb{Z}_{2^k}$; while the additive sub-linear term in [4] is square root of the size of the program, in our protocol it is *logarithmic* in the program's size.

4.1 Verifying Correctness of Multiplications with Abort

In this section, we show how the parties can verify correctness of many multiplication triples with sub-linear communication complexity in the number of triples. A multiplication triple in a ring R is a secret shared tuple $[\![x]\!], [\![y]\!], [\![z]\!]$ such that $z = x \cdot y$. In other words, a triple shares both the inputs and the output of a multiplication instruction.

At the beginning of the protocol, the parties hold sharings of many multiplication triples denoted by $([\![x_1]\!], [\![y_1]\!], [\![z_1]\!]), \ldots, ([\![x_m]\!], [\![y_m]\!], [\![z_m]\!])$ and want to verify that $z_i = x_i \cdot y_i$ for each $i \in [m]$. The ideal functionality we compute is defined in Functionality 4.1. Observe that it allows the ideal world adversary $\mathcal{S}$ to force rejection even if all triples are correct. In contrast, if there exists a triple which is incorrect, then the output will always be reject. Note also that $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{abort}}$ hands $\mathcal{S}$ the corrupted parties' shares of all triples and the additive difference $d_k = z_k - x_k \cdot y_k$ when $d_k \neq 0$ (i.e., the triple is incorrect). This is justified by the fact that, as we will see, these are known anyway to the adversary in the main protocol that works in the $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{abort}}$-hybrid model. Moreover, in many private multiplication protocols, the adversary is even allowed to choose the additive difference (see [16, 29, 8]).

FUNCTIONALITY 4.1 ($\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{abort}}$- Verify Correctness of Multiplications)
Let $\mathcal{S}$ be the ideal world adversary controlling a subset of $< n/2$ corrupted parties. The functionality $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{abort}}$ is invoked by the honest parties sending their shares of m multiplication triples $\{(x_k, y_k, z_k)_{k=1}^m\}$ to $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{abort}}$.
Then, $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{abort}}$ computes all secrets and the corrupted parties' shares which are sent to $\mathcal{S}$.
Then, it checks that $z_k = x_k \cdot y_k$ for all $k \in [m]$. If this holds, it sends accept to $\mathcal{S}$. In this case, it waits for $\mathcal{S}$ to send $\mathsf{out}_j \in \{\mathsf{accept}, \mathsf{reject}\}$ which is then handed to the honest party P_j. Otherwise, $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{abort}}$ sends reject to $\mathcal{S}$ and the honest parties. In addition, it sends $d_k = z_k - x_k \cdot y_k$ for each $k \in [m]$ for which $d_k \neq 0$ to $\mathcal{S}$.

To compute this functionality efficiently, the parties take a random linear combination $\beta = \sum_{k=1}^m \theta_k \cdot (z_k - x_k \cdot y_k)$ (where θ_k is random and jointly chosen by the parties) and wish to check that $\beta = 0$. Observe that since β is a 2-degree function of $\{(x_k, y_k, z_k)_{k=1}^m\}$, and these are secret shared via a linear threshold scheme among the parties, it follows that the parties can locally compute an additive sharing of β. At this point, we would want the parties to open the sharing of β and check equality to 0. However, an additive sharing has no robustness in it and so the parties have no way to verify that the received shares are correct. To overcome this, we first ask the parties to secret share their additive shares of $\psi = \sum_{k=1}^m \theta_k \cdot (x_k \cdot y_k)$ in a verifiable way. Denote by ψ^i the additive share of ψ held by party P_i. Once the parties hold $[\![\psi^i]\!]$ for each $i \in [n]$, the parties can compute $[\![\beta]\!] = \sum_{k=1}^m \theta_k \cdot [\![z_k]\!] - \sum_{i=1}^n [\![\psi^i]\!]$ and reconstruct the value of β. By the properties of the reconstruct procedure, the corrupted parties cannot do any harm beyond causing an abort. However, this is not enough since a corrupted party can share any value it wishes. Thus, the parties need to verify that each party shared the correct value. Towards achieving this, recall that one of the properties of the secret sharing scheme, is that it allows local conversion from $[\![x_k]\!], [\![y_k]\!]$ to $[\![x_k^i]\!], [\![y_k^i]\!]$ where x_k^i, y_k^i are the shares of x_k, y_k held by party P_i respectively. Thus, the parties wish to verify that

$$\forall i \in [n] : \sum_{k=1}^{m} \theta_k \cdot ([\![x_k^i]\!] \cdot [\![y_k^i]\!]) - [\![\psi^i]\!] = 0. \tag{3}$$

Letting $[\![c^i]\!] = [\![\psi^i]\!]$, $[\![a_k^i]\!] = \theta_k \cdot [\![x_k^i]\!]$ and $[\![b_k^i]\!] = [\![y_k^i]\!]$ we have that the parties ensure that $\forall i \in [n] : [\![c^i]\!] - \sum_{k=1}^m [\![a_k^i]\!] \cdot [\![b_k^i]\!] = 0$. This is exactly the type of statement that can be verified using $\mathcal{F}_{\text{proveDeg2Rel}}^{\text{abort}}$ defined in Section 3. Hence, the parties call $\mathcal{F}_{\text{proveDeg2Rel}}^{\text{abort}}$ and proceed only if it outputs accept. The formal description of the protocol and a security proof appear in the full version of the paper.

