Efficient Fully Secure Computation via Distributed Zero-Knowledge Proofs*

Elette Boyle† Niv Gilboa‡ Yuval Ishai § Ariel Nof ¶

November 17, 2020

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 sublinear-communication distributed zero-knowledge proofs of Boneh *et al.* (Crypto 2019) 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 fully secure protocol builds on a new intermediate honest-majority protocol for verifying the correctness of multiplication triples by making a *general* use of distributed zero-knowledge proofs. While this intermediate 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 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 [Yao86, GMW87, BGW88, CCD88] enable a set of parties with private inputs to compute a joint function of their inputs while revealing nothing but the output.

*This is a full version of [BGIN20].

†IDC Herzliya, Israel. email: eboyle@alum.mit.edu.

‡Ben-Gurion University, Israel email: gilboan@bgu.ac.il.

§Technion, Israel. email: yuvali@cs.technion.ac.il.

¶Technion, Israel. email: ariel.nof@cs.technion.ac.il.

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 [RB89] (where the latter can be realized from a public-key infrastructure using digital signatures [DS83]), or alternatively using only secure channels assuming $t < n/3$ [BGW88, CCD88]. However, despite extensive 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).[1]

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 [IKP+16, GLS19, BHPS19, BGIN19, GSZ20] 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. [BGIN19] in the special case of 3 parties (i.e., $n = 3, t = 1$), and Goyal et al. [GSZ20] that approaches the goal for general n.

For the special case of 3 parties, the fully secure protocol of Boyle et al. [BGIN19] matches the amortized cost of the best known semi-honest protocol in this setting (due to Araki et al. [AFL+16]). More specifically, the protocol from [BGIN19] 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.[2] 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. [GSZ20] 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 a black-box use of a PRG instead of insisting on information-theoretic security, the semi-honest baseline can be improved. Finally, the protocol of [GSZ20] only applies to the case

[1] As opposed to expensive cryptographic tools such as fully homomorphic encryption [RAD+78, Gen09], 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.

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

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[3] n. We further settle for computational security, but restrict parties to make a black-box access to a PRG. Even in this setting, designing fully secure protocols is a challenging task. Indeed, concretely efficient protocols for $n = 3, 4$ or 5 parties, of which only $t = 1$ may be corrupted, have been the target of several previous works (e.g., [IKKP15, MRZ15, PR18, GRW18, BHPS19, BGIN19]). 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 security thresholds t.

For a constant threshold t and $n = 2t + 1$, the relevant semi-honest baseline is a protocol from [BBC+19] that optimizes a protocol of Damgård and Nielsen [DN07] using pseudorandom secret sharing [GI99, CDI05]. This protocol applies to an arithmetic circuit over a ring R, with amortized communication cost of $\frac{3t}{2t+1}$ (< 1.5) R-elements per party per multiplication gate. This sets our target communication goal while achieving full security.

Relaxing full security to *security with abort*, this goal was recently met by Boneh et al. [BBC+19]. 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 [FL19] and then for $n = 2t + 1$ parties by Goyal and Song [GS20]. (A similar result, with a bigger sublinear additive term, can be obtained from the technique of [BBC+19].) 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* (or logarithmic) number of parties $n = 2t + 1$ that achieves full security against up to t malicious parties, with the *same amortized communication* as the best known semi-honest protocol mentioned above. In particular, each party communicates less than 1.5 ring elements per multiplication. Our protocol applies to both Boolean and arithmetic circuits, and even over the rings $\mathbb{Z}_{2^k}$. It uses secure point-to-point channels and a broadcast channel $\mathcal{F}_{\mathsf{bc}}$ (necessary to achieve full security in this setting, where broadcast is not possible without setup [PSL80]), and makes only black-box use of a PRG. The total size of strings communicated over $\mathcal{F}_{\mathsf{bc}}$ is small (in particular, sublinear in the circuit size), and so implementing $\mathcal{F}_{\mathsf{bc}}$ using digital signatures [DS83] would give the same amortized cost over point-to-point channels alone.

Our protocol relies on *replicated secret sharing* [ISN89], which enables an efficient use of pseudorandom secret sharing techniques [CDI05] for reducing the amortized cost of generating random shared secrets. The price one pays is a one-time setup whose communication cost scales exponentially with n as well as (per gate) *local computation* that scales exponentially with n. The latter

[3] More generally, our main protocol incurs computation and storage costs that scale exponentially with n, and are polynomial in the security parameter whenever n is at most logarithmic. However, these costs involve only symmetric cryptography and can be shifted almost entirely to an offline phase, before the inputs are known.

computational overhead is reasonable in practice for a small number of parties, especially given the speed of hardware-supported symmetric cryptography. Asymptotically, it makes the complexity of our protocol polynomial (in the security parameter) as long as the number of parties is at most logarithmic.

A basic building block in our fully secure protocol is an arbitrary n-party protocol Π_{mult} for *private multiplication* based on replicated secret-sharing. 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. Note that this does not impose any *correctness* requirement in the presence of malicious parties. The usefulness of replicated secret sharing for simplifying general secure computation protocols was first pointed out by Maurer [Mau06]. The most communication-efficient instance of a protocol of this type was given by Boneh et al. [BBC+19], combining the approach of Damgård and Nielsen [DN07] with the pseudorandom secret sharing technique of Cramer et al. [CDI05] (see also [GI99]).

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, and is typically sublinear in the circuit size. The general formulation of this result, relying on *any* private multiplication protocol, can be useful when considering other settings or networks (e.g., a "bulletin board" model where broadcast has unit cost). Adapting our general protocol to these alternative settings only requires a redesign of the optimized private multiplication sub-protocol.

At a very high level, the fully secure 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 [BBC+19], 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. [HMP00]) with an authentication mechanism (cf. [RB89]). 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 [DN07, BBC+19], 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, informal) *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 [GSZ20], 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 [GSZ20], here we can match the amortized cost of the best known semi-honest protocol even when cheating occurs. However, unlike the protocol from [GSZ20], 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 [BFO12, BBC+19, GS20], 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 [BBC+19, BGIN19], the verification cost is logarithmic in the size of the circuit. This is similar to a verification procedure from [GS20] and better than the square-root complexity of an earlier triple verification protocol from [BBC+19]. Compared to the protocol from [GS20], 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 [BBC+19]. Another advantage of our triple verification protocol over that of [GS20] 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 [Sha79]. 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 [GS20], 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.

1.2 Technical Overview

To present our protocols, it will be convenient to use the complexity-theoretic notion of a *straight-line program*: a sequence of instructions, where each instruction can specify an atomic operation such as addition or multiplication of memory registers (see Section 2.1 for details). Any circuit C with S multiplication gates can be transformed into a straight-line program P of (multiplicative) size S using any topological order of the gates, where the width W is the maximal number of values that need to be kept in memory at any given time.

Consider secure evaluation of a straight-line program P. As is common in the setting of full security, the high-level structure of our protocol breaks the computation of P into segments, and executes each segment one by one via a semi-honest protocol. Before moving to the next segment, the parties run a sub-protocol verifying correctness of the previous. If the verification is accepted, the parties are safe to continue to the next segment, and the "last approved state" is updated; if verification is rejected, then the task of the protocol is to identify a pair of parties at least one of whom is corrupted (we will refer to these two parties as a *semi-corrupt pair*), and recover back to the last approved state with these parties removed.

Intuitively, because of the segment structure, redoing the computation from the last approved state (once recovered) will incur overhead comparable only to the segment size, sublinear in the full program size S. And because each pair of removed parties includes at least one corrupt party, this process can occur at most t times. The goal and challenge is thus to achieve these tasks—namely, (1) verifying the semi-honest computation, (2) identifying a semi-corrupt pair of parties, and (3) recovering the information necessary to revert to the last state—with overall communication *sublinear* in S.

Verifying the semi-honest computation. A promising first approach for verification with sublinear communication is to make use of distributed zero-knowledge (ZK) proofs recently developed by Boneh et al. [BBC+19]. A distributed ZK proof system enables a prover party to prove a statement x in zero knowledge, where x is held in a distributed manner across multiple verifiers. In their work, Boneh et al. propose two approaches for using distributed ZK proofs to compile a protocol with semi-honest security to a protocol with malicious security *with abort* with sublinear $o(S)$ additive communication overhead.

The first approach in [BBC+19] is for each party to independently prove via distributed ZK that its messages in the semi-honest protocol were computed honestly. Here the "statement" x proved by party i is the collection of all its incoming and outgoing messages within the semi-honest protocol, which is known in entirety by party i (Prover), and held distributed across the other parties (Verifiers). This approach works if the semi-honest protocol satisfies certain properties: in particular, that each message sent by party i is computable as a low-degree function of inputs that are *robustly* shared among the parties (that is, the shares held by the honest parties alone determine the secret), and where party i *knows all the shares*.

This approach was used in [BBC+19] to obtain a maliciously secure protocol with abort in the setting of *3 parties and one corruption*. This 3-party protocol was extended to *full security* in [BGIN19] with small additional overhead. Their main observation is that in each proof of party i, the messages sent by the verifiers depend only on public randomness and on inputs that are known to the prover i. This means that once the randomness is chosen, the prover knows what messages are expected to be sent by the verifiers. Thus, in case the proof is rejected, the prover can identify a cheating verifier and then the prover together with the accused verifier are eliminated from the

protocol. That is, in cases where the *first* approach of Boneh et al. [BBC+19] successfully provides security with abort, then the techniques of Boyle et al. [BGIN19] will boost to full security with low overhead.

However, this first approach does *not* work (even for security with abort) for protocols with more than 3 parties; more accurately, for cases with more than a single corrupted party. The problem is that the requirement on the underlying semi-honest protocol is satisfied in the special protocol used in [BGIN19] where there is a single corrupted party, but not necessarily for existing protocols with 2 or more corruptions. To see this, consider for example the DN protocol [DN07] for multiplying shared inputs (this is the fastest semi-honest protocol for any number of parties known to this date). In this protocol, each party locally computes an additive sharing of the output, masks it and sends it to party P_1, which reconstructs the masked output and sends it back to the parties. When applying the above approach, party P_1 is required to prove that it sent the correct message in the second round. However, its message is not a function of inputs that are robustly secret shared between the parties, since the inputs are *additive* shares of the masked output. Thus, it is not possible to use the above machinery to achieve malicious security.

To overcome this challenge, a second approach was suggested by Boneh et al. [BBC+19]. The idea in this approach is that all parties *jointly* emulate the role of (a single) prover in the protocol, and instead of proving correctness of the individual sent messages, the parties simply verify that the collective sharing they hold on the output wire of each multiplication gate is consistent with the corresponding sharing of the gate inputs. Here the parties leverage the fact that after the semi-honest execution, they hold a robust secret sharing of both the values on the input and output wires. Based on this approach, a second protocol was presented by [BBC+19], that achieves malicious security with abort for any constant number of parties, at the same amortized cost of the underlying semi-honest multiplication protocol used. Note that the joint proof executed after the semi-honest execution is eventually a protocol to verify the correctness of a set of multiplication triples (each triple corresponds to the two inputs and to the output of one multiplication gate). This protocol indeed serves as the starting point of this work.[4]

However, unlike the first compiler approach, there no longer exists any single prover that knows all the inputs in the verification protocol and can identify a cheating verifier. Moreover, once the verification protocol ends with a reject, we do not know whether cheating took place in the semi-honest execution or in the verification protocol itself. This means that identifying a semi-corrupt pair of parties toward attaining full security remains a challenge.

