One Proof to Rule Them All: Practical, Sublinear Verification for Actively Secure MPC on $\mathbb{Z}_{2^k}$ with Dishonest Majority and a Dealer (Full Version)*

Andreas Brüggemann
brueggemann@encrypto.cs.tu-darmstadt.de
Technical University of Darmstadt
Darmstadt, Germany

Ariel Nof
ariel.nof@biu.ac.il
Bar-Ilan University
Ramat-Gan, Israel

Thomas Schneider
schneider@encrypto.cs.tu-darmstadt.de
Technical University of Darmstadt
Darmstadt, Germany

Abstract

Towards bridging the gap between passively and actively secure multiparty computation (MPC), the use of sublinear distributed zero-knowledge (DZK) proofs gained popularity. Such proofs enable extending a passively secure protocol by adding a verification step whose communication is *sublinear* in the circuit size. For arbitrarily many parties and a dishonest majority, adding a trusted dealer enables efficient computation, as recently shown by Asterisk (IEEE S&P'24) without requiring DZK. This setting is also compatible with DZK, as shown by Boyle et al. (CRYPTO'21). Unfortunately, their approach is not tailored to computation over a ring $\mathbb{Z}_{2^k}$, often favored for concrete efficiency and practicality, resulting in high computational overhead. In the honest majority setting with few parties, Li et al. (CCS'24) optimized DZK to rings, achieving significant performance improvements.

In this work, we propose the first sublinear verification protocol that is both, designed for the n-party dishonest majority setting with a dealer, and tailored to computation over a ring $\mathbb{Z}_{2^k}$, combining and improving upon both approaches above. Previous approaches used n DZK proofs to check correct behavior for each individual party. Instead, we show how to verify in a single, novel DZK proof that all parties *together* behave correctly. This decreases the communication complexity for verification from $O(n \cdot \log m)$ to $O(n+\log m)$ ring elements per party for m multiplications. Hence, for the first time, active security using DZK scales well with the number of parties n. We provide the first public implementation for DZK with arbitrary n and a dealer and show its practical efficiency. For $m = 10^6$ multiplications across 30 layers, communication increases by only 0.7% over the passively secure base protocol with only moderate computation overhead. This becomes especially useful in a WAN setting, where we achieve active security at only 34% run time overhead over the passive variant. Compared to Asterisk (IEEE S&P'24), our protocol has 1.8× better communication and improves run time by 2.5× in WAN and 11.9× in LAN.

Keywords

Secure multiparty computation; distributed zero knowledge proofs

1 Introduction

The rapid growth of automated collection and processing of private information raises substantial privacy and security concerns. As a current example, large language models are trained on enormous data sets, often while disregarding privacy concerns. Yet, utilizing private, distributed data is possible *without* disclosure of private information by using modern cryptographic tools such as secure multiparty computation (MPC). There already exists a diverse toolkit of highly-efficient MPC protocols, e.g., [2, 3, 20], that protect against an adversary that is *passive*, i.e., tries to derive additional information from its messages seen without actively trying to change the computation, and that is only able to corrupt a minority of parties (called the *honest-majority* setting). A more challenging problem is to achieve good efficiency when the adversary is even *active*, i.e., it can behave arbitrarily and can also corrupt an arbitrary subset of parties (called the *dishonest-majority* setting). This is highly relevant as assuming the adversary to stay passive is a strong assumption. Similarly, an honest majority is a rather strong assumption, especially in settings with only a few parties.

For active adversaries and a dishonest majority, the most prominent protocol is SPDZ [26] with many variants [6, 22, 24, 38] based on that approach. There, the additional protection against an active adversary comes at a cost that is linear in the size of the function to be computed, leaving a large gap in costs between passively and actively secure MPC. For the honest majority setting, the use of distributed zero-knowledge (DZK) proofs based on a primitive called *Zero-Knowledge Fully Linear Interactive Oracle Proof* (zk-FLIOP) has emerged as an option to achieve active security at zero amortized communication overhead over passively secure MPC protocols [7, 10, 11]. This was then translated to a dishonest majority in [12].