Extending the protocol to the ring $\mathbb{Z}_{2^k}$. If the parties work over the ring $\mathbb{Z}_{2^k}$, then the statistical error of the protocol is only 1/2. To achieve an error which is sufficiently small, the parties can choose $\theta_1, \ldots, \theta_m$ from a larger ring $\mathbb{Z}_{2^{k+s}}$. Then, the probability that $\beta = 0$ when $\exists k \in [m] : \; d_k = z_k - x_k \cdot y_k \neq 0$ will be at most 2^{-s}.

Communication Complexity. Note that in the protocol each party only shares one element and reconstructs one element. The cost of computing $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{abort}}$ thus equals to the cost of calling n copies of $\mathcal{F}_{\text{proveDeg2Rel}}^{\text{abort}}$ plus a small constant cost. By the analysis in Section 3.1, we conclude that the cost is $O(n \log m + n)$.

4.2 The Main Protocol

Our main protocol works in the $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{abort}}$-hybrid model. In the protocol, the parties first verifiably secret shares their inputs to the other parties. Then, they compute the program using Π_{mult}. Before revealing the outputs, they call $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{abort}}$ to verify the correctness of all multiplication triples. If the output received from $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{abort}}$ is reject, then they abort. Otherwise, they proceed to reconstruct the output. The formal description appears in the full version of the paper.

Communication complexity. Let m be the number of multiplication gates in the program and let $|\Pi_{\text{mult}}|$ be the communication cost per party when running Π_{mult}. Thus, the communication cost is $|\Pi_{\text{mult}}| \cdot m + O(\log m \cdot n)$. Amortized over the size of the program and assuming that $m >> n$, we have that the cost per gate is $|\Pi_{\text{mult}}|$.

Practical instantiations. Our protocol can be instantiated using both replicated and Shamir's secret sharing schemes (see Section 2.2). The former is usually used for small number of parties and when working over rings, whereas the latter is usually preferred when the number of parties grows, due to the fact that the size of each share grows at most logarithmically with n. For Π_{mult}, it is possible to use protocols such as [1, 28] (for 3 parties) or the DN protocol [11] for any number of parties. As shown in Section 2.3 (see also [4], the communication cost of the semi-honest DN protocol with replicated secret-sharing and pseudorandom secret sharing is less than 1.5 ring elements per party per multiplication. This dominates the amortized cost of our main protocol.

4.3 Concrete Efficiency

To illustrate the efficiency of our protocol, we measured the exact communication cost of our verification protocol, for various program sizes and number of parties. In Table 1, we present the number of *field elements* sent per party amortized over the size of the program, when instantiating our protocol with Shamir's secret sharing scheme. The reported numbers in the table can be seen as the cost of strengthening security from semi-honest to malicious, per multiplication instruction. As can be seen, the communication overhead of our verification protocol is so low, that even when the number of parties is increased to 1000, the cost is still less just 0.76 field element per instruction. We note that when the field is small, the verification protocol can be "lifted" to an extension field $\mathbb{F}$ of the same characteristic, without changing the base semi-honest protocol. As a result, the statistical error can be reduced to (roughly) an inverse of the size of the extension field.

In Table 2 we present the communication cost when our protocol is used to compute a program defined over the ring $\mathbb{Z}_{2^k}$ for some $k \geq 1$ (when $k = 1$ this is equivalent to computing a binary circuit), with replicated secret sharing as

# of Multiplication Triples (m)	Field Elements per Party per Triple			
	$n = 25$	$n = 50$	$n = 500$	$n = 1000$
2^{15}	0.02	0.03	0.38	0.76
2^{20}	0.0007	0.001	0.01	0.02
2^{25}	0.00002	0.00005	0.0005	0.001
2^{30}	0.0000009	0.000002	0.00002	0.0003

Table 1. Field elements sent per party in the verification of m multiplication triples, per one triple, when Shamir's secret sharing is used, for different sizes of m and number of parties n. The numbers are computed via the formula $(10n + n \cdot \log m) \cdot \frac{1}{m}$ and the statistical error is $\frac{2\log m+4}{|\mathbb{F}|-5}$.

# of Multiplication Triples (m)	Ring Elements sent per Party per Triple				
	$n = 3$	$n = 5$	$n = 7$	$n = 9$	$n = 11$
2^{15}	0.002	0.13	0.22	0.41	0.97
2^{20}	0.00008	0.005	0.008	0.01	0.03
2^{25}	0.000003	0.0002	0.0003	0.0005	0.001
2^{30}	0.0000001	0.000007	0.00001	0.00001	0.00005

Table 2. Ring elements sent per party in the verification of m multiplication triples, per one triple, for different sizes of m and number of parties n, when the semi-honest computation is over the ring $\mathbb{Z}_{2^k}$ and using replicated secret sharing scheme. The numbers are computed via the formula $\left(\binom{n-1}{t} \cdot 2 + 2.5n + n\log(m)\right) \cdot \frac{1}{m} \cdot d$, where the extension degree d satisfies the condition $d > 40 + \log(2\log m + 4)$ to achieve statistical error of 2^{-40}.