Identifying a semi-corrupt pair. To solve this problem we first make the following crucial observation: while the parties cannot use the first single-prover approach to prove correctness of sent messages in the semi-honest execution, it can be used to prove correctness of messages sent in the *verification* protocol, since each of these messages is a 2-degree function of inputs that are robustly shared among the parties. Thus, we add a step inside the verification protocol where each party proves that the messages it sent during the verification protocol are correct (given the shared inputs to the verification protocol). Once this is proved, the parties know that if the output of the verification protocol is reject, then cheating took place in one of the executions of the semi-honest multiplication protocol. We note that the proof of each party in the verification protocol may also be rejected. However, in this case we can utilize the idea of [BGIN19] to let the prover identify

[4] We will later discuss an alternative to the verification protocol from [BBC+19] that can be used to improve the sublinear additive term.

a cheating verifier. We thus have that the verification protocol ends with one of the next three possibilities: (1) the parties output accept. In this case, the parties know that no cheating took place; (2) the parties hold a semi-corrupt pair of parties to eliminate; (3) the parties output reject knowing that cheating took place in the semi-honest execution to compute the circuit.

The case to address is possibility (3). In this case, the parties will proceed to perform fault localization by running a binary search on the computation segment. In each step of the search, the parties execute the verification protocol described so far on a smaller set of multiplication triples (each triple corresponds to the inputs and output of a single multiplication). The property that we have here is that if the search ends without finding a pair of conflicting parties, then it means that a specific multiplication triple which is incorrect was obtained. It remains to run any fully secure protocol to find a pair of parties with conflicting views in the computation of this single incorrect triple. As this is executed at most once per corrupt party in the entire protocol and for a single multiplication, any implementation of such a localization protocol will suffice.

At the conclusion of this phase, either all parties were accepted as performing the computation segment honestly, or a semi-corrupt pair has been identified.

Recovering back to the last approved state. If the semi-corrupt pair is made up of parties i and j, then the goal is to remove these parties from the protocol and recover the computation. The challenge is that some of the shares that parties i and j hold must be transferred to parties that are still active in the protocol. To verify this statement, note that the access structure of the replicated secret sharing includes all sets of $t+1$ parties, but must include all sets of t parties after the removal of the semi-corrupt pair. In any current set T of $t+1$ parties that includes both parties i and j, only $t-1$ parties remain after their removal. Therefore, the shares of this set must be provided to a party k that is not part of T.

The main complication here arises from the fact that the corrupted parties may try to change these shares during the process. To prevent this, we use an authentication mechanism that computes an authentication tag for a set of shares as an affine function of a secret key. The key is used by this set of parties for the *entire* computation, and is shared in an authenticated way across *all* parties. Given the reconstructed authentication key, party k can verify the authenticity of a set of shares.

A straightforward solution is to have all the parties in T provide party k with their sets of shares. At least one set passes authentication since at least one party out of the $t+1$ parties in T is honest. We observe that there is a more efficient approach. If $t=1$ then party k must be honest and therefore all the parties can provide it with their inputs, letting party k complete the computation. If $t>1$ then there must be at least one party $\ell \in T$ such that $\ell \neq i,j$. Party ℓ transmits the shares of T to party k who proceeds to use the authentication key to check the shares. If the authentication passes, then party k has the appropriate shares and the computation can resume without parties i and j. However, if the authentication fails then k and ℓ are a new semi-corrupt pair and both semi-corrupt pairs must be removed from the protocol. This process continues until either the protocol is resumed in the same segment with a new threshold $t' < t$ and $2t'+1$ active parties, or alternatively an honest party is identified and is given all of the inputs. We observe that when using this approach, the cost incurred by the recovery process is offset by the reduced communication cost of computing future segments (due to having less parties participating in the computation). Consequently, the cost of the recovery process is amortized away over the entire computation.

Summary of the fully secure protocol. To sum up, our protocol works by dividing the circuit into segments. At the beginning of each segment, the parties hold shares of the inputs to this segment and authentication tags over these shares. Then, the parties compute the segment using a semi-honest protocol (and authentication tags for the output layer of the segment), followed by a verification step with sublinear communication complexity. If the verification succeeds, the parties proceed to the next segment. Otherwise, the parties find a pair of parties to eliminate, remove them by updating the secret sharing on the input layer to the segment using the authentication tags, and recompute the segment with less parties. The size of the segment is set such that the amortized communication complexity per gate is the same as that in the semi-honest protocol.

The simpler case of security-with-abort. As an intermediate and independently useful step, we develop an improved circuit evaluation protocol that is only secure-with-abort and has the same amortized communication cost as the baseline semi-honest protocol. In this protocol, once the parties reject the single proof of some party, or identify that cheating took place in the semi-honest execution, they simply abort the protocol. The goal of achieving security-with-abort for general arithmetic circuits reduces to verifying the consistency of S secret-shared *multiplication triples* (a_i, b_i, c_i), namely verifying that $c_i = a_i \cdot b_i$ for every i. An efficient design of such a "triple verification" subprotocol lies at the heart of most efficient protocols in the honest majority setting. However, until recently, such protocols (e.g., the ones from [BFO12, CGH+18, NV18]) had communication cost that scales linearly with S. The goal of triple verification with sublinear communication for $t < n/2$ corrupted parties has recently been met in [BBC+19] and [GS20] (and for $t < n/3$ corrupted parties in [FL19]). However, our protocol has several advantages compared to these previous protocols.

The triple verification protocol of [GS20] has similar asymptotic communication to ours (logarithmic in S), but it is formulated in an ad-hoc way. In particular, it only applies to triples that were shared using Shamir's secret-sharing scheme, and requires $O(\log S)$ rounds to get the best achievable communication of $O(\log S)$ field elements. Our protocol is described and analyzed for any linear secret sharing scheme and the number of rounds in the verification protocol can be made constant (still with $O(\log S)$ communication) using the Fiat-Shamir transform [FS86]. The latter is enabled by the use of a *public-coin* verification protocol. In contrast, in [GS20], each step of the verification protocol consists of a joint secure computation and so it is not clear how to apply the Fiat-Shamir transform to reduce rounds of this protocol. Our approach is also more general in that it can be based on any distributed zero-knowledge proof for degree-2 languages, which in turn reduces to zero-knowledge fully linear IOPs (zk-FLIOP) for such languages [BBC+19]. This reduces the goal of further improving concrete efficiency to an improved design of these simpler building blocks.

In the secure-with-abort protocol of [BBC+19], the sublinear additive term scales linearly with the *square root* of the number of triples (or circuit size) S, and the protocol is presented only for *constant* number of parties (though the latter limitation is not inherent). More importantly, this protocol relies on a special property of the underlying zk-FLIOP that makes the prover's computation easy to distribute. In contrast, our protocol works for an arbitrary number of parties, the sublinear additive term is *logarithmic* in S, and the protocol can be generally based on any zk-FLIOP for the degree-2 language of consistent triples.

This improvement is achieved by eliminating the need to distribute the prover, and instead relying on a small number of proofs in the easier single-prover setting. This allows us to benefit from

the logarithmic proof size of the recursive zk-FLIOP construction from [BBC+19]. The original approach from [BBC+19] could not rely on the recursive zk-FLIOP construction because of its higher prover complexity that made distributing the prover difficult. Instead, it relies on a non-recursive variant with $O(\sqrt{S})$ proof size.

Indeed, identifying a means for leveraging the single-prover approach of [BBC+19] with $t \geq 2$ is the technical core of our solution. The challenge, observed in [BBC+19], is that protecting against possible collusions between prover and verifier(s) requires the prover's *statement* to be held robustly across the verifiers—e.g., taking the statement to include all parties' robust secret shares of values computed in the semi-honest protocol. But, in such case, the proving party *does not know* the full statement that he is supposed to prove, as he is not privy to all of these shares.

We instead take a new approach, applying the distributed zero-knowledge proofs in a different way. Our new insight is that for any set of robust secret shares of a value x, we can simultaneously view this set of shares as a robust sharing of *each secret share* x_i *itself*. For example, if parties hold Shamir secret shares of a value corresponding to a polynomial evaluation at input 0, we can symmetrically view these as secret shares of each party's evaluation at corresponding input i. Thus "for free" we already have a robust encoding of the values held and computed on by each party, encapsulating a robustly held statement on which the party can prove correctness.

We provide a detailed analysis of the concrete efficiency of our secure-with-abort protocol and comparison with previous work in Section 4.3.

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 [Cle90]. 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) *The class of* Straight-line programs 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*.*

Note that for simplicity the definition assumes that each party has a single input and receives a single output. Our constructions can be easily adapted to the setting where there are multiple inputs or outputs per party. We remark that every arithmetic circuit with S multiplication gates can be converted into a straightline program of size S by sorting its gates in an arbitrary topological order.

2.2 MPC Security Definition

In this work, we consider security against a malicious adversary who may act arbitrarily. We use the standard definition of security based on the ideal/real model paradigm [Can00, Gol04]. When we say that a protocol "securely computes an ideal functionality with abort", then we consider non-unanimous abort (sometimes referred to as "selective abort"). This means that the adversary first receives the output, and then determines for each honest party whether they will receive abort or receive their correct output. It is easy to modify our protocols so that the honest parties unanimously abort by running a single (weak) Byzantine agreement at the end of the execution before the output is revealed [GL05]; we therefore omit this step for simplicity. When we say that a protocol "securely computes an ideal functionality", then we consider full security, including guaranteed output delivery.

2.3 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, while $v = \mathsf{reconstruct}([\![v]\!]_T, i)$ for T only if $|T| \geq t + 1$. 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 the procedure, 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 received 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 [FY19] 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 [RB89] based on pairwise authentication of shares.

Local linear operations. 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 $[\![c+x]\!] = c + [\![x]\!]$.

Multiplication. 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 will be able to locally generate a consistent sharing $[\![x^i]\!]$, where x^i is the share of x held by party P_i.

2.3.1 Instantiation 1: Replicated Secret Sharing [ISN89]

To share a secret x, for each subset T of t parties the dealer hands a share x_T to the parties in $\overline{T} = \{P_1, \ldots, P_n\} \setminus T$, under the constraint that $x = \sum_{T \subset \{P_1,\ldots,P_n\}:\ |T|=t} x_T$ (note that when $n = 2t+1$ as in our setting, this is equivalent to giving one share to each subset of $t+1$ parties). 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 and allows local multiplication to obtain an additive sharing of the product.

Finally, local conversion from $[\![x]\!]$ to $[\![x^i]\!]$ can be done by sharing each x_T for which $P_i \in T$ separately. Specifically, for each T with $P_i \in T$, let $x_{T,S}$ be the share of x_T held by the set S of $t+1$ parties. Then, $[\![x_T]\!]$ is defined by setting $x_{T,T} = x_T$ and $x_{T,S} = 0$ for each $S \neq T$. Finally, define $[\![x^i]\!] = \{[\![x_T]\!]\}_{T \mid P_i \in T}$.

Pair-wise 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 of T, which both are a part of. This can be done efficiently by having these parties compare a hash of their joint shares. Observe that if pair-wise inconsistency is detected, we can ask the dealer to publish the conflicted share, as in this case, this share is already known to the adversary.

2.3.2 Instantiation 2: Shamir's Secret Sharing [Sha79]

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 $x \cdot y$. Since $n = 2t + 1$, the parties can multiply their shares with the appropriate Lagrange coefficients to obtain an additive sharing 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.

2.4 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 $\vec{u}_J$. Then, for every adversary $\mathcal{A}$ controlling up to t parties, and for every two vectors of shares $\vec{u}_J, \vec{u}'_J$ the view that $\mathcal{A}$ has in the protocol when the honest parties hold $\vec{u}_J$ is computationally indistinguishable from its view when the honest parties hold $\vec{u}'_J$.*

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.