In our work, we investigate the efficient use of DZK for an $(n+1)$-party setting which extends the active security, dishonest majority setting by a non-colluding, trusted dealer that provides the other parties with correlated randomness. Using such a dealer has two key advantages: First, it eliminates the need for expensive asymmetric cryptography and relying on their hardness assumptions. Second, it enables a communication cost per party and multiplication which is constant instead of linear in the number of parties n, yielding better scalability. Our dealer is a true dealer and not just a helper: It can provide correlated randomness for many computations during the *offline phase* and then leave. Later, once the other parties know their actual protocol inputs, they can execute the highly efficient *online phase* entirely without the dealer. For instance, this dealer could be an additional impartial institution providing correlated randomness, similar to [9, 38]. For the approach of SPDZ [26], Asterisk [38] has recently shown that using such a dealer yields communication improvements of multiple orders of magnitude compared to not

*Please cite the conference version of this paper [16], published at CCS 2026.

using a dealer. Finally, as demonstrated in [12], the dealer can later also be collectively emulated by the parties using a generic MPC protocol. As MPC protocols with a distributed dealer have not yet been implemented, the practicality of this approach needs to be investigated further and we provide directions in §D.

MPC over Rings. Many applications of MPC require to compute linear operations in the form of additions and multiplications. Because multiplication is relatively large when expressed as a Boolean circuit, arithmetic computation is usually done with multi-round protocols based on arithmetic linear secret sharing. Possible MPC applications that heavily use arithmetic operations include privacy-preserving machine learning (e.g., linear regression, logistic regression, support vector machines, neural networks, graph neural networks, and transformers), privacy-preserving biometric identification, graph algorithms, and many more. Further applications are given in mixed-protocol MPC frameworks like [13, 28, 31, 36, 49].

While there are many MPC protocols that are designed to work over a field $\mathbb{F}$, e.g., [6, 25, 26, 33, 38], computation over a ring $\mathbb{Z}_{2^k}$ is a much more natural choice for applications and users of MPC. The ring $\mathbb{Z}_{2^k}$ for $k = 32$ corresponds to a 32-bit integer, which is available in essentially all high-level programming languages. This gives users a natural interface and datatype for using MPC and is supported by many MPC compilers that compile from a high-level or domain-specific language into an arithmetic or mixed boolean/arithmetic circuit over rings, e.g., [13, 14, 18, 21, 29, 30, 35, 40, 43, 51]. Additionally, a ring allows for natural MPC protocols for bit decomposition to achieve mixed-mode MPC protocols that combine arithmetic circuits for linear operations and Boolean circuits for non-linear operations [13, 28, 49], e.g., for privacy-preserving machine learning applications. Finally, as rings of sizes $k \in \{16, 32, 64, \dots\}$ also have corresponding CPU instructions, the concrete computation complexity of such operations is much better than when computing over a field [22].

While MPC over rings has many benefits, it is often more challenging to achieve. Some actively secure protocols for a dishonest majority [22, 24] work over a ring that needs to be extended by statistical security parameter σ many bits. For $k = 32$ and $\sigma = 40$, these protocols work over a ring with at least $k + \sigma = 72$ bits, which more than doubles the concrete online communication. Using DZK [12] enables active security at negligible communication overhead, but works over a *Galois ring extension*, resulting in substantial computational overhead. In [46], it was shown how a novel zk-FLIOP construction specifically designed for rings enables significantly faster computation; however, it operates in an honest majority setting with a low number of parties. Our protocol for the first time achieves the best of both worlds in the $(n+1)$-party setting: concretely efficient communication *and* computation.