the underlying secret sharing scheme. Recall that in this case, the verification protocol is carried-out over an extension ring (see the end of Section 3.1). To compute the number of ring elements sent in the verification protocol, we thus multiply the communication cost obtained over fields with the degree extension d (since the size of each element is increased by a factor of d). The extension degree depends on the desired statistical error, which is approximately $\frac{2\log m+4}{2^d}$. This means in particular that for security of s bits, the extension degree should satisfy the condition $d > s + \log(2\log m + 4)$. In Table 2, we report the number of sent ring elements per instruction for each party, with statistical error of at most 2^{-40}, and so it suffices to set $d = 46$. In addition, each opening of a secret requires each party to send $\binom{n-1}{t}$ elements. However, note that this is not the case for sharing a secret, since here we can have all subsets except one derive their share from a pre-distributed seed (known also to the dealer), and have the dealer send just one share (to adjust the secret) to one subset of $t+1$ shares. This means that sharing a secret yields cost of 0.5 ring elements per party, exactly as for Shamir's secret sharing. Due to the fast increase of the share's size in this scheme, we report the cost up to 11 parties. Note that even for $n = 11$, programs of size $\geq 2^{15}$) can be computed in the presence of malicious adversaries, while paying an extra cost of *less than 1 ring elements per instruction* beyond the cost of semi-honest security.

For the computational cost, we saw that in $\mathcal{F}^{\text{abort}}_{\text{proveDeg2Rel}}$ the number of local operations is $O(m)$ with small constants. Observe that in $\mathcal{F}^{\mathsf{abort}}_{\mathsf{vrfy}}$ the parties only

	Communication per party (field elements)	**# of rounds**
Nordholt et al. [32]	$O(m+n)$	$O(1)$
Boneh et al. [4]	$O(n\sqrt{m}+n)$	$O(1)$
Goyal et al. [22]	$O(n\log m+n)$	$O(\log m)$
This work (with Fiat-Shamir)	$\mathbf{O(n\log m+n)}$	$\mathbf{O(1)}$

Table 3. Comparison to previous works of communication and round complexity, when verifying m multiplication triples by n parties.

need to compute a linear combination of m inputs and so the cost is roughly m operations. Since we have n calls to $\mathcal{F}^{\text{abort}}_{\text{proveDeg2Rel}}$, the overall cost is $n \cdot O(m)$.

Comparison to previous works. In Table 3 we compare our security-with-abort verification protocol with previous works. As can be seen, our work as well as [4, 22] achieve sublinear communication, whereas [32] achieves only linear communication in the amount of verified triples m. Our improvement compared to [4] is that our sublinear additive term is logarithmic in m rather than just square root of m. Compared to [22], we are able to use the Fiat-Shamir transform to achieve constant number of rounds (see Section 3.1), whereas in their protocol, the parties carry out a joint multiparty computation is each step of the protocol, and so it is unclear how to reduce interaction via the Fiat-Shamir transform.

5 Achieving Full Security for Constant Number of Parties

In this section, we show how to augment our base construction to full security, including fairness and guaranteed output delivery, without changing the amortized communication cost.

Our protocol works by having the parties divide the program into segments and compute each segment separately. For each segment, the parties work in the same way as before, that is, computing it first using a private multiplication protocol and then running a verification protocol. However, we change the verification protocol so that it will give the parties more information besides outputting merely accept or reject. Specifically, in the case of reject, the verification protocol will also output a pair of parties in conflict, such that at least one of them is guaranteed to be corrupted. Once such a pair is known, the parties will remove both parties from the protocol and recompute the segment without them. Since one of the eliminated parties is corrupt, it follows that an honest majority is maintained even though the number of parties was reduced by two. Removing two parties and restarting the segment computation without them raises several challenges. In particular, the parties need to carefully move from a t-out-of-n sharing to a $(t-1)$-out-of-$(n-2)$ secret sharing. Our solution to this includes having authentication tags over the shares, which prevent corrupted parties from cheating in the process. We present a novel technique for computing these tags efficiently, requiring a single tag for all the shares held by a subset of $t+1$ parties and using sublinear communication in the number of shares. We stress that authentication is required only for the *secrets that are stored in memory when moving from one layer to the next layer*. This fact together with

the sublinear communication of our verification protocol implies that the overall amortized communication cost per multiplication instruction remains $|\Pi_{\text{mult}}|$.

The construction in this section is designed for replicated secret sharing scheme only and thus we assume that the number of parties n is *constant*. Our construction depends on two properties that hold for replicated secret sharing: (1) Pair-wise consistency: when opening a secret, the opening will fail if there exist two parties which do not agree on a certain share. If we know in advance that the sharing was consistent, such a disagreement can occur only with a corrupted party. This is used in our protocol to find a pair of disputed parties, where at least one of them is guaranteed to be corrupt. (2) For each input held by a party P_i, we can define a consistent secret sharing of this input, which is known to P_i. This holds since any secret held by P_i is known to t other parties and so it is possible to define a sharing where the share of one subset of $t+1$ parties is the input itself, whereas the shares of the other subsets is 0. This property is required in our verification protocol when each party proves it behaved honestly when sharing a secret.