2.4.1 Instantiation: The DN [DN07] 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 [CDI05] (from one set of replicated keys to any number of replicated or shamir sharings) or using interaction but with reduced computational overhead for large number of parties [DN07] (using hyper-invertible matrices). We refer the reader to [CGH+18, LN17] for exact details. We note that in the latter case, by using a PRG, $\langle r \rangle$ can be produced without any interaction, while producing $[\![r]\!]$ requires transmitting roughly one element per party. Thus, overall, the offline step can be processed without any communication (in a way that is efficient only for a constant number of parties) or with communication cost of one sent element per party (for any number of parties).

2.4.2 Other Instantiations for Π_{mult}

While the DN protocol yields small constant communication overhead per party for any number of parties, there are other multiplication protocols that may be preferred in some settings. For example, in the three-party setting, the protocols of [AFL+16, KKW18] have communication cost of 1 ring element per multiplication sent by each party. The protocol of [AFL+16] achieves this in one round of communication, while [KKW18] pushes roughly 1/3 of the communication to an input-independent offline phase. For larger number of parties, the GRR [GRR98] protocol works by having the parties locally compute an additive sharing of the output and then secret share their additive shares to the other parties. By adding all the received shares, the parties can obtain a t-out-of-n secret sharing of the output. Applying the computational optimization where each party P_i distributes seeds to t parties, from which their shares are derived every time P_i is required to share a secret, we have that the communication cost of this protocol is t elements sent by each party per multiplication, in one round of interaction. For the setting of 5 parties and 2 corruptions, for instance, this is translated to sending 2 elements per party. While this is slightly more than the cost of the DN protocol, the round complexity is reduced by half, which may have an advantage in some cases (e.g., when latency of messages is a bottleneck).

2.5 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 [CDI05, DN07]. We remark that for replicated secret sharing, the functionality can be realized without any interaction (except for a setup step) [CDI05], 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, even in an information-theoretic way, by taking $r, r^2, ...$).

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 [PSL80]. Full security of $\mathcal{F}_{\mathsf{bc}}$ is achievable given PKI setup [RB89]. 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 [BBC+19]. 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 [BBC+19], 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 Ideal Functionality $\mathcal{F}^{\text{abort}}_{\text{proveDeg2Rel}}$ - Prove Correctness with Abort

We begin by a protocol that is secure with abort, i.e., it allows a malicious verifier to cause a 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)[5]. 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 it self. 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

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

determine deterministically 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).

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.

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 [BBC+19], there is still one difference. The zero-knowledge property in the definition of [BBC+19] 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. As we will see, one of our technical contribution in this work, is showing that the machinery from [BBC+19] can work even if the prover does not know the shares held by the verifiers.

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 [BBC+19] with low communication.

The idea works as follows. First, the parties define a g-gate as

$$g\left(\nu_1, \ldots, \nu_L\right) = \sum_{\ell=1}^{L/2} \nu_{2\ell-1} \cdot \nu_{2\ell}.$$

We now can write Eq. (1) as

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

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 see more formally 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.

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.

Next, we prove the security of the protocol.

Theorem 3.3 *Protocol 3.2 securely computes $\mathcal{F}_{\text{proveDeg2Rel}}^{\text{abort}}$ over a finite field $\mathbb{F}$, with statistical error $\frac{2\log L+4}{|\mathbb{F}|-5}$ in the $(\mathcal{F}_{\text{coin}}, \mathcal{F}_{\text{rand}})$-hybrid model, in the presence of malicious adversaries controlling up to t parties.*

Proof: Let $\mathcal{S}$ be the ideal world simulator and let $\mathcal{A}$ be the real world adversary. The simulation $\mathcal{S}$ is invoked by $\mathcal{F}_{\text{proveDeg2Rel}}^{\text{abort}}$ handing him an index of the prover i, an output out and the shares of the corrupted parties of each of the inputs. In the simulation, $\mathcal{S}$ plays the role of $\mathcal{F}_{\text{coin}}$ handing all required randomness to $\mathcal{A}$ and $\mathcal{F}_{\text{rand}}$. There are two cases:

CASE 1: P_i IS CORRUPT. In this case, $\mathcal{S}$ also receives P_i's inputs and the honest parties' shares of c. Note first that this means that $\mathcal{S}$ can simulate perfectly the opening of $[\![b]\!]$ and $q(r)$, since it has the honest parties shares' of c and receives the honest parties' shares of the points on q from $\mathcal{A}$ during the simulation. It remains to show how to simulate the opening of $f_1(r)$ and $f_2(r)$. Since $\mathcal{S}$ knows the inputs, it knows the actual values of $f_1(r)$ and $f_2(r)$. Thus, $\mathcal{S}$ only chooses random shares for the honest parties under the constraint that they, together with corrupted parties' shares, will open to the correct values. To see that the view of $\mathcal{A}$ is distributed the same here as in the real execution, observe that for each $e \in \{1,2\}$ we have that

$$f_e(r) = \lambda_0(r) \cdot f_e(0) + \lambda_1(r) \cdot f_e(1) + \lambda_2(r) \cdot f_e(2) \tag{3}$$

where $\lambda_0(r), \lambda_1(r), \lambda_2(r)$ are the Lagrange coefficients. Since the shares of $f_e(0)$ held by the honest parties are uniformly distributed in the field (since they were chosen by $\mathcal{F}_{\text{rand}}$) under the constraint that, together with the shares of the corrupted parties, they will open to $f_e(0)$, then so are their shares of $f_e(r)$. Thus, the distribution is the same in both executions.

Now, if $\mathsf{out} = \mathsf{accept}$, but some honest party P_j outputs reject in any step of the simulation, then $\mathcal{S}$ sends reject_j to $\mathcal{F}_{\text{proveDeg2Rel}}^{\text{abort}}$ and outputs whatever $\mathcal{A}$ outputs. If $\mathsf{out} = \mathsf{reject}$, but the honest parties output accept at the end of the execution, then $\mathcal{S}$ outputs fail and halts.

Observe that when the event where $\mathcal{S}$ outputs fail does not occur, then the simulation is in fact perfect. Thus, the only difference between the simulation and the real execution is the event of $\mathcal{S}$ outputting fail. However, note that the probability of this event is exactly the probability that the execution ends with the honest parties outputting accept when the statement being proved is incorrect (i.e., Eq. (1) does not hold). Thus, $\Pr[\mathsf{fail}] \leq \frac{2\log L+4}{|\mathbb{F}|-5}$, which is exactly the statistical error allowed by the theorem.

CASE 2: P_i IS HONEST. In this case, the output initially received from $\mathcal{F}_{\text{proveDeg2Rel}}^{\text{abort}}$ is always accept. This means that while $\mathcal{S}$ does not know the inputs, it knows that b should be 0 in each iteration and $q(r)$ should equal to $g(f_1(r), \ldots, f_L(r))$ in the last iteration, unless the corrupted parties send incorrect shares when opening these values. Since $\mathcal{S}$ knows the corrupted parties' shares of the inputs, it can simulate the openings correctly.

In more details, for each sharing of $q(1), q(2)$ and $q(3)$ (and $f_1(0), f_2(0), q(0), q(4)$ in the last step) in the simulation, $\mathcal{S}$ plays the role of P_i by sending random shares for the corrupted parties to $\mathcal{A}$. Now, since $\mathcal{S}$ knows the corrupted parties' shares of $c, q(1)$ and $q(2)$, it can compute their shares of $b_l = c - q(1) - q(2)$. Then, it can choose the honest parties' shares under the constraint that $b = \sum_{l=1}^{\log l} \gamma_l b_l$ will reconstruct to 0. Next, $\mathcal{S}$ can use the corrupted parties' shares of $f_1(u), \ldots, f_L(u)$ for each $u \in \{1,2\}$,$q(1)$,$q(2)$ and $q(3)$ to compute the corrupted parties' shares of $f_1(r), \ldots, f_L(r)$ and $q(r)$. Then, it can simulate the execution of the next iteration as before. In the last step, $\mathcal{S}$ uses he corrupted parties' shares of $f_1(0), f_1(1), f_1(2), f_2(0), f_2(1), f_2(2)$ and $q(0), \ldots, q(4)$ to compute

the corrupted parties' shares of $f_1(r), f_2(r)$ and $q(r)$. Then, $\mathcal{S}$ simulates the opening of $b, f_1(r), f_2(r)$ and $q(r)$ a follows:

- To simulate the opening of b, $\mathcal{S}$ chooses random shares for the honest parties under the constraint that all the shares together will reconstruct to 0.
- To simulate the opening of $f_1(r), f_2(r)$, $\mathcal{S}$ chooses random shares for the honest parties.
- To simulate the opening of $q(r)$, $\mathcal{S}$ chooses random shares for the honest parties under the constraint that the reconstructed $q(r)$ will satisfy the equation: $q(r) = g(f_1(r), f_2(r))$.

Then, if $\mathcal{A}$ sends inconsistent shares to some honest party P_j causing any of the opening to fail, then $\mathcal{S}$ sends reject_j to $\mathcal{F}^{\text{abort}}_{\text{proveDeg2Rel}}$. Otherwise, it sends accept_j. Finally, $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs.

We claim that the adversary's view is identically distributed in both the simulation and the real world execution. Observe that $\mathcal{A}$'s view consists of (i) shares sent by P_i for the points on q; (ii) shares by P_i for the points $f_1(0)$ and $f_2(0)$ in the last step of the protocol; (iii) the opened b and (iV) the opened $f_1(r), f_2(r)$ and $q(r)$. Now, by the secrecy of the sharing scheme, it is clear that the up to t shares seen in (i) and (ii) are uniformly distributed in both executions. Next, since the prover P_i is honest, $b = 0$. Thus, since in both executions $\mathcal{A}$ sees random shares that, together with corrupted parties' shares, reconstruct to 0, the view in (iii) is distributed the same in both executions. Finally, the claim that the view in (iv) is also the follows from Eq. (3). Since for each $e \in \{1, 2\}$ we have that $f_e(0)$ is uniformly distributed in the field, it follows that $f_e(r)$ is also uniformly distributed. Recall that in the simulation, $f_e(r)$ is chosen randomly from the field by $\mathcal{S}$. Thus, the distribution of $f_e(r)$ is the same in both executions. To conclude the proof, observe that since P_i is honest it holds that $q(r) = g(f_1(r), f_2(r))$, which means that $q(r)$ is random under the constraint that the equation holds. This is exactly the distribution of $q(r)$ in both executions. This concludes the proof. ■

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 [BBC+19, BGIN19] 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 [BBC+19, BGIN19], this enables to define enough points on the polynomial that allow interpolation. 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 [BBC+19, BGIN19] 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.

3.1.1 Cost Analysis

We now compute the communication cost of our protocol. 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 to yield a small constant cost.

For the computational cost, we remark that while our protocol requires many Lagrange-based 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.

3.1.2 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. We next show how to use the Fiat-Shamir transform [FS86] 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.

To solve this problem, we change the protocol in the following way. Assume that the parties hold a random sharing $[\![s_l]\!]$ for each round l, where s_l is known to the prover P_i (in our protocol, we need three such sharings for each round). Then, when the prover wishes to share an element x, it suffices to send $d_l = x - s_l$ to all the parties, who can then locally compute $[\![x]\!] = [\![s_l]\!] + d_l$. Now, the message sent in each round of the proof is public, and so the parties can compute the randomness on their own.