1.1 Our Contributions

Active Security for MPC on $\mathbb{Z}_{2^k}$ at a Low Concrete Overhead. For the dishonest majority $(n+1)$-party setting (n parties and a trusted dealer), we provide the first sublinear distributed zero-knowledge proof which is specifically designed for rings $\mathbb{Z}_{2^k}$. In contrast, prior works are natively for fields and their extension to a ring $\mathbb{Z}_{2^k}$ has high overhead, especially in computation. By extending a passively secure protocol using our proof, we provide an actively secure MPC protocol at an additional communication cost logarithmic in the size of the evaluated arithmetic circuit. Our proof combines ideas from the approaches of [12] (which shares our setting, but is designed primarily for fields) and [46] (which is designed for $\mathbb{Z}_{2^k}$, but restricted to an honest majority setting with few parties). In contrast to [12], we use slightly larger rings instead of computationally expensive interpolation on a Galois ring extension. We also provide support for securely evaluating dot products (§F). Furthermore, we make the following improvements:
New Optimizations for Distributed Zero-Knowledge (DZK). In our proof, all parties except for the dealer act simultaneously as prover and verifier within a single, joint proof, while the dealer acts as an additional verifier. This contrasts prior approaches [7, 12, 46], where each party acts as a prover in a separate proof, which resulted in the total complexity per party increasing linearly in the number of parties n. For proving statements of size m, we decrease the total proving complexity from $O(n \cdot \log m)$ to $O(\log m)$ ring elements plus low $2n\kappa$ bits per party, where κ is the computational security parameter. By doing so, we for the first time achieve good scalability to many parties. Furthermore, we utilize PRF-based optimizations and prove their security, which also translates to [46] who omitted this due to lack of security arguments that we provide in §E.1.
First Implementation of DZK-based MPC for Arbitrarily Many Parties. We provide a full implementation of our protocol for $\mathbb{Z}_{2^k}$, including our novel DZK proof at https://encrypto.de/code/TheOneProof. To the best of our knowledge, this is the first implementation of DZK on $\mathbb{Z}_{2^k}$ in a dishonest majority setting, as well as the first implementation of DZK for an arbitrary number of parties.
Extensive Evaluation and Comparison. Using our implementation, we benchmark our protocol for varying numbers of parties n, circuit sizes m, and network settings. For $m = 1$ million multiplications across 30 layers and $n = 3$ parties, our protocol requires less than 5.2 MiB communication per party, taking 0.6 s in a LAN and 6.2 s in a WAN setting. On top of the passively secure protocol, our verification to achieve active security adds less than 1% more communication and run time overheads of only 148% in LAN and 34% in WAN. We also compare to Asterisk [38], which provides active security over fields in the $(n+1)$-setting based on non-DZK verification, improving both offline and online. Asterisk's bottleneck is its offline phase, where we improve communication by more than 3× and run time by at least 29× in LAN and 9× in WAN. We also improve online performance. As for DZK-based works, no implementation is publicly available. Hence, we compare the estimated communication costs and discuss the expected computational bottleneck, establishing our approach as the new state of the art.

1.2 Comparison to Related Work

Passively Secure Baseline. For passively secure computation in a dishonest majority, the GWM-protocol [32] remains a popular choice. These days, it is common to compute secret shared Beaver triples [4] in an input- and function-independent offline phase, i.e., triples (a, b, c) where $c = a \cdot b$. These triples can be computed, e.g., using oblivious transfer (OT) [28] or silently by pseudorandom correlation generators (PCGs) [8], or simply be distributed by a trusted dealer [37]. In the online phase, once the parties' inputs are known, a circuit is evaluated, consuming one triple for each multiplication. Using *masked secret sharing*, ASTRA [20] improves upon

Table 1: Overview and comparison with related works. All works are in the $(n+1)$-party dishonest majority setting and use function-dependent preprocessing (some works originally use function-independent preprocessing, but are rewritten here for function-dependent preprocessing). Parameters: finite field $\mathbb{F}$, ring $\mathbb{Z}_{2^k}$ of bitlength k, statistical security parameter σ, and m multiplication gates. Asymptotics are given w.r.t. the number of parties n and multiplications m.