This section is organized as follows. In Section 5.1 we present the updated verification protocol which allows identification of a pair of conflicting parties to eliminate. In Section 5.2 we present two additional sub-protocols which are required for our construction. Finally, in Section 5.3 we present the main protocol for computing any arithmetic program.

5.1 Joint Verification of Multiplications with Cheating Identification

In this section, we present the verification protocol, with the property that when cheating took place in the execution of the private multiplication protocol, the parties will be able to identify a pair of conflicting parties (and not just reject the computation). Our protocol realizes the functionality $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{full}}$ formally described in Functionality 5.1, which is defined similarly to $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{abort}}$ but with two differences: first, the parties always receive the same output. Second, if the trusted party computing $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{full}}$ outputs reject (which means that there exists an incorrect multiplication triple), then the ideal world adversary can pick one of two options: provide a pair of parties to eliminate, where at least one of them is a corrupted party, or let $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{full}}$ detect such pair. In the latter, $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{full}}$ receives the inputs, randomness and views of the honest parties when computing some incorrect multiplication triple. Then, based on this information, $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{full}}$ finds a pair of conflicting parties and outputs it to the parties.

Our protocol to compute $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{full}}$ is an extension of our protocol from Section 4.1. In order to add the cheating identification property to our verification protocol, we need to provide a mechanism to identify a pair of conflicting parties in each step for which the parties may output reject in the original protocol. There are 4 such steps: (i) when the VSS protocol to share the additive shares fails due to inconsistency; (ii) when $\mathcal{F}_{\text{proveDeg2Rel}}^{\text{abort}}$ returns reject; (iii) when the opening of β fails due to inconsistency; and (iv) when the parties output reject since $\beta \neq 0$.

Note that in (i), we can simply ask the dealer to broadcast any share for which pair-wise inconsistency exist. Since this can happen only with shares that are known to the adversary, no secret information is never revealed. To identify

FUNCTIONALITY 5.1 ($\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{full}}$- Verify Mult. with Cheating Identification)
Let $\mathcal{S}$ be the ideal world adversary controlling a subset $< n/2$ of corrupted parties. The functionality $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{full}}$ is invoked by the honest parties sending their shares of m multiplication triples $\{(x_k, y_k, z_k)_{k=1}^m\}$ to $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{full}}$.
Then, $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{full}}$ computes all secrets and the corrupted parties' shares. These shares are sent to $\mathcal{S}$.
Then, it checks that $z_k = x_k \cdot y_k$ for all $k \in [m]$. If this holds, it sends accept to $\mathcal{S}$. Otherwise, it sends reject to $\mathcal{S}$ and $d_k = z_k - x_k \cdot y_k$ for each $k \in [m]$ such that $d_k \neq 0$. Then:

- If $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{full}}$ sent accept, then it waits for $\mathcal{S}$ to send $\mathsf{out} \in \{\mathsf{accept}, \mathsf{reject}\}$ which is then handed to the honest parties. If $\mathsf{out} = \mathsf{reject}$, then $\mathcal{S}$ is required to send a pair of indices (i, j) to $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{full}}$ with at least one of them being a corrupted party. Then, $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{full}}$ hands (i, j) to the honest parties.
- If $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{full}}$ sent reject, then $\mathcal{S}$ chooses one of the next two options:
  - Send a pair of indices (i, j) to $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{full}}$ with at least one of them being a corrupted party. Then, $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{full}}$ hands (i, j) to the honest parties.
  - Ask $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{full}}$ to find a pair of conflicting parties in the $\bar{k}$th multiplication. Then, $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{full}}$ commands the honest parties to send their inputs, randomness and views in the execution to compute $\bar{k}$th triple. Then, based on this information, $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{full}}$ computes the messages that should have been sent by each corrupted party, and find a pair of parties P_i, P_j , where P_j received an incorrect message. Then, $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{full}}$ sends (i, j) to the honest parties and $\mathcal{S}$.

a pair of conflicting parties in case (iii), we use the pairwise-consistency check of replicated secret sharing to identify a disputed pair. Namely, that inconsistency can occur only when an honest party and a corrupted party disagree on the value of a share held by both of them. Note that in addition we need that the messages in the consistency check will broadcast (via $\mathcal{F}_{\mathsf{bc}}$), otherwise the parties may not agree on the disputed pair they output. For (ii), we simply use $\mathcal{F}_{\text{proveDeg2Rel}}^{\text{cheatIdntfy}}$. Recall that our protocol to realize $\mathcal{F}_{\text{proveDeg2Rel}}^{\text{cheatIdntfy}}$ requires that the proving party will know the shares held by the other parties. This indeed holds for replicated secret sharing, since the parties convert $[\![x_k]\!], [\![y_k]\!], [\![z_k]\!]$ to $[\![x_k^i]\!], [\![y_k^i]\!], [\![z_k^i]\!]$ by setting the shares of all subsets T for which $P_i \notin T$ to be 0 (see Section 2.2). Finally, for case (iv), if the parties reject since $\beta \neq 0$, we observe that this means that no one cheated in the verification protocol itself (with high probability). Thus, the parties can conclude that cheating took place in one of the calls to the private multiplication protocol to compute the program. The parties thus continue to localize the fault by running a binary search on the set of multiplication triples, aiming to find the *first* triple k where the corrupted party have cheated and $z_k \neq x_k \cdot y_k$. In each step of the search, the parties repeat the verification protocol on a smaller set of triples. The search will continue at the worst case (i.e., if no execution have ended with obtaining a pair of conflicting parties), until the parties are left with one incorrect triple. Finally, the parties can check the execution of the multiplication protocol for computing this triple,

and use it to find a pair of disputing parties. For this final check, we define an ideal functionality $\mathcal{F}_{\mathsf{miniMPC}}$ that receives the input, randomness and view of each honest party in the multiplication protocol and output the first pair of parties for which incoming and sent messages do not match. Observe that this functionality is called just once for the entire computation and so its cost is amortized away, regardless of the way it is realized. We provide a formal description of the protocol and proof of security in the full version of the paper.