Note that all the random sharings can be prepared in advanced in one round of interaction. In our protocol, this is translated to preparing $[\![s_{l,1}]\!]$,$[\![s_{l,2}]\!]$ and $[\![s_{l,3}]\!]$ for each $l \in [\log(L-1)]$. Then, in the lth round, the prover P_i locally computes $q(1), q(2), q(3)$, and $d_{l,1} = q(1) - s_{l,1}$, $d_{l,2} = q(2) - s_{l,2}$, $d_{l,3} = q(3) - s_{l,3}$. The random point r is then computed by applying H on the communication transcript so far (i.e., $d_{1,1}||d_{1,2}||d_{1,3}||\cdots||d_{l,1}||d_{l,2}||d_{l,3}$). The prover then sends $\{(d_{l,1}, d_{l,2}, d_{l,3})\}_{l=1}^{\log(L-1)}$ to the parties in one shot, who can then perform the local computations of the protocol in each round. Note that the parties need to ensure that they received the same messages. This can be checked with constant small cost by taking a random linear combination of the messages or using a collision-resistance hash function. Finally, the parties execute the last round of the protocol as in the description, deciding whether to accept or reject.

3.1.3 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 [NV18].

Let $\{(a_k, b_k, c_k)\}_{k=1}^{n}$ be n multiplication triples to verify. First, define two polynomials $f_1(x), f_2(x)$ of degree $n-1$, such that for all $k \in [n]$: $f_1(k) = a_k$ and $f_2(k) = b_k$. Then, let $h(x)$ be a polynomial defined as $h(x) = f_1(x) \cdot f_2(x)$. It follows that for all $k \in [n]$: $h(k) = c_k$. However, $h(x)$ is of degree $2(n-1)$ and so the parties compute $n-1$ more points on h by interpolating over f_1 and f_2 and running a multiplication protocol over the shared points (i.e., locally compute $[\![f_1(k)]\!]$ and $[\![f_2(k)]\!]$ for each $k \in \{n+1, \ldots, 2n-1\}$ and run a multiplication protocol to obtain $h(k)$). Finally, the parties sample a point $r \in \mathbb{F} \setminus \{1, \ldots, 2n-1\}$, locally compute $[\![f_1(r)]\!], [\![f_2(r)]\!]$ and $[\![h(r)]\!]$, open the three secrets and check in the clear that $h(r) = f_1(r) \cdot f_2(r)$. If one of the original triples is incorrect, by the Schwartz-Zippel lemma, this check will pass with probability of at most $\frac{2n-1}{|\mathbb{F}|}$.

To carry-out the above, the parties need to run n multiplications and 3 openings. Using the DN [DN07] protocol (see Section 2.4), it is possible to multiply two shared values with communication cost of 1.5 elements per party. Thus, the communication cost of this batched check is $1.5(n-1) + 3(n-1) \approx 4.5n$ field elements per party.

The overall communication per party of running n proofs in parallel is therefore

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

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 3.4. 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.

FUNCTIONALITY 3.4 ($\mathcal{F}^{\text{cheatIdntfy}}_{\text{proveDeg2Rel}}$- Prove Correctness with Cheating Identification)
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 $\in \{\text{accept}, \text{reject}\}$. 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.

To compute functionality we use Protocol 3.2 from the previous section, with one additional step: in case 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 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 3.5 *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 in Protocol 3.6. 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.

PROTOCOL 3.6 (Securely Computing $\mathcal{F}_{\text{proveDeg2Rel}}^{\text{cheatIdntfy}}$)

- **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.
- **The protocol:**
  1. Same as in Protocol 3.2.
  2. (a) The parties run Step (3a)-(3g) in Protocol 3.2.
     (b) The parties run $\mathsf{reconstruct}([\![b]\!], j)$, $\mathsf{reconstruct}([\![q(r)]\!], j)$, $\mathsf{reconstruct}([\![f_1(r)]\!], j)$ and $\mathsf{reconstruct}([\![f_2(r)]\!], j)$ for each $j \in [n]$, where each message is sent via $\mathcal{F}_{\mathsf{bc}}$. If the party received $\perp$ in any of these executions, or if $q(r) \neq g(f_1(r), \ldots, f_L(r))$ or if $b \neq 0$, then the parties output reject. Otherwise, the parties output accept.
     (c) If the parties output reject, then P_i identifies a party P_j who sent incorrect messages in the previous step, and sends j to the other parties via $\mathcal{F}_{\mathsf{bc}}$. Then, the parties output the pair (i, j).

Theorem 3.7 *If the inputs' shares of all parties are known to the prover P_i, then Protocol 3.6 securely computes $\mathcal{F}_{\text{proveDeg2Rel}}^{\text{cheatIdntfy}}$ with statistical error $\frac{2\log L+4}{|\mathbb{F}|-5}$ in the $(\mathcal{F}_{\text{coin}}, \mathcal{F}_{\mathsf{bc}})$-hybrid model, in the presence of malicious adversaries controlling up to t parties.*

To prove Theorem 3.7, we can follow the proof of Theorem 3.3 and add a step where a cheating party is identified when the parties output reject. Specifically, when the prover is corrupted, then the real world adversary $\mathcal{A}$ broadcasts an index which is then handed by the simulator $\mathcal{S}$ to the ideal functionality $\mathcal{F}_{\text{proveDeg2Rel}}^{\text{cheatIdntfy}}$. In contrast, when the prover is honest, then since $\mathcal{S}$ knows the corrupted parties' shares, then it can compute the messages that should be sent by $\mathcal{A}$ and, playing the role of P_i, identify a cheater. There is however a subtle issue that can be easily missed here. In the former case, where P_i is corrupted and $\mathcal{S}$ emulates the honest parties, the simulation works by choosing shares for the honest parties, such that all shares will reconstruct to the input held by P_i, which is known to $\mathcal{S}$. However, note that now in our protocol P_i knows all shares and so for the simulation to be correct, $\mathcal{S}$ must use the *exact* shares held by the honest parties. To overcome this, we could change the definition of $\mathcal{F}_{\text{proveDeg2Rel}}^{\text{cheatIdntfy}}$ and say that in this case it hands $\mathcal{S}$ the honest parties' shares as well. Since in our use of the protocol, the inputs of P_i determine deterministically the honest parties' shares, then $\mathcal{S}$ can compute these shares and the simulation is perfect. Thus, there is no need instruct $\mathcal{F}_{\text{proveDeg2Rel}}^{\text{cheatIdntfy}}$ to explicitly hand these shares to the simulator.

Another issue that may arise and was slightly ignored in our description of the protocol, is what happens when the prover P_i deals inconsistent shares when sharing points on the different polynomials defined during the protocol. In the protocol, the prover uses a VSS protocol to share these shares. By our definition in Section 2, this means that after P_i sends its shares, a consistency check is carried-out. The way this check is performed and its outcome depends on the secret sharing scheme being used. Nevertheless, for linear secret sharing schemes we consider in this work, the dealer always knows the messages that should be sent by all parties in the consistency check, and so we can employ the same mechanism as above, letting the prover point at a cheating party.

Reducing interaction via the Fiat-Shamir transform. Since the only difference between Protocol 3.2 and Protocol 3.6 is in the last round, we can apply the Fiat-Shamir transform [FS86] on this protocol to achieve constant number of rounds as explained in Section 3.1.2.

Batching n proofs together and communication cost. In section 3.1.3 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 2.4. 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 2.3 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 [GS20] for finite fields and arbitrary number of parties. When using replicated secret sharing as the underlying secret sharing scheme, the obtained protocol improves upon the result of [BBC+19] for constant number of parties over the ring $\mathbb{Z}_{2^k}$; while the additive sub-linear term in [BBC+19] 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 [GIP+14, LN17, CGH+18]).

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

$$\forall i \in [n] : \sum_{k=1}^{m} \theta_k \cdot ([\![x_k^i]\!] \cdot [\![y_k^i]\!]) - [\![\psi^i]\!] = 0. \tag{4}$$

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 appears in Protocol 4.2.

PROTOCOL 4.2 (Securely Computing $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{abort}}$)

- **Inputs:** The parties hold a vector of $3m$ sharings $((\llbracket x_1 \rrbracket, \llbracket y_1 \rrbracket, \llbracket z_1 \rrbracket), \ldots, (\llbracket x_m \rrbracket, \llbracket y_m \rrbracket, \llbracket z_m \rrbracket))$.
- **The protocol:**
  1. The parties call $\mathcal{F}_{\text{coin}}$ to receive random $\theta_1, \ldots, \theta_m \in R$.
  2. The parties locally compute
$$\langle \psi \rangle = \left\langle \sum_{k=1}^{m} \theta_k \cdot (x_k \cdot y_k) \right\rangle = \sum_{k=1}^{m} \theta_k \cdot (\llbracket x_k \rrbracket \cdot \llbracket y_k \rrbracket).$$
  3. Denote the additive share of ψ held by P_i by ψ^i. Then, each party P_i verifiably secret shares (VSS) ψ^i to the other parties.
  4. For each $i \in [n]$:
     (a) The parties locally convert $\llbracket x_k \rrbracket, \llbracket y_k \rrbracket$ to $\llbracket x_k^i \rrbracket, \llbracket y_k^i \rrbracket$ for each $k \in [m]$.
     (b) The parties define $\llbracket c^i \rrbracket = \llbracket \psi^i \rrbracket$, $\llbracket a_k^i \rrbracket = \theta_k \cdot \llbracket x_k^i \rrbracket$ and $\llbracket b_k^i \rrbracket = \llbracket y_k^i \rrbracket$.
     (c) The parties send $\llbracket c^i \rrbracket$ and $\left(\llbracket a_k^i \rrbracket, \llbracket b_k^i \rrbracket\right)_{k=1}^{m}$ to $\mathcal{F}_{\text{proveDeg2Rel}}^{\text{abort}}$.
     (d) If any party received reject from $\mathcal{F}_{\text{proveDeg2Rel}}^{\text{abort}}$ then it sends it to the other parties and outputs reject.
  5. If the parties received accept from $\mathcal{F}_{\text{proveDeg2Rel}}^{\text{abort}}$ in all n invocations, then they proceed to the next step.
  6. The parties locally compute $\llbracket \beta \rrbracket = \sum_{k=1}^{m} \theta_k \cdot \llbracket z_k \rrbracket - \sum_{i=1}^{n} \llbracket \psi^i \rrbracket$.
  7. The parties run $\mathsf{reconstruct}(\llbracket \beta \rrbracket, i)$ for each $i \in [n]$. If any party received $\perp$ or if $\beta \neq 0$, then it outputs reject. Otherwise, it outputs accept.

Theorem 4.3 *Protocol 4.2 securely computes $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{abort}}$ over fields with statistical error $\frac{1}{|\mathbb{F}|}$ in the $(\mathcal{F}_{\text{coin}}, \mathcal{F}_{\text{proveDeg2Rel}}^{\text{abort}})$-hybrid model, in the presence of malicious adversaries controlling up to t parties.*

Proof: Let $\mathcal{S}$ be the ideal world adversary and let $\mathcal{A}$ be the real world adversary. $\mathcal{S}$ is invoked by $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{abort}}$ which sends it all the corrupted parties' shares of $(x_1, y_1, z_1), \ldots, (x_m, y_m, z_m)$ and $\mathsf{out} \in \{\mathsf{reject}, \mathsf{accept}\}$ and $d_k = z_k - x_k \cdot y_k$ for all $k \in [m]$.