approach	domain	amortized communication per multiplication ($\times m$)		additional, sublinear (in m) communication per party (offline + online)	verification rounds	computation per party
		offline (from $\mathcal{D}$)	online (per party)			
Passively Secure Baseline	$\mathbb{F}$	$1 \times \mathbb{F}$	$2 \times \mathbb{F}$	—	—	$O(m) \times \mathbb{F}$
e.g. [20]	$\mathbb{Z}_{2^k}$	$1 \times \mathbb{Z}_{2^k}$	$2 \times \mathbb{Z}_{2^k}$	—	—	$O(m) \times \mathbb{Z}_{2^k}$
SPDZ Approach with MACs	$\mathbb{F}$	$3 \times \mathbb{F}$	$2 \times \mathbb{F}$	$O(n) \times \mathbb{F}$	$O(1)$	$O(m) \times \mathbb{F}$
e.g. [22, 26, 38, 50]	$\mathbb{Z}_{2^k}$	$3 \times \mathbb{Z}_{2^{k+\sigma}}$	$2 \times \mathbb{Z}_{2^{k+\sigma}}$	$O(n) \times \mathbb{Z}_{2^{k+\sigma}}$	$O(1)$	$O(m) \times \mathbb{Z}_{2^{k+\sigma}}$
DZK Proofs Approach	$\mathbb{F}$	$1 \times \mathbb{F}$	$2 \times \mathbb{F}$	$O(n \cdot \log m) \times \mathbb{F}$	$O(\log m)$	$O(n \cdot m) \times \mathbb{F}$
[9, 12]	$\mathbb{Z}_{2^k}$	$1 \times \mathbb{Z}_{2^k}$	$2 \times \mathbb{Z}_{2^k}$	$O(n \cdot \log m) \times (\mathbb{Z}_{2^k})^{\geq \sigma}$	$O(\log m)$	$O(n \cdot m) \times (\mathbb{Z}_{2^k})^{\geq \sigma}$
Ours	$\mathbb{Z}_{2^k}$	$1 \times \mathbb{Z}_{2^k}$	$2 \times \mathbb{Z}_{2^k}$	$O(n + \log m) \times \mathbb{Z}_{2^{k+O(\log m)}}$	$O(\log m)$	$O(n+m) \times \mathbb{Z}_{2^{k+O(\log m)}}$

that using a *function-dependent* preprocessing in the $(2+1)$-setting, i.e., with two parties and a trusted dealer. This approach enables to cut online communication cost in half, which is why we focus on the function-dependent case in this paper. It generalizes to an arbitrary $(n+1)$-setting as can be seen, e.g., in [38] when leaving out the parts required to reach active security. The resulting protocol serves as a passively secure baseline protocol in our work, to which we compare our actively secure protocol to determine the overhead for achieving active security. Tab. 1 provides the complexity for this protocol on domains $\mathbb{F}$ or $\mathbb{Z}_{2^k}$.

For reaching active security, works in the n or $(n+1)$-setting follow the same blueprint: The prior passively secure protocol is executed optimistically and an additional verification step is added at the end of the computation, just before any output is revealed. The primary two options for this verification are as follows:

SPDZ Approach with MACs. SPDZ [26] provides a method to compute on *authenticated shares* over a field $\mathbb{F}$ by using MAC values. For each multiplication triple (a, b, c), SPDZ additionally uses the triple of MACs $\Delta \cdot a, \Delta \cdot b, \Delta \cdot c$, where Δ is kept secret. This type of correlated randomness yields an efficient subprotocol to verify correctness, with a constant number of rounds and communication cost that is independent of the circuit size. However, for a circuit with m multiplication gates, the correlated randomness size is increased by $O(m \cdot \sigma)$, where σ is the statistical security parameter. Assisted MPC [50] has extended SPDZ, originally without a dealer, to the $(n+1)$-setting. While these works use function-independent preprocessing, they can be restructured into function-dependent preprocessing protocols as shown by Asterisk [38] (with a dealer[1]) and Turbospeedz [6] (without a dealer). This leads to an amortized communication overhead per multiplication over the passively secure baseline by 3× offline, while the online phase only has a communication overhead independent of the circuit, as shown in Tab. 1. We benchmark Asterisk against our protocol in §5.2.8.