Cheating probability. Assume that there is one incorrect triple. Then, if the adversary does not cheat in the verification protocol, then this triple will be tested in at most $\log m$ executions of the protocol. In each execution, the probability that it will pass the test is bounded by $\frac{1}{|\mathbb{F}|}$. This holds since the parties will output accept in this case only if the random linear combination cause the opened value to be 0. Note that if the output of the parties is accept when examining a set of triples, then they stop the search in this set. Thus, an incorrect triple has $\log m$ attempts to be accepted. The overall cheating probability is therefore bounded by $\log m \cdot \frac{1}{|\mathbb{F}|}$.

We remark that the protocol can be extended to work over a ring in the same way as for $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{abort}}$. See the remark at the end of Section 4.1.

Communication cost. Our protocol is recursive. In jth step of the recursion, the parties secret share one element, reconstruct one element (using $\mathcal{F}_{\mathsf{bc}}$) and call $\mathcal{F}_{\text{proveDeg2Rel}}^{\text{cheatIdntfy}}$ for each party over a set of triples of size $m/2^j$. Sharing a secret requires each party to send $\binom{n}{t}$ elements (we ignore here the consistency check which can be typically done with constant cost), reconstruction requires sending $\binom{n-1}{t}$ elements by each party and the cost of n invocations of $\mathcal{F}_{\text{proveDeg2Rel}}^{\text{cheatIdntfy}}$, as shown in Section 3.2, is $n\log(m/2^j) + 4n \cdot \binom{n-1}{t} \cdot |\mathcal{F}_{\mathsf{bc}}|$ per party. Overall, the obtained cost per party is roughly

$$\binom{n}{t} \cdot \log m + \binom{n-1}{t} \cdot \log m \cdot 4n \cdot |\mathcal{F}_{\mathsf{bc}}| + n \cdot \log m \cdot \log \sqrt{m} \ \ \mathsf{ring\ elements}. \quad (4)$$

For constant number of parties, the asymptotic cost is roughly $O\left(\log m \cdot \log \sqrt{m}\right))$, which is sublinear in m. We remind the reader that when the triples were computed over the ring $\mathbb{Z}_{2^k}$, then the verification protocol is carried-out over an extension ring; see the end of Section 4.2 for more details.

5.2 Two Additional Building Blocks

Computing Authentication Tags We next show how to compute an authentication tag over shares held by a subset T of $t+1$ parties. Let $x_1^T, \ldots, x_L^T$ be the shares held by the parties in T. The authentication tag τ^T is computed as follows: $\tau^T = \sum_{k=1}^{L} u_k^T \cdot x_k^T + v^T$, where $\boldsymbol{u}^T = (u_1^T, \ldots, u_L^T)$ and v^T are random secret keys that are shared among the parties using *authenticated* secret sharing (see definition in Section 2). Observe that for the long vector u^T it is possible to secret share a random seed from which the key is expanded, thus using the expensive mechanism of authenticated secret sharing only small constant number of times.

To compute the tag we observe that the parties can first locally compute an additive sharing of $\sum_{k=1}^{m} u_k^T \cdot x_k^T$. This is done by taking $[\![u_k^T]\!] \cdot [\![x_k^T]\!]$, where $[\![x_k^T]\!]$

is simply defined such that the share held by subset T is x_k^T and the shares held by the other subsets is 0. Then, we let each party secret share each additive share and prove that it shared the correct secret. The observation here is that we can utilize the functionality $\mathcal{F}_{\text{proveDeg2Rel}}^{\text{cheatIdntfy}}$ for this proof, as the additive share each party computes and shares to the other parties, is a 2-degree function of inputs that are verifiably shared among the other parties. If all proofs passed the check, then the parties can locally add the shared secrets, add $[\![v^T]\!]$ to the result and reconstruct the obtained tag. If the reconstructions fails due to pair-wise inconsistency, then the parties obtain a conflicting pair of parties.