In the simulation, $\mathcal{S}$ plays the role of $\mathcal{F}_{\text{coin}}$, thus choosing and handing $\mathcal{A}$ random $\theta_1, \ldots, \theta_m \in \mathbb{F}$. In addition, $\mathcal{S}$ plays the role of $\mathcal{F}_{\text{proveDeg2Rel}}^{\text{abort}}$. The simulation begins with $\mathcal{S}$ choosing random shares for the corrupted parties for each ψ^j where P_j is an honest party, and handing these to $\mathcal{A}$. Then, $\mathcal{S}$ receives the honest parties' shares for each ψ^i where P_i is a corrupted party (if the shares dealt by $\mathcal{A}$ are inconsistent, then by the VSS properties, the honest parties detect it and abort). Since an honest majority exists, $\mathcal{S}$ can use the honest parties' shares to compute ψ^i for all i for which P_i is corrupted and the corrupted parties' shares. Thus, it can also check whether Eq. (4) holds or not (recall that it receives x_k^i, y_k^i and z_k^i from the ideal party computing $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{abort}}$), and so it can simulate $\mathcal{F}_{\text{proveDeg2Rel}}^{\text{abort}}$ handing accept or reject accordingly. If the output is reject for any $i \in [n]$ (which can happen if Eq. (4) does not hold or if $\mathcal{A}$ instructs $\mathcal{F}_{\text{proveDeg2Rel}}^{\text{abort}}$ to output reject), then $\mathcal{S}$ hands reject to $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{abort}}$ and outputs whatever $\mathcal{A}$ outputs.

At this point, if the simulation has not ended with a reject, then it means that all ψ^i are correct. Thus, $\beta = \sum_{k=1}^{m} \theta_k \cdot (z_k - x_k \cdot y_k) = \sum_{k=1}^{m} \theta_k \cdot d_k$. Since $\mathcal{S}$ can compute the corrupted parties'

shares of β, it can now choose shares for the honest parties, given the value of β and given the corrupted parties' shares. Then, using this shares, it simulates the honest parties in the execution of the reconstruct procedure. Now, there are two case:

- If $\mathsf{out} = \mathsf{accept}$ but an honest party P_j received inconsistent shares from $\mathcal{A}$, then $\mathcal{S}$ sends reject_j to $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{abort}}$. Otherwise, it must hold that $\beta = 0$ (by the properties of the reconstruct procedure) and so $\mathcal{S}$ sends accept_j to $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{abort}}$.
- If $\mathsf{out} = \mathsf{reject}$ but the honest parties output accept (i.e., $\beta = 0$), then $\mathcal{S}$ outputs fail and halts.

Observe that $\mathcal{A}$'s view consists of (i) t random shares of ψ^j for each honest party P_j; (ii) message handed by $\mathcal{F}_{\text{proveDeg2Rel}}^{\text{abort}}$ and (iii) the revealed β. By the secrecy of the secret sharing scheme, the view in (i) is the same in the simulation and the real execution. As we saw above, $\mathcal{S}$ can simulate $\mathcal{F}_{\text{proveDeg2Rel}}^{\text{abort}}$ perfectly and so this is the same for (ii) as well. Finally, since $\mathcal{S}$ knows the value of β, it can simulate its opening perfectly. The only difference between the simulation and real execution is thus the event where $\mathcal{S}$ outputs fail. Note that this happens when $\exists k \in [m] : \ d_k \neq 0$ (which is why $\mathsf{out} = \mathsf{reject}$) but $\beta = \sum_{k=1}^{m} \theta_k \cdot d_k = 0$. This can happen with probability $\frac{1}{|\mathbb{F}|}$, which is exactly the statistical error allowed by the theorem. ■

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 3.1.3, we conclude that the cost is $O(n \log m + n)$.

Verifying multiplications with any distributed zero-knowledge proof. As explained in Section 3, we can realize $\mathcal{F}_{\text{proveDeg2Rel}}^{\text{abort}}$ with any distributed zero-knowledge proof. Thus, we obtain a protocol to verify correctness of multiplications that makes use of any distributed zero-knowledge proof for degree-2 relations, by calling the distributed zero-knowledge proof protocol n times.

4.2 The Main Protocol

We are now ready to present the main protocol that computes (with abort) any functionality represented by a straight-line program. The protocol works in the $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{abort}}$-hybrid model as follows:

$\underline{\Pi_f}$**:**

1. Each party verifiably secret shares its inputs to the other parties.
2. The parties compute the program over the shared inputs instruction by instruction in a predetermined order induced by the program. For multiplication instructions, they use the protocol Π_{mult} (as defined in Section 2.4).

3. Let $((\llbracket x_1 \rrbracket, \llbracket y_1 \rrbracket, \llbracket z_1 \rrbracket), \ldots, (\llbracket x_m \rrbracket, \llbracket y_m \rrbracket, \llbracket z_m \rrbracket))$ be the sharings of the inputs and outputs of all multiplication instructions. The parties run a batch consistency check over all these shares. If the check fails, then the parties abort the protocol. Otherwise, they proceed to the next step.

4. The parties send $((\llbracket x_1 \rrbracket, \llbracket y_1 \rrbracket, \llbracket z_1 \rrbracket), \ldots, (\llbracket x_m \rrbracket, \llbracket y_m \rrbracket, \llbracket z_m \rrbracket))$ to $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{abort}}$ to receive out. If $\mathsf{out} = \mathsf{reject}$, then the parties abort the protocol. Otherwise, $\mathsf{out} = \mathsf{accept}$ and the parties proceed to the next step.

5. For each output instruction, the parties reconstruct the secret towards the party that should receive the output. If the party received inconsistent shares, then it aborts the execution and outputs $\perp$. Otherwise, it computes the output and outputs it.

We prove the security of the protocol in Appendix A.

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 2.3). 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 [AFL+16, KKW18] (for 3 parties) or the DN protocol [DN07] for any number of parties. As explained in Section 2.4 (see also [BBC+19]), 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. With Shamir's secret sharing, the cost of semi-honest DN is at most 2.5 elements per party (or 1.5 but with computational overhead that grows exponentially with the number of parties). The cost of Π_{mult} 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, one may need to repeat the verification protocol to achieve sufficiently small statistical error. Since in most cases the cost of a single execution is already extremely low, the overall cost will remain low even though the sub-protocol is repeated several times.

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 the underlying secret sharing scheme. Recall that in this

# 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}.

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 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

	Communication per party (field elements)	# of rounds
Nordholt et al. [NV18]	$O(m+n)$	$O(1)$
Boneh et al. [BBC+19]	$O(n\sqrt{m}+n)$	$O(1)$
Goyal et al. [GS20]	$O(n \log m + n)$	$O(\log m)$
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.

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 [BBC+19, GS20] achieve sublinear communication, whereas [NV18] achieves only linear communication in the amount of verified triples m. Our improvement compared to [BBC+19] is that our sublinear additive term is logarithmic in m rather than just square root of m. Compared to [GS20], we are able to use the Fiat-Shamir transform to achieve constant number of rounds (see Section 3.1.2), 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 parties in dispute, 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 Protocol 4.2 from Section 4.1 and is formally described in Protocol 5.2. 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 ever revealed. To identify 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 be 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 2.3). Finally, for case (iv), if the

FUNCTIONALITY 5.1 ($\mathcal{F}^{\mathsf{full}}_{\mathsf{vrfy}}$- Verification of Multiplications with Cheating Identification)
Let $\mathcal{S}$ be the ideal world adversary controlling a subset $< n/2$ of corrupted parties. The functionality $\mathcal{F}^{\mathsf{full}}_{\mathsf{vrfy}}$ is invoked by the honest parties sending their shares of m multiplication triples $\{(x_k, y_k, z_k)_{k=1}^m\}$ to $\mathcal{F}^{\mathsf{full}}_{\mathsf{vrfy}}$.
Then, $\mathcal{F}^{\mathsf{full}}_{\mathsf{vrfy}}$ 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 $\mathcal{F}^{\mathsf{full}}_{\mathsf{vrfy}}$ 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 $\mathcal{F}^{\mathsf{full}}_{\mathsf{vrfy}}$ with at least one of them being a corrupted party. Then, $\mathcal{F}^{\mathsf{full}}_{\mathsf{vrfy}}$ hands (i, j) to the honest parties.
- If $\mathcal{F}^{\mathsf{full}}_{\mathsf{vrfy}}$ sent reject, then $\mathcal{S}$ chooses one of the next two options:
  - Send a pair of indices (i, j) to $\mathcal{F}^{\mathsf{full}}_{\mathsf{vrfy}}$ with at least one of them being a corrupted party. Then, $\mathcal{F}^{\mathsf{full}}_{\mathsf{vrfy}}$ hands (i, j) to the honest parties.
  - Ask $\mathcal{F}^{\mathsf{full}}_{\mathsf{vrfy}}$ to find a pair of conflicting parties in the $\bar{k}$th multiplication, $1 \leq \bar{k} \leq m$. Then, $\mathcal{F}^{\mathsf{full}}_{\mathsf{vrfy}}$ commands the honest parties to send their inputs, randomness and views in the execution to compute the $\bar{k}$th triple. Then, based on this information, $\mathcal{F}^{\mathsf{full}}_{\mathsf{vrfy}}$ computes the messages that should have been sent by each corrupted party, and finds a pair of parties P_i, P_j , where P_j received an incorrect message. Then, $\mathcal{F}^{\mathsf{full}}_{\mathsf{vrfy}}$ sends (i, j) to the honest parties and $\mathcal{S}$.

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 has 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.

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 causes 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 $\log m \cdot \frac{1}{|\mathbb{F}|}$.

PROTOCOL 5.2 (Securely Computing $\mathcal{F}_{\text{vrfy}}^{\text{full}}$)

- **Inputs:** The parties hold $\{([\![x_k]\!], [\![y_k]\!], [\![z_k]\!])\}_{k=1}^{m}$.
- **The protocol:**

1. The parties call $\mathcal{F}_{\text{coin}}$ to receive random $\theta_1, \ldots, \theta_m \in R$.
2. The parties locally compute $\langle \psi \rangle$ as in Protocol 4.2. Denote the additive share of ψ held by P_i by ψ^i. Then, each party P_i verifiably secret shares (VSS) ψ^i to the other parties.
3. For each $i \in [n]$:
   (a) The parties locally convert $[\![x_k]\!], [\![y_k]\!]$ to $[\![x_k^i]\!], [\![y_k^i]\!]$ for each $k \in [m]$.
   (b) The parties define $[\![c^i]\!] = [\![\psi^i]\!]$, $[\![a_k^i]\!] = \theta_k \cdot [\![x_k^i]\!]$ and $[\![b_k^i]\!] = [\![y_k^i]\!]$. Then, they send $[\![c^i]\!]$ and $\left([\![a_k^i]\!], [\![b_k^i]\!]\right)_{k=1}^{m}$ to $\mathcal{F}_{\text{proveDeg2Rel}}^{\text{cheatIdntfy}}$.
   (c) If the parties received $\mathsf{reject}, (i,j)$ from $\mathcal{F}_{\text{proveDeg2Rel}}^{\text{cheatIdntfy}}$ then they output it and halt.
4. If the parties received accept from $\mathcal{F}_{\text{proveDeg2Rel}}^{\text{cheatIdntfy}}$ in all n invocations, then they proceed to the next step.
5. The parties locally compute $[\![\beta]\!] = \sum_{k=1}^{m} \theta_k \cdot [\![z_k]\!] - \sum_{i=1}^{n} [\![\psi^i]\!]$.
6. The parties run $\mathsf{reconstruct}([\![\beta]\!], i)$ for each $i \in [n]$, with all messages being sent via $\mathcal{F}_{\mathsf{bc}}$. Then:
   - If the parties receive $\perp$ then it means that the shares are inconsistent, and so the parties pick the first pair of parties P_i, P_j for which pairwise-inconsistency exists and output $\mathsf{reject}, (i,j)$.
   - If $\beta = 0$, then the parties output accept.
   - If $\beta \neq 0$, the parties execute a fault localization procedure to find the first incorrect triple by running a binary search on the input triples.
     In each step of the search, the parties run two executions of the above protocol on two half-sized sets of input triples. Then:
     * If the parties output accept in both executions, then the parties output accept and halt.
     * If any execution has ended with the parties holding a pair of conflicting parties (i,j), then the parties output $\mathsf{reject}, (i,j)$ and halt.
     * If $\beta \neq 0$ in both executions, then they continue the search on one of the sets.
     * If $\beta \neq 0$ in one of the two executions, and the parties output accept in the second, then they continue the search on the set for which $\beta \neq 0$.