To profit from the benefits of computations over a ring $\mathbb{Z}_{2^k}$, an extension of the SPDZ approach is needed, known as the *SPDZ$\mathbb{Z}_{2^k}$-trick* [22]. This requires to lift computation to a larger ring $\mathbb{Z}_{2^{k+\sigma}}$ for statistical security parameter σ to ensure its correctness modulo 2^k. This introduces an additional overhead of σ bits per value, but otherwise leaves the costs unchanged. While [38] is for function-dependent preprocessing in the $(n+1)$-setting over a field $\mathbb{F}$, we can combine it with the SPDZ$\mathbb{Z}_{2^k}$-trick to obtain the costs for SPDZ-style active security for a ring $\mathbb{Z}_{2^k}$ shown in Tab. 1.

Distributed Zero-Knowledge (DZK) Proofs. Alternatively, sublinear distributed zero-knowledge (DZK) proofs, introduced by Boneh et al. [7], can be used to go from passive to active security at zero amortized cost per multiplication. These proofs rely on an abstract primitive called *Zero-Knowledge Fully Linear Interactive Oracle Proofs* (zk-FLIOP). In the honest majority setting, this approach was used to close the gap between passive and active security by several works [10, 11, 34]. Boyle et al. [12] then showed how to extend it to the dishonest majority setting with a dealer. The main strength of this technique in the dishonest majority setting, as introduced by [12], is that the dealer does not need to authenticate each triple separately, but instead only provides a batched authentication that is used in the DZK-based verification protocol. Hence, in contrast to using the SPDZ approach, the offline communication from the dealer *per multiplication* remains the same as in the passively secure baseline (see Tab. 1). DZK-based verification over $\mathbb{F}_2$ in the $(2+1)$-setting was recently implemented and shown to be practical in [9].

In the previous DZK-based works, the verification complexity scales logarithmically with the number of multiplications m, and linearly in the number of parties n, because one zk-FLIOP is used per party. The verification's round complexity is logarithmic, too, compared to the constant round complexity for the SPDZ approach.[2]

When used over rings, the aforementioned works do not require using a larger ring during circuit evaluation, but only for the amortized verification part. Hence, the online cost per multiplication

[1] Asterisk [38] actually uses a helper that participates also in the online phase for the final verification to achieve fairness. Yet, a dealer version for security with abort can be obtained, e.g., using techniques from [26].

[2] There also exists a constant-round variant at the cost of higher communication. We do not consider that here as we focus primarily on minimizing communication volume.

matches the passively secure case in contrast to using the SPD$\mathbb{Z}_{2^k}$-trick. Although very promising, this approach has a significant drawback. As [12] relies on polynomial interpolation, translating it to a ring requires using a Galois ring extension for verification, which increases the size of the number representation by more than $\sigma\times$ for statistical security parameter σ. Moreover, both polynomial interpolation and computation on this extension ring are serious computational bottlenecks.

As summarized in Tab. 1, our protocol has the same amortized communication cost per multiplication gate as the previous DZK approach, but we achieve lower overall communication and, in contrast to [12], better scalability for many parties n. We compare concrete communication costs in §5.1. Moreover, we reduce the computation costs over the prior approach by not relying on Galois extension rings. We discuss the expected resulting savings in §5.2.7.