Formally, The parties work as follows:

$\Pi_{auth}(x_1^T, \ldots, x_L^T, [\![\boldsymbol{u}^T]\!], [\![v^T]\!])$:

1. The parties locally compute $\langle z^T \rangle = \left\langle \sum_{k=1}^{L} u_k^T \cdot x_k^T \right\rangle = \sum_{k=1}^{L} [\![u_k^T]\!] \cdot [\![x_k^T]\!]$
2. Let $z^{T,i}$ the additive share of z^T held by P_i. Note that by definition $z^{T,i} = 0$ for each $P_i \notin T$. Then, each party $P_i \in T$ verifiably secret shares (VSS) $z^{T,i}$ to the other parties.
3. For each $i \in [n]$ such that $P_i \in T$, the parties convert $[\![u_k^T]\!]$ to $[\![u_k^{T,i}]\!]$ for each $k \in [L]$ and send $[\![z^{T,i}]\!]$ and $\left([\![u_k^T]\!], [\![x_k^T]\!]\right)_{k=1}^{L}$ to $\mathcal{F}_{\text{proveDeg2Rel}}^{\text{cheatIdntfy}}$.
4. If the parties received $\mathsf{reject}, (i, j)$ from $\mathcal{F}_{\text{proveDeg2Rel}}^{\text{cheatIdntfy}}$ in any of the calls in the previous step, then the parties output the first pair of conflicting parties (P_i, P_j). Otherwise, they proceed to the next step.
5. The parties locally compute $[\![\tau^T]\!] = \sum_{i \mid P_i \in T} [\![z^{T,i}]\!] + [\![v^T]\!]$.
6. The parties reveal τ^T by sending their shares via $\mathcal{F}_{\mathsf{bc}}$ to each other. If the shares are inconsistent, then the parties output the first pair of parties for which pair-wise consistency exists. Otherwise, they output τ^T.

Communication Complexity. We note that in practice the parties can call $\mathcal{F}_{\text{proveDeg2Rel}}^{\text{cheatIdntfy}}$ once per party for all shares (over the same layer of instructions). Thus, the cost is dominated by each party secret sharing its additive sharing of z^T, and opening the shared tag at the end. Overall, this means that for each subset T of $t+1$ parties, the cost per party is $\binom{n}{t} + |\mathcal{F}_{\text{proveDeg2Rel}}^{\text{cheatIdntfy}}| + \binom{n-1}{t} \cdot |\mathcal{F}_{\mathsf{bc}}|$.

Player Elimination and Recovery We next show how the parties can remove a pair of conflicting parties and restart the computation without them.

Denote the parties to eliminate by P_i and P_j. The goal is to recompute the segment, but with less parties. Since we are guaranteed that at least one of the parties is corrupted, then we move from a t-out-of-n secret sharing to a $(t-1)$-out-of-$(n-2)$ secret sharing (i.e., the number of parties is reduced by 2 and the threshold is reduced by 1). In order to achieve this, we distinguish between three types of shares:

- *Shares that are known to either P_i or P_j*: In this case, no action is needed by the parties, as each such share is now known to t active parties, which is exactly what needed by the updated threshold.
- *Shares that are known to both P_i and P_j*: Shares in this category are held by a subset T of $t+1$ parties, with $P_i, P_j \in T$. Since we require that from now on each share will be held by a subset of t parties, it suffices to reveal

this share to a subset T' of t parties, which will add the share to its current share. To minimize communication, we can take $T' = T \setminus \{P_i, P_j\} \cup \{P_k\}$ for some $P_k \notin T$. This implies that we need all parties in T to send the share to P_k.This is where the authentication tags are being used. Each party that holds the share sends it to P_k. However, corrupted parties may send incorrect values. Thus, the keys used to authenticate the share are also being revealed (recall that they are secret shared using an authentication secret sharing scheme and so cheating is not possible when opening these values). Once the keys are revealed, party P_k checks for each share it received, that the tag is correct given the authentication keys (i.e., that $\tau^T = \sum_{k=1}^{L} u_k^T \cdot x_k^T + v^T$). Since in each subset there exists at least one honest party, then at least one of the possible shares is correct, and so the check will pass for this share.

– *Shares that are not known to both P_i and P_j*: Note that each such share is known to a set of $t+1$ active parties. Since the threshold is now reduced to t, we just let one subset of t parties (there are exactly $\binom{t+1}{t} = t+1$ such subsets) locally add this share to the share already held by it. Note that the parties can locally update the authentication tag for the updated share of this subset, by simply adding the tag of the added share to the existing tag.

Observe that only for shares in the second category interaction is required. There are $\binom{n-2}{t-1}$ such shares, which are transmitted from $t+1$ parties to a single party. Recall that this cost is paid only for shares that are stored between segments of the program. Nevertheless, later we will see that for specific instantiations, it is possible to eliminate this cost completely.

5.3 The Main Protocol

In this section, we describe our main protocol to compute any straight-line program. Our protocol computes the program segment by segment. Throughout the protocol we maintain the following invariant: at the beginning of each segment's computation, the parties hold a consistent sharing of the values on the input layer of the segment, an authentication tag for the shares held by each subset of $t+1$ parties on the input layer and an authenticated secret sharing of the keys used to compute the tag. A computation of a segment includes using private multiplication and computing authentication tags for the shares on the output layer of the segment. Then, the parties use the verification protocol to verify that the output is correct. If the verification succeeds, then the parties can proceed to the next segment. Otherwise, the parties hold a pair of parties to eliminate. In this case, they apply the player elimination and recovery subprotocol and recompute the segment with less two parties and updated secret sharing of the input layer. To achieve fairness when outputs are revealed we use again the authentication mechanism. Here however, we cannot authenticate all shares held by a subset T together, since the shares may be intended to different parties. Thus, for the output layer of the entire program, the parties compute new authentication tags for each subset of shares intended to party P_i and held by a subset of parties T. The formal description can be found in the full version of this paper.