     If the parties didn't receive any output, then they reach a triple k for which $z_k \neq x_k \cdot y_k$. Then, the parties send their inputs, randomness and view when computing z_k to $\mathcal{F}_{\mathsf{miniMPC}}$ which returns a pair of parties (i,j) with conflicting views. The parties output $\mathsf{reject}, (i,j)$.

Theorem 5.3 *Protocol 5.2 securely computes $\mathcal{F}_{\text{vrfy}}^{\text{full}}$ over a finite field $\mathbb{F}$ with statistical error $\log m \cdot \frac{1}{|\mathbb{F}|}$ in the $(\mathcal{F}_{\text{coin}}, \mathcal{F}_{\mathsf{bc}}, \mathcal{F}_{\text{proveDeg2Rel}}^{\text{cheatIdntfy}}, \mathcal{F}_{\mathsf{miniMPC}})$-hybrid model, in the presence of malicious adversaries controlling up to t parties.*

Proof: Let $\mathcal{S}$ be the ideal world adversary and let $\mathcal{A}$ be the real world adversary. $\mathcal{S}$ is invoked by $\mathcal{F}_{\text{vrfy}}^{\text{full}}$ which sends it all the corrupted parties' shares of $(x_1, y_1, z_1), \ldots, (x_m, y_m, z_m)$ and $\mathsf{out} \in \{\mathsf{reject}, \mathsf{accept}\}$ and $d_k = z_k - x_k \cdot y_k$ for all $k \in [m]$.

In the simulation, $\mathcal{S}$ plays the role of $\mathcal{F}_{\text{coin}}$, thus choosing and handing $\mathcal{A}$ random $\theta_1, \ldots, \theta_m \in \mathbb{F}$. In addition, $\mathcal{S}$ plays the role of $\mathcal{F}_{\text{proveDeg2Rel}}^{\text{cheatIdntfy}}$, $\mathcal{F}_{\mathsf{bc}}$ and $\mathcal{F}_{\mathsf{miniMPC}}$. As in the proof of Theorem 4.3, $\mathcal{S}$

first chooses random shares for the corrupted parties for each ψ^j where P_j is an honest party, and handing these to $\mathcal{A}$. Then, $\mathcal{S}$ receives the honest parties' shares for each ψ^i where P_i is a corrupted party. If the shares dealt by $\mathcal{A}$ are inconsistent, then the consistency check detects it and $\mathcal{A}$ sends the correct shares to $\mathcal{S}$. If $\mathcal{A}$ causes the consistency check for one of the shares distributed by $\mathcal{S}$ to fail, then $\mathcal{S}$ sends the share again to $\mathcal{A}$. Since an honest majority exists, $\mathcal{S}$ can use the honest parties' shares to compute ψ^i for all i for which P_i is corrupted and the corrupted parties' shares. Thus, for each $i \in [n]$, it can simulate $\mathcal{F}^{\text{cheatIdntfy}}_{\text{proveDeg2Rel}}$ handing accept or reject to $\mathcal{A}$ accordingly. If the output is reject for any $i \in [n]$ (which can happen if Eq. (4) does not hold or if $\mathcal{A}$ instructs $\mathcal{F}^{\text{abort}}_{\text{proveDeg2Rel}}$ to output reject), then $\mathcal{A}$ sends an index of a party j to $\mathcal{S}$, which together with the index of the prover i form a pair of disputed parties. Then, $\mathcal{S}$ sends reject and (i, j) to $\mathcal{F}^{\mathsf{full}}_{\mathsf{vrfy}}$, outputs whatever $\mathcal{A}$ outputs and halts.

At this point, if the simulation has not ended with a reject, then it means that all ψ^is are correct. Thus, as in the proof of Theorem 4.3, $\mathcal{S}$ can compute $\beta = \sum_{k=1}^{m} \theta_k \cdot (z_k - x_k \cdot y_k) = \sum_{k=1}^{m} \theta_k \cdot d_k$ and choose random shares for the honest parties, given the value of β and given the corrupted parties' shares (known to $\mathcal{S}$). Then, using these shares, it simulates the execution of the $\mathsf{reconstruct}$ procedure. Consider the following cases:

- If $\mathcal{A}$ sent incorrect shares, causing the opening of β to fail, then $\mathcal{S}$ takes the first pair of parties P_i, P_j for which pairwise-inconsistency occurred (as would the honest parties in the real execution) and sends $\mathsf{reject}, (i, j)$ to $\mathcal{F}^{\mathsf{full}}_{\mathsf{vrfy}}$, outputs whatever $\mathcal{A}$ outputs and halts.

- If $\beta = 0$: if $\mathsf{out} = \mathsf{reject}$ (meaning that the honest parties output accept), then $\mathcal{S}$ outputs fail and halts. If $\mathsf{out} = \mathsf{accept}$, then $\mathcal{A}$ sends accept to $\mathcal{F}^{\mathsf{full}}_{\mathsf{vrfy}}$ outputs whatever $\mathcal{A}$ outputs and halts.

- If $\beta \neq 0$, then the simulation proceeds to the binary search, where $\mathcal{S}$ simulates each step exactly as described so far. If a pair of disputed parties is located, then it is sent to $\mathcal{F}^{\mathsf{full}}_{\mathsf{vrfy}}$. If the honest parties output accept, then $\mathcal{S}$ outputs fail (here it must hold that $\mathsf{out} = \mathsf{reject}$, since otherwise the simulation would have not reach the binary search phase). If the parties found an incorrect triple $x_{\bar{k}}, y_{\bar{k}}, z_{\bar{k}}$ for which $z_{\bar{k}} \neq x_{\bar{k}} \cdot y_{\bar{k}}$ without identifying a disputed pair, then $\mathcal{S}$ asks $\mathcal{F}^{\mathsf{full}}_{\mathsf{vrfy}}$ to find such a pair by sending it $\bar{k}$. Upon receiving (i, j) from $\mathcal{F}^{\mathsf{full}}_{\mathsf{vrfy}}$, $\mathcal{S}$ simulates $\mathcal{F}_{\mathsf{miniMPC}}$ handing (i, j) to $\mathcal{A}$. Finally, $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs. We note that an event where the $\bar{k}$th triple is correct is not possible, because in this case β must equal to 0.

Observe that $\mathcal{A}$'s view consists of (i) t random shares of β^j for each honest party P_j; (ii) message handed by $\mathcal{F}^{\text{abort}}_{\text{proveDeg2Rel}}$ and (iii) the revealed β and (iv) a message from $\mathcal{F}_{\mathsf{miniMPC}}$. The argument for the identical distribution of $\mathcal{A}$'s view in (i) (ii) and (ii) is the same as in the proof of Theorem 4.3. For (iv), since $\mathcal{S}$ receives a pair of parties with conflicting views in the computation of the $\bar{k}$th triple from $\mathcal{F}^{\mathsf{full}}_{\mathsf{vrfy}}$, then it can simulate the role of $\mathcal{F}_{\mathsf{miniMPC}}$ perfectly. Hence, the only difference between the simulation and real execution is the event where $\mathcal{S}$ outputs fail. Note that this happens when $\exists k \in [m] : \ d_k \neq 0$ (which is why $\mathsf{out} = \mathsf{reject}$) but the parties eventually output accept. This occurs when $\beta = 0$ in one of binary search steps. Since there are $\log m$ step, and $\Pr[\beta = 0] = \frac{1}{|\mathbb{F}|}$ in each step, we conclude that $\Pr[\mathsf{fail}] \leq \frac{\log m}{|\mathbb{F}|}$, which is exactly the statistical error allowed by the theorem. ■

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

# of Multiplication Triples (m)	Ring Element sent per Party per Triple				
	$n=3$	$n=5$	$n=7$	$n=9$	$n=11$
2^{15}	$0.6+0.5\|\mathcal{F}_{\mathsf{bc}}\|$	$1.42+2.53\|\mathcal{F}_{\mathsf{bc}}\|$	$4.05+11.8\|\mathcal{F}_{\mathsf{bc}}\|$	$14.7+53\|\mathcal{F}_{\mathsf{bc}}\|$	$60+233\|\mathcal{F}_{\mathsf{bc}}\|$
2^{20}	$0.03+0.02\|\mathcal{F}_{\mathsf{bc}}\|$	$0.07+0.1\|\mathcal{F}_{\mathsf{bc}}\|$	$0.18+0.5\|\mathcal{F}_{\mathsf{bc}}\|$	$0.63+2.21\|\mathcal{F}_{\mathsf{bc}}\|$	$2.52+9.73\|\mathcal{F}_{\mathsf{bc}}\|$
2^{25}	$0.001+0.0008\|\mathcal{F}_{\mathsf{bc}}\|$	$0.003+0.004\|\mathcal{F}_{\mathsf{bc}}\|$	$0.007+0.2\|\mathcal{F}_{\mathsf{bc}}\|$	$0.02+0.08\|\mathcal{F}_{\mathsf{bc}}\|$	$0.1+0.38\|\mathcal{F}_{\mathsf{bc}}\|$
2^{30}	$0.00006+0.00003\|\mathcal{F}_{\mathsf{bc}}\|$	$0.0001+0.0001\|\mathcal{F}_{\mathsf{bc}}\|$	$0.0003+0.0007\|\mathcal{F}_{\mathsf{bc}}\|$	$0.0009+0.003\mathcal{F}_{\mathsf{bc}}\|$	$0.004+0.01\|\mathcal{F}_{\mathsf{bc}}\|$

Table 4: Ring elements sent by each party in the verification of m multiplication triples by n parties. The reported numbers for the *worst case*, i.e., when cheating took place and a semi-corrupted pair was not found before the last step of the Protocol. The numbers are computed via the formula $(Eq.(3)) \cdot d \cdot \frac{1}{m}$. The extension degree d is set as in Table 2. i.e., $d = 46$ to achieve statistical security of 2^{-40}. $|\mathcal{F}_{\mathsf{bc}}|$ is the cost per party of a secure broadcast channel.

Communication cost. Protocol 5.2 is recursive. In jth step of the recursion, the parties secret share one element, reconstruct one element (using $\mathcal{F}_{\mathsf{bc}}$) and call $\mathcal{F}^{\text{cheatIdntfy}}_{\text{proveDeg2Rel}}$ 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 $\mathcal{F}^{\text{cheatIdntfy}}_{\text{proveDeg2Rel}}$, 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. Thus, the cost in the jth step per party is

$$\binom{n}{t} + \binom{n-1}{t} \cdot |\mathcal{F}_{\mathsf{bc}}| + n\log(m/2^j) + 4n \cdot \binom{n-1}{t} \cdot |\mathcal{F}_{\mathsf{bc}}| \quad \text{ring elements.}$$

In the worst case scenario (when a pair of conflicting parties is not identified), there will be $\log m$ steps. Overall, the obtained cost per party is

$$\left(\binom{n}{t} + \binom{n-1}{t} \cdot |\mathcal{F}_{\mathsf{bc}}| + 4n \cdot \binom{n-1}{t} \cdot |\mathcal{F}_{\mathsf{bc}}|\right) \cdot \log m + n \cdot \sum_{j=1}^{\log m} \log(m/2^j)$$

which since

$$\sum_{j=1}^{\log m} \log(m/2^j) = \sum_{j=1}^{\log m} (\log m - j) = \log m \cdot \log m - \frac{\log m}{2}(1+\log m) \leq \frac{\log m \cdot \log m}{2} = \log m \cdot \log\sqrt{m}$$

is roughly

$$\binom{n}{t} \cdot \log m + \binom{n-1}{t} \cdot \log m \cdot 4n \cdot |\mathcal{F}_{\mathsf{bc}}| + n \cdot \log m \cdot \log\sqrt{m} \quad \text{ring elements.} \tag{5}$$

For constant number of parties, the asymptotic cost is roughly $O(\log m \cdot \log\sqrt{m}))$, 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.