In the honest majority setting with few parties, [46] provides a zk-FLIOP construction specifically designed for computation on $\mathbb{Z}_{2^k}$, yielding significant performance improvements compared to using Galois ring extensions. While originally for another setting, their construction serves as a starting point for our novel verification method, alongside techniques used by [12] to enable the use of DZK in our $(n+1)$-setting. In contrast to [12, 46], we for the first time use a single joint proof, instead of one proof per party, thereby achieving good scalability to many parties. Recently, [44] proposed further optimizations that use computation over both $\mathbb{Z}_{2^k}$ and prime fields in an honest majority setting for three parties. Translating these optimizations to our setting may be possible and further improve performance, and we leave a thorough investigation of that to future work.

Orthogonal Directions. For MPC protocols with a trusted dealer, this dealer may be replaced by trusted, local hardware modules as demonstrated, e.g., in [27, 47]. In this work, we consider arithmetic secret sharing requiring multi-round protocols. There are also alternative constant-round approaches, such as arithmetic garbling [1, 48], but they require more communication. Mixed-mode MPC protocols, e.g., [28, 31, 42], allow to combine arithmetic and binary computation for practical applications, using different sub-protocols for different domains. Our work can provide the arithmetic domain to such a mixed-mode protocol. The protocol Auxiliator from [17] in the $(2+1)$-setting considers an extension of passively secure ASTRA [20], where only the dealer is allowed to deviate from the protocol while the parties can only be corrupted passively. This is the reverse situation to our setting with a trusted dealer, but potentially malicious parties. We discuss how both ideas may be merged in §D.

Remark on the Use of PCGs. Pseudorandom Correlation Generators (PCGs) [8] can reduce offline costs by enabling the dealer to distribute short seeds from which the parties locally compute their shares of triples. However, currently, efficient PCG constructions neither exist for function-dependent correlations that we use in this work or for authenticated triples used in the SPDZ approach. Hence, we cannot use them to reduce the offline costs in Tab. 1.

Organization

After the preliminaries (§2), we introduce several building blocks for our protocol (§3). Then, we introduce our novel distributed zero-knowledge proof and resulting actively secure protocol (§4). We provide an extensive evaluation regarding communication and run time (§5), and then conclude the paper (§6). The appendix provides information on our artifacts (§A), a note on ethical considerations (§B), additional building blocks based on standard constructions (§C), directions for alternative dealer instantiations (§D), the full security analysis of our protocol (§E), an extension of our protocol to dot products (§F), and additional empirical comparisons (§G).

2 Preliminaries

We consider n parties, denoted by $\mathcal{P}_1, ..., \mathcal{P}_n$, and a dealer $\mathcal{D}$. By *the parties*, we always just refer to the n parties excluding the dealer. Between each pair of parties and between each party and the dealer, all communication is done over a secure channel. The computation domain is $\mathbb{Z}_{2^k}$ for, e.g., $k = 32$ or $k = 64$. By σ, we denote the statistical security parameter, and κ is the computational security parameter. By $x \in_R A$ for some finite set A, we denote that x is picked uniformly at random from A.

For our novel proof, we use multiple parameters. First, m is the number of multiplication gates to be evaluated. We use a compression factor ω, and $s \in \mathbb{N}$ specifying the lifted ring $\mathbb{Z}_{2^{k+s}}$ that parts of our proof run on. The choice of s is later elaborated in §4.2.6.