Size of the segments. Each time we repeat the computation of a segment, it means that one corrupted party was eliminated. Thus, each segment can be

computed at most t times. If we split the program to $O(n^2)$ equally sized segments (i.e., with the same amount of multiplication instructions), then amortized over the entire program, it can be shown that the average number of repetitions per instruction is approximately 1.

Communication Complexity for constant number of parties. For each segment with $m/O(n^2)$ multiplication instructions, we call Π_{mult} for each multiplication, call Π_{auth} for each subset of $t+1$ parties at the output layer and call $\mathcal{F}_{\text{vrfy}}^{\text{full}}$ once. The asymptotic cost of $\mathcal{F}_{\text{vrfy}}^{\text{full}}$ per party for a segment of size m/n^2 is $O(\log(m/n^2) \cdot \log \sqrt{m/n^2})$. Thus, the cost of computing the segment is $\frac{m}{n^2} \cdot |\Pi_{\text{mult}}| + O(\log(m/n^2) \cdot \log \sqrt{m/n^2})$. Summing over all $O(n^2)$ segments, the cost per party is thus $m \cdot |\Pi_{\text{mult}}| + O(n^2 \log(m/n^2) \cdot \log \sqrt{m/n^2})$. Letting the program's size S be its number of multiplication instructions, and assuming that n is constant, the cost of our protocol per multiplication per party is $|\Pi_{\text{mult}}| + o(S)$.

If cheating took place, then the parties need to recover shares held by the eliminated parties for each secret stored in memory between the segments. The number of such secrets is bounded by the width of the program W. Thus, in case of cheating the cost per party is $|\Pi_{\text{mult}}| \cdot S + O(W) + o(S)$. Note that $W \leq S$ and in many cases, W will be much smaller than S, and so $O(W)$ can be ignored.

Removing the $O(W)$ term when Π_{mult} is Instantiated with [11] If we instantiate Π_{mult} with the DN protocol [11], then as explained in Section 2.3, the cost of Π_{mult} is *1.5 elements per party*. We next show how it is possible to recover from cheating without increasing the communication cost, improving upon our general construction from Section 5.2. Recall that in the DN protocol, the output shares (of each multiplication) are computed by taking $[\![r]\!] + (xy - r)$, where $[\![r]\!]$ is a sharing of a random r that was generated in the offline step (possibly without any interaction), and $xy - r$ is computed by party P_1 (the parties send him masked additive shares of $x \cdot y$). Note that $xy - r$ is in fact sent from P_1 only to one subset of $t+1$ parties (including P_1 itself), denoted by T. Now, assume that cheating was detected and two parties, say P_i and P_j are eliminated. To recover the computation, it suffices that the parties will generate a new $[\![r]\!]$ with the updated $t-1$ threshold, and that one subset of t active parties will add $xy - r$ to its share of r. If the eliminated parties are not both in T, then this can be done without interaction. However, if both of them are in T, then $xy - r$ is known now only to $t-1$ active parties. Thus, we require that some party $P_k \notin T$ will learn $xy - r$. To this end, we ask party $P_\ell \in T$ ($\ell \neq i, j$) to send $xy - r$ to P_k. To detect whether P_ℓ sent the correct value, we use the authentication mechanism as before. Specifically, the parties compute authentication tags for all $xy - r$ received during the computation (for secrets that are outputs of segments only). Thus, if the authentication succeeds, then P_k has the correct $xy - r$ and the parties can recompute the segment. Otherwise, P_k accuse P_ℓ of sending him an incorrect value. Note that in this case, we know again that either P_k or P_ℓ are corrupted. Moreover, this is a new pair of conflicted parties that does not overlap with the original pair. In this case, we restart the recovery process to remove 4 parties and update the sharings to a $(t-2)$-out-of-$(n-4)$ secret

sharing. As before, we ask a party from T to send $xy - r$ to a party outside of T, with both parties not being one of the eliminated parties, and so on. Note that the process can end with two outcomes: (1) At some point, no one complains. In this case, the parties successfully removed $t' < t$ pair of parties, where in each pair, one of the parties is guaranteed to be corrupted. The parties thus can continue the computation. (2) The parties keep adding pair of conflicted parties to the list, until we are left with one honest party. This holds since we started with $t - 1$ active parties in T, and t outside of T. Thus, at some point there will remain one party outside of T. This party must be honest since we overall eliminated t pairs of semi-corrupted parties, with the property that one of them must be corrupted. Since there are t corrupted parties, the remaining party is honest. In this case, following the 3-party construction of [5], this honest party can be used as a trusted party and complete the computation. Note that in the above process, each pair that is eliminated requires the transmission of one element. However, in future multiplications, the overall communication is reduced by at least one element, since a party that is eliminated, will not be part of the interaction anymore. Thus, amortized over the circuit, the recovery process is communication-free. The overall cost of our entire protocol is thus $1.5 \cdot S + o(S)$, with no dependency on the width of the circuit.