In Table 4 we present the concrete communication cost of our verification protocol for different number of multiplication triples m and parties n, as a function of the cost of a secure broadcast channel $\mathcal{F}_{\mathsf{bc}}$. The reported numbers are computed using Eq. (5) and so they represent the cost in

the worst case scenario, i.e., when the triples are incorrect and a pair of disputed parties was not found until the last step of the protocol. It is not surprising that when the m is small, the $\binom{n}{t}$ term becomes dominant as n increases. However, when m is large (e.g., $m \geq 2^{25}$), our verification protocol remains extremely cheap even for 11 parties.

5.2 Two Additional Building Blocks

5.2.1 Computing Authentication Tags

In this section, we 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 $\vec{u}^T = (u_1^T, \ldots, u_L^T)$ and v^T are random secret keys that are shared among the parties using *authenticated* secret sharing (see definition in Section 2). We remark 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:

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

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 $\binom{n}{t}+|\mathcal{F}_{\text{proveDeg2Rel}}^{\text{cheatIdntfy}}|+\binom{n-1}{t}\cdot|\mathcal{F}_{\mathsf{bc}}|$.

5.2.2 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, we are guaranteed that at least one of the possible shares is correct, and that 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. More formally:

Input sharing step. This step begins with the parties holding inputs to the program. At the end of this step, the parties will hold a consistent sharing of the value on each input wire and an authentication tag over the shares held by each subset of $t+1$ parties. In details:

1. The parties set: $\bar{n} = n$ and $\bar{t} = t$.
2. Each part P_i (with $i \in [n]$) verifiable secret shares (vss) its input to the other parties using a $\bar{n}$-out-of-$\bar{t}$ scheme.
3. For each subset T of $\bar{t}+1$ parties, let $x_1^T, \ldots, x_m^T$ be the shares held by T. The parties generate authenticated secret sharing $[\![\vec{u}^T]\!]$ and $[\![v^T]\!]$ and run $\Pi_{auth}(x_1^T, \ldots, x_m^T, [\![\vec{u}^T]\!], [\![v^T]\!])$ to obtain the authentication tag τ^T.
4. If the computation of the tag ended successfully for all subsets, then the parties proceed to the next step. Otherwise, the parties hold a pair of conflicting parties to eliminate[6]. Then, the parties set $\bar{n} = \bar{n} - 2$ and $\bar{t} = t - 1$ go back to step 2.

Computing each segment. The parties divide the program P into segments $P_1, \ldots, P_S$. The computation of each segment P_s (with $s \in [S]$) begins with the parties holding a consistent sharing of the inputs to the segment and an authentication tag over the shares held by each subset of $\bar{t}+1$ parties. Then:

5. The parties compute the segment using Π_{mult} and store the shares of each multiplication instructions' input and outputs.
6. The parties run a pairwise consistency check on all the shares stored in the previous step (with all messages being sent via $\mathcal{F}_{\mathsf{bc}}$ in this check). If the check fails, then the parties identify a pair of conflicting parties to eliminate, run the player elimination and recovery subprotocol and go back to Step 5 with $\bar{n} = \bar{n} - 2$ and $\bar{t} = \bar{t} - 1$.
7. For each subset T of $\bar{t}+1$ parties, the parties run Π_{auth} on the shares held by T on the *output* layer of the segment to obtain an authentication tag τ_s^T.

[6]Throughout the description, whenever we say that the parties hold a pair of parties to eliminate, note that there might be more than one such pair. In this case, the parties choose the pair with the lowest index.

8. If the tag computation has ended successfully for all subsets, then the parties proceed to the next step.
   Otherwise, the parties hold a pair of conflicting parties to eliminate. In this case, the parties run the player elimination and recovery subprotocol and go back to Step 5 with $\bar{n} = \bar{n} - 2$ and $\bar{t} = \bar{t} - 1$.

9. The parties call $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{full}}$ to verify correctness of all multiplication triples generated when computing multiplication instruction.

10. It the output received from $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{full}}$ is accept, then the parties proceed to the next segment.
    Otherwise, the parties receive from $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{full}}$ a pair of parties to eliminate. In this case, the parties run the player elimination and recovery subprotocol and go back to Step 5 with $\bar{n} = \bar{n} - 2$ and $\bar{t} = \bar{t} - 1$.

Output reconstruction. At the beginning of this step, the parties hold consistent shares of each output and an authentication tag over the shares held by each subset of $\bar{t} + 1$ parties. Then:

11. For each *share* held by a subset T of $\bar{t} + 1$ of an output o_k, let P_i be the party that should receive the output. Then, the parties run Π_{auth} on all shares held by T and intended to P_i, to compute an authentication tag τ_i^T with a pair of new authenticated keys.

12. If all tag computations has ended successfully, then the parties proceed to the next step.
    Otherwise, the parties hold a pair of parties to eliminate. Then, they run the player elimination and recovery subprotocol, set $\bar{n} = \bar{n} - 2$ and $\bar{t} = \bar{t} - 1$ and go back to Step 11.

13. For each output o_k, the parties reconstruct the output towards the party P_i who should receive the output: each party sends its shares of o_k to P_i. In addition, the parties reveal the authentication keys used to generate the tag (recall that these are shared using authentication secret sharing and so their reconstruction will always succeed). Then, P_i checks correctness of the shares it received using the authentication keys and tag. Finally, P_i uses the shares that were authenticated successfully to compute the output (recall that since each share is held by at least one honest party, we are guaranteed that one share for each subset will be correct).

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, we have that on average, each instruction will be evaluated $\frac{1}{c \cdot n^2}(t-1) + \frac{1}{c \cdot n^2}(c \cdot n^2 - 1) < 1 + \frac{1}{c \cdot n}$ times (for some constant $c > 1$). Since $c > 1$ and $n > 2$, the average number of repetitions per instruction is approximately 1.

Theorem 5.4 *Let R be a finite field or the ring $\mathbb{Z}_{2^k}$ and let f be a n-party functionality represented by a straight-line program P over R. Then, if Π_{mult} is a replicated private multiplication protocol as defined in Section 2.4, then our main protocol, as described in the text, securely computes f in the $(\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{full}}, \mathcal{F}_{\text{proveDeg2Rel}}^{\text{cheatIdntfy}}, \mathcal{F}_{\mathsf{bc}})$-hybrid model in the presence of malicious adversaries controlling up to t parties.*

The proof appears in Appendix B.

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 $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{full}}$ once. The asymptotic cost of $\mathcal{F}_{\mathsf{vrfy}}^{\mathsf{full}}$ 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 given that n is assumed to be constant, we conclude that 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 $W \leq S$. Moreover, in most cases, when the program is large, W will be much smaller, and so $O(W)$ can be ignored.

Optimistic computation. It is possible to combine our two protocols and run first the secure-with-abort protocol, and then, only if cheating took place (causing the parties to abort), run the fully secure protocol. This has the advantage of avoiding the use of the authentication mechanism and expensive broadcast channels (though they are used only $\log S$ times in our protocol) in case the parties behave honestly.

5.4 Optimizations for the Case Π_{mult} is Instantiated with [DN07]

5.4.1 Removing the $O(W)$ term

If we instantiate Π_{mult} with the DN protocol [DN07], then as explained in Section 2.4, 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. The claim that this party is honest follows from the fact that we overall eliminated t pairs of parties, with the property that one of them must be corrupted. Since there are t corrupted parties, it follows that the remaining party is honest. In this case, following the 3-party construction of [BGIN19], we can treat this party 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, note that 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. Thus, the overall cost of our entire protocol when using the DN multiplication protocol, is $1.5 \cdot S + o(S)$, with no dependency on the width of the circuit.

5.4.2 Pushing the $\binom{n}{t}$-computational overhead to a "silent" pre-processing

Our fully secure protocol relies on replicated secret sharing, for which the size of the share held by each party is $\binom{n-1}{t}$. This affects both computation and communication. Specifically, each time the parties locally compute $[\![x]\!] \cdot [\![y]\!]$ in the private multiplication protocol, this is translated to carrying-out roughly $\binom{n-1}{t}^2$ local multiplication operations. Communication, on the other hand, is affected by the growth of the share's size only at the layers between segments and in the verification protocol, where sublinear number of sharing and reconstruction operations are taking place.

It is possible to move the computational overhead to an offline phase by using local conversions from replicated secret sharing to Shamir [CDI05]. Specifically, the parties can locally convert each $[\![r]\!]$ from replicated sharing form to a Shamir form and run all executions of Π_{mult} with Shamir shares. At the same time, the parties can store the output of all multiplication instructions also in a replicated form, by taking $[\![r]\!] + xy - r$ with the replicated version of $[\![r]\!]$ and use these in the verification protocol. By doing this, we avoid the $\binom{n}{t}$ computational overhead when computing the program (now $[\![x]\!] \cdot [\![y]\!]$ is translated into a single operation per party), at the expense of running expensive conversions of random sharings from replicated form to Shamir, which can be done in an offline input-independent phase.

Acknowledgments

E. Boyle supported by ISF grant 1861/16, AFOSR Award FA9550-17-1-0069, and ERC Project HSS (852952). N. Gilboa supported by ISF grant 2951/20, ERC grant 876110, and a grant by the BGU Cyber Center. Y. Ishai supported by ERC Project NTSC (742754), NSF-BSF grant 2015782, BSF grant 2018393, and ISF grant 2774/20. A. Nof supported by ERC Project NTSC (742754).

References

[AFL+16] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 805–817, 2016.

[BBC+19] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Zero-knowledge proofs on secret-shared data via fully linear pcps. In *Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part III*, pages 67–97, 2019. Full version: ePrint report 2019/188.

[BFO12] Eli Ben-Sasson, Serge Fehr, and Rafail Ostrovsky. Near-linear unconditionally-secure multiparty computation with a dishonest minority. In *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, pages 663–680, 2012.

[BGIN19] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. Practical fully secure three-party computation via sublinear distributed zero-knowledge proofs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 869–886, 2019.

[BGIN20] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. Efficient fully secure computation via distributed zero-knowledge proofs. In *Asiacrypt*, 2020.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 1–10, 1988.

[BHPS19] Megha Byali, Carmit Hazay, Arpita Patra, and Swati Singla. Fast actively secure five-party computation with security beyond abort. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 1573–1590, 2019.

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

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 11–19, 1988.

[CDI05] Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In *Theory of Cryptography, Second Theory of Cryptography Conference, TCC 2005, Cambridge, MA, USA, February 10-12, 2005, Proceedings*, pages 342–362, 2005.

[CGH+18] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority MPC for malicious adversaries. In *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Proceedings, Part III*, pages 34–64, 2018.

[Cle90] Richard Cleve. Towards optimal simulations of formulas by bounded-width programs. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA*, pages 271–277, 1990.

[DN07] Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In *Advances in Cryptology - CRYPTO 2007, 27th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2007, Proceedings*, pages 572–590, 2007.

[DS83] Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM J. Comput.*, 12(4):656–666, 1983.

[FL19] Jun Furukawa and Yehuda Lindell. Two-thirds honest-majority MPC for malicious adversaries at almost the cost of semi-honest. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 1557–1571, 2019.

[FS86] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, pages 186–194, 1986.

[FY19] Serge Fehr and Chen Yuan. Towards optimal robust secret sharing with security against a rushing adversary. In *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part III*, pages 472–499, 2019.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 169–178, 2009.

[GI99] Niv Gilboa and Yuval Ishai. Compressing cryptographic resources. In *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, pages 591–608, 1999.

[GIP+14] Daniel Genkin, Yuval Ishai, Manoj Prabhakaran, Amit Sahai, and Eran Tromer. Circuits resilient to additive attacks with applications to secure computation. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 495–504, 2014.

[GL05] Shafi Goldwasser and Yehuda Lindell. Secure multi-party computation without agreement. *J. Cryptology*, 18(3):247–287, 2005.

[GLS19] Vipul Goyal, Yanyi Liu, and Yifan Song. Communication-efficient unconditional MPC with guaranteed output delivery. In *Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part II*, pages 85–114, 2019.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 218–229, 1987.

[Gol04] Oded Goldreich. *The Foundations of Cryptography - Volume 2, Basic Applications*. Cambridge University Press, 2004.

[GRR98] Rosario Gennaro, Michael O. Rabin, and Tal Rabin. Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC '98, Puerto Vallarta, Mexico, June 28 - July 2, 1998*, pages 101–111, 1998.

[GRW18] S. Dov Gordon, Samuel Ranellucci, and Xiao Wang. Secure computation with low communication from cross-checking. In Thomas Peyrin and Steven D. Galbraith, editors, *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part III*, pages 59–85, 2018.

[GS20] Vipul Goyal and Yifan Song. Malicious security comes free in honest-majority MPC. *IACR Cryptol. ePrint Arch.*, 2020:134, 2020.

[GSZ20] Vipul Goyal, Yifan Song, and Chenzhi Zhu. Guaranteed output delivery comes free in honest majority MPC. In *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part II*, pages 618–646, 2020.

[HMP00] Martin Hirt, Ueli M. Maurer, and Bartosz Przydatek. Efficient secure multi-party computation. In *ASIACRYPT 2000*, pages 143–161, 2000.

[IKKP15] Yuval Ishai, Ranjit Kumaresan, Eyal Kushilevitz, and Anat Paskin-Cherniavsky. Secure computation with minimal interaction, revisited. In *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*, pages 359–378, 2015.

[IKP+16] Yuval Ishai, Eyal Kushilevitz, Manoj Prabhakaran, Amit Sahai, and Ching-Hua Yu. Secure protocol transformations. In *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference*, pages 430–458, 2016.

[ISN89] Mitsuru Ito, Akira Saito, and Takao Nishizeki. Secret sharing scheme realizing general access structure. *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)*, 72(9):56–64, 1989.

[KKW18] Jonathan Katz, Vladimir Kolesnikov, and Xiao Wang. Improved non-interactive zero knowledge with applications to post-quantum signatures. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 525–537, 2018.

[LN17] Yehuda Lindell and Ariel Nof. A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 259–276, 2017.

[Mau06] Ueli M. Maurer. Secure multi-party computation made simple. *Discrete Applied Mathematics*, 154(2):370–381, 2006.

[MRZ15] Payman Mohassel, Mike Rosulek, and Ye Zhang. Fast and secure three-party computation: The garbled circuit approach. In *ACM CCS*, pages 591–602, 2015.

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

[PR18] Arpita Patra and Divya Ravi. On the exact round complexity of secure three-party computation. In *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part II*, pages 425–458, 2018.

[PSL80] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

[RAD+78] Ronald L Rivest, Len Adleman, Michael L Dertouzos, et al. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.

[RB89] Tal Rabin and Michael Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority (extended abstract). In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washigton, USA*, pages 73–85, 1989.

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

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986*, pages 162–167, 1986.

A Proof of Security for the Security-with-Abort Main Protocol

Theorem A.1 *Let R be a finite field or the ring $\mathbb{Z}_{2^k}$ and let f be a n-party functionality represented by a straight-line program over R. Then, if Π_{mult} is a private multiplication protocol as defined in Section 2.4, then protocol Π_f, as described in the text, securely computes f with abort in the $\mathcal{F}_{\text{vrfy}}^{\text{abort}}$-hybrid model in the presence of malicious adversaries controlling up to t parties.*

Proof: We construct a simulator $\mathcal{S}$ which interacts with the real world adversary $\mathcal{A}$ as follows. $\mathcal{S}$ secret shares the input 0 as the input of the honest parties, and hands $\mathcal{A}$ the corrupted parties' shares. Upon receiving the shares of the honest parties from $\mathcal{A}$, it uses them to compute and extract the inputs (this is possible because an honest majority exists). If the shares received from $\mathcal{A}$ are not consistent, then $\mathcal{S}$ sends abort to the trusted party computing f (this is detected by the properties of VSS).

Then, $\mathcal{S}$ emulates the program computation by playing the role of the honest parties in Π_{mult}. At the end of this step, $\mathcal{S}$ holds all the shares of the output and inputs of each instruction (since the honest parties' shares alone determine both the secret and the corrupted parties' shares) and so it knows whether $z_k = x_k \cdot y_k$ for each multiplication instruction k or not (which means that cheating took place). If the shares are not consistent, then $\mathcal{S}$ simulates the honest parties aborting the protocol in the consistency check. Otherwise, given this information, $\mathcal{S}$ can play the role of $\mathcal{F}_{\text{vrfy}}^{\text{abort}}$. If cheating took place, then $\mathcal{S}$ sends reject to $\mathcal{A}$, simulates the honest parties aborting in the real world and sends abort to the trusted party computing f. If cheating did not take place, then $\mathcal{S}$ sends accept to $\mathcal{A}$ and waits to receive back $\mathsf{out} \in \{\mathsf{accept}, \mathsf{reject}\}$. If $\mathsf{out} = \mathsf{reject}$, then as before, $\mathcal{S}$ simulates an abort and halts. Otherwise, if $\mathsf{out} = \mathsf{accept}$, then it proceeds to the next step.

Finally, $\mathcal{S}$ sends the corrupted parties' inputs to the trusted party to receive back their outputs. Then, it simulates the output reconstruction. For every output intended to a corrupted party, it chooses shares for the honest parties, given the output and the corrupted parties' shares and sends them to $\mathcal{A}$. For each output intended to an honest party P_j, it receives the corrupted parties' shares sent from $\mathcal{A}$ to the honest party. If $\mathcal{A}$ sent incorrect shares, then $\mathcal{S}$ sends abort_j to the trusted party. Otherwise, it sends $\mathsf{continue}_j$. Finally, $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs.

Observe that the difference between the simulation and real world execution is that in the simulation $\mathcal{S}$ uses incorrect inputs for the honest parties. However, by the secrecy of the secret sharing scheme, this does not make any difference in the input sharing step and by the privacy property of Π_{mult} this does not make any difference in the program emulation step. Finally, given that the $\mathcal{A}$'s view until the last step in distributed the same in both executions, it follows that the view in the output reconstruction is also the same, since $\mathcal{A}$ only sees random shares that are chosen under the constraint that they will open to its output. This concludes the proof. ■

B Proof of Theorem 5.4

Proof: Let $\mathcal{S}$ be the ideal world simulator and $\mathcal{A}$ be the real world adversary. We describe briefly how $\mathcal{S}$ works in each step of the protocol:

- Input sharing step: In this step, $\mathcal{S}$ uses 0 as the honest parties' inputs and hands the corrupted parties' shares to $\mathcal{A}$. Upon receiving the honest parties' shares of the corrupted parties' inputs from $\mathcal{A}$, simulator $\mathcal{S}$ uses them to extract the inputs and compute the remaining shares held by

the corrupted parties. Note that in this step, $\mathcal{A}$ sees random t shares of inputs that could open to any value in both the simulated and real world execution and thus the view of $\mathcal{A}$ is identically distributed in both executions.

- Generating authentication tags: In this step, $\mathcal{S}$ needs to simulate $\mathcal{F}^{\text{cheatIdntfy}}_{\text{proveDeg2Rel}}$, a secret sharing of additive shares of the tag and an opening of the tag. For simulating $\mathcal{F}^{\text{cheatIdntfy}}_{\text{proveDeg2Rel}}$, simulator $\mathcal{S}$ uses its knowledge of all shares held by the corrupted parties to compute the secret that should be shared by each corrupted party. Then, it uses the shares that $\mathcal{A}$ sent to reconstruct the secret and compare it to what should have been sent and send accept or reject to $\mathcal{A}$ accordingly. In case of reject, it receives from $\mathcal{A}$ an index of another party to eliminate together with the prover. If the output is accept, the simulation proceed to the exposure of the tag. If the opening of the tag fails, then the first pair with pair-wise inconsistency is chosen to be eliminate. Since the tag is uniformly distributed in both the simulation and the real execution, then so is $\mathcal{A}$'s view in both executions.

- Player elimination and recovery: Given a pair of parties (P_i, P_j) to eliminate, $\mathcal{S}$ needs to simulate the publication of shares that are held by both parties. Since each such share is held by at least one honest party, $\mathcal{S}$ simulates the broadcast of the share by that party. Then, it simulates the opening of the authentication keys by playing the role of the honest parties.

- Segment computation: For each segment, simulator $\mathcal{S}$ plays the role of the honest parties in the execution of Π_{mult}. For each output wire of a multiplication gate, it computes the secret on the wire (using the honest parties' shares). If the shares on the output wires are not consistent (and so do not define any value), then the honest parties, played by $\mathcal{S}$, detect it in the consistency check. In this case, the simulation outputs a pair of conflicting parties and $\mathcal{S}$ simulates the player and elimination recovery step as described above. Otherwise, if the shares are consistent, then for each gate, where the output is not correct (given the inputs), $\mathcal{S}$ computes the difference between the values and records the view of the honest parties during the computation. Then, it simulates the tag computation over the output layer of the segment as described above. Finally, it simulates $\mathcal{F}^{\mathsf{full}}_{\mathsf{vrfy}}$. If cheating took place, then it sends reject and all d_k for which $d_k \neq 0$ to $\mathcal{A}$. Then, it waits to A to either send a pair of parties to eliminate (where one is corrupted) or ask $\mathcal{F}^{\mathsf{full}}_{\mathsf{vrfy}}$ to find such a pair. In the latter, $\mathcal{S}$ finds a pair of parties to eliminate by computing the messages that should have been sent by each corrupted party and compare it to what was received by each honest party. Then, $\mathcal{S}$ sends the obtained pair to $\mathcal{A}$. In this case, the simulation proceed by executing the player elimination and recovery protocol as described above, and recompute the segment. If no cheating took place, then $\mathcal{S}$ hands $\mathcal{A}$ the output accept. In this case, $\mathcal{A}$ can decide to change the output, but then a pair of parties is eliminated as above. Otherwise, the simulation proceed to the next segment.

- Output reconstruction: First, $\mathcal{S}$ sends the inputs of the corrupted parties to the trusted party computing f to receive the corrupted parties' outputs. Then, it replaces the shares held by a subset of only honest parties by a new share that will allow opening to the received output. Then, $\mathcal{S}$ simulates the tag authentication for the shares on each wire as described above. Once this step ends successfully, $\mathcal{S}$ simulates the reconstruction of the outputs, which includes sending shares and revealing the authentication keys.

The difference between the simulated and real execution is that in the simulation, the simulator $\mathcal{S}$ uses incorrect inputs. However, this does not make change in Π_{mult} by the privacy property, and

the remaining messages $\mathcal{A}$ sees during the execution, consist of random shares that could open to any value or opening of authentication tags for shares that already known to him. Thus, $\mathcal{A}$'s view is distributed the same in both executions. Finally, given that up to the reconstruction step $\mathcal{A}$'s view is identically distributed in both executions, the simulation in this final step is perfect. This concludes the proof. ■