We consider three different sharing types, summarized in Tab. 2: By $\langle x\rangle'$ for some $x \in \mathbb{Z}_{2^\ell}$ ($\ell = k$ or $\ell = k + s$ depending on the context), we denote an additive linear secret sharing $x = \langle x\rangle'_1 + \cdots + \langle x\rangle'_n$ between the parties where party $\mathcal{P}_i$ holds share $\langle x\rangle'_i \in \mathbb{Z}_{2^\ell}$. We also use additive sharings $\langle x\rangle$ where additionally, $\mathcal{D}$ knows all shares. This prevents any malicious party from changing its own share without causing an inconsistency from the additionally added redundancy. Our primary sharing type is the *star-sharing* $[x]$ that consists of $\langle\lambda_x\rangle$ for a *mask* $\lambda_x \in_R \mathbb{Z}_{2^\ell}$ alongside the *masked value* m_x that is known to all parties s.t. $x = \mathsf{m}_x + \lambda_x$. Such star-sharings are frequently used for MPC with function-dependent preprocessing with a dealer, e.g., in [20, 38]. For ease of notation, we write $\lambda_x^i := \langle\lambda_x\rangle_i$. All sharing types are linear, e.g., for a constant $c \in \mathbb{Z}_{2^\ell}$ and $[x], [y]$, $[cx + y]$ can be computed non-interactively by applying the linear combination $cx + y$ share-wise.

Table 2: Secret sharing types used throughout the paper.

	$\mathcal{D}$	$\mathcal{P}_1$	...	$\mathcal{P}_n$	Correlation
$\langle x\rangle'$	–	$\langle x\rangle'_1$	...	$\langle x\rangle'_n$	$x = \sum_{i=1}^{n} \langle x\rangle'_i$
$\langle x\rangle$	$(\langle x\rangle_1, \ldots, \langle x\rangle_n)$	$\langle x\rangle_1$	...	$\langle x\rangle_n$	$x = \sum_{i=1}^{n} \langle x\rangle_i$
$[x]$	$(\lambda_x^1, \ldots, \lambda_x^n)$	$(\mathsf{m}_x, \lambda_x^1)$	...	$(\mathsf{m}_x, \lambda_x^n)$	$\lambda_x = \sum_{i=1}^{n} \lambda_x^i$ $x = \mathsf{m}_x + \lambda_x$

To enable $\mathcal{D}$ and an arbitrary party to non-interactively generate the same pseudorandom values, we use PRFs as, e.g., in [38] and let $\mathcal{D}$ for each party $\mathcal{P}_i$ sample a random PRF seed and hand it to this party. Finally, we use the random oracle model (ROM) [5], assuming access to an oracle H which, on queries with arbitrary length, outputs uniformly random 2κ-bit values except for, if a query is repeated, providing the same output again.

Security Model. In this work, we assume an active adversary that can corrupt all but one party. The dealer, which is only available in the offline phase, is assumed to be trusted. We prove security in the simulation-based real-world/ideal-world paradigm [19] in §E.2.

The security proof is provided for the entire protocol and not on a per subprotocol basis. This decision stems from the fact that, as we will see in the following building blocks in §3, some subprotocols utilize a verification phase which is delayed to achieve better batching over multiple verifications. Still, for security, it is crucial *when* the verification happens in the overall protocol and ignoring that in the proof has been shown to have severe consequences, including break of privacy [15]. To tackle the security proof, we provide security considerations, including a *subsimulator* for each subprotocol. These give the security intuition while also providing necessary arguments, which we eventually pick up again in the overall security proof. We will later provide a simulator for the entire protocol, and the subsimulators provided throughout the paper simply describe subroutines that the complete simulator will call.

3 Building Blocks

3.1 Reconstruction

Reconstruction for a random sharing $\langle x \rangle$ is implemented by collecting all shares and computing their sum/XOR. We call the generic subprotocol for that Π_{rec} and give two possible instantiations: $\Pi_{\text{rec}}^{\text{bc}}$ lets each party broadcast its share to all other parties so that each party can compute the sum/XOR on its own. $\Pi_{\text{rec}}^{\text{king}}$ designates one party as $\mathcal{P}_{\text{king}}$, all parties send their shares to $\mathcal{P}_{\text{king}}$, $\mathcal{P}_{\text{king}}$ computes the sum/XOR, and then sends the result to all other parties. $\Pi_{\text{rec}}^{\text{bc}}$ uses only one round, but requires communication quadratic in n, whereas