References

1. Araki, T., Furukawa, J., Lindell, Y., Nof, A., Ohara, K.: High-throughput semi-honest secure three-party computation with an honest majority. In: ACM CCS (2016)
2. Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In: ACM Symposium on Theory of Computing (1988)
3. Ben-Sasson, E., Fehr, S., Ostrovsky, R.: Near-linear unconditionally-secure multi-party computation with a dishonest minority. In: CRYPTO (2012)
4. Boneh, D., Boyle, E., Corrigan-Gibbs, H., Gilboa, N., Ishai, Y.: Zero-knowledge proofs on secret-shared data via fully linear pcps. In: CRYPTO (2019)
5. Boyle, E., Gilboa, N., Ishai, Y., Nof, A.: Practical fully secure three-party computation via sublinear distributed zero-knowledge proofs. In: ACM CCS (2019)
6. Byali, M., Hazay, C., Patra, A., Singla, S.: Fast actively secure five-party computation with security beyond abort. In: ACM CCS (2019)
7. Chaum, D., Crépeau, C., Damgård, I.: Multiparty unconditionally secure protocols (extended abstract). In: ACM Symposium on Theory of Computing (1988)
8. Chida, K., Genkin, D., Hamada, K., Ikarashi, D., Kikuchi, R., Lindell, Y., Nof, A.: Fast large-scale honest-majority MPC for malicious adversaries. In: CRYPTO (2018)
9. Cleve, R.: Towards optimal simulations of formulas by bounded-width programs. In: ACM Symposium on Theory of Computing (1990)
10. Cramer, R., Damgård, I., Ishai, Y.: Share conversion, pseudorandom secret-sharing and applications to secure computation. In: TCC (2005)
11. Damgård, I., Nielsen, J.B.: Scalable and unconditionally secure multiparty computation. In: CRYPTO (2007)
12. Dolev, D., Strong, H.R.: Authenticated algorithms for byzantine agreement. SIAM J. Comput. **12**(4), 656–666 (1983)

13. Fehr, S., Yuan, C.: Towards optimal robust secret sharing with security against a rushing adversary. In: EUROCRYPT (2019)
14. Fiat, A., Shamir, A.: How to prove yourself: Practical solutions to identification and signature problems. In: CRYPTO (1986)
15. Furukawa, J., Lindell, Y.: Two-thirds honest-majority MPC for malicious adversaries at almost the cost of semi-honest. In: ACM CCS (2019)
16. Genkin, D., Ishai, Y., Prabhakaran, M., Sahai, A., Tromer, E.: Circuits resilient to additive attacks with applications to secure computation. In: STOC (2014)
17. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: ACM symposium on Theory of computing (2009)
18. Gilboa, N., Ishai, Y.: Compressing cryptographic resources. In: CRYPTO (1999)
19. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or A completeness theorem for protocols with honest majority. In: ACM Symposium on Theory of Computing (1987)
20. Gordon, S.D., Ranellucci, S., Wang, X.: Secure computation with low communication from cross-checking. In: ASIACRYPT (2018)
21. Goyal, V., Liu, Y., Song, Y.: Communication-efficient unconditional MPC with guaranteed output delivery. In: CRYPTO (2019)
22. Goyal, V., Song, Y.: Malicious security comes free in honest-majority MPC. IACR Cryptol. ePrint Arch. (2020)
23. Goyal, V., Song, Y., Zhu, C.: Guaranteed output delivery comes free in honest majority MPC. In: CRYPTO (2020)
24. Hirt, M., Maurer, U.M., Przydatek, B.: Efficient secure multi-party computation. In: ASIACRYPT (2000)
25. Ishai, Y., Kumaresan, R., Kushilevitz, E., Paskin-Cherniavsky, A.: Secure computation with minimal interaction, revisited. In: CRYPTO (2015)
26. Ishai, Y., Kushilevitz, E., Prabhakaran, M., Sahai, A., Yu, C.: Secure protocol transformations. In: CRYPTO (2016)
27. Ito, M., Saito, A., Nishizeki, T.: Secret sharing scheme realizing general access structure. Electronics and Communications in Japan (1989)
28. Katz, J., Kolesnikov, V., Wang, X.: Improved non-interactive zero knowledge with applications to post-quantum signatures. In: ACM CCS (2018)
29. Lindell, Y., Nof, A.: A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In: ACM CCS (2017)
30. Maurer, U.M.: Secure multi-party computation made simple. Discrete Applied Mathematics (2006)
31. Mohassel, P., Rosulek, M., Zhang, Y.: Fast and secure three-party computation: The garbled circuit approach. In: ACM CCS (2015)
32. Nordholt, P.S., Veeningen, M.: Minimising communication in honest-majority MPC by batchwise multiplication verification. In: ACNS (2018)
33. Patra, A., Ravi, D.: On the exact round complexity of secure three-party computation. In: CRYPTO (2018)
34. Pease, M.C., Shostak, R.E., Lamport, L.: Reaching agreement in the presence of faults. J. ACM **27**(2), 228–234 (1980)
35. Rabin, T., Ben-Or, M.: Verifiable secret sharing and multiparty protocols with honest majority. In: ACM Symposium on Theory of Computing (1989)
36. Rivest, R.L., Adleman, L., Dertouzos, M.L., et al.: On data banks and privacy homomorphisms. Foundations of secure computation (1978)
37. Shamir, A.: How to share a secret. Commun. ACM **22**(11), 612–613 (1979)
38. Yao, A.C.: How to generate and exchange secrets (extended abstract). In: Symposium on Foundations of Computer Science (1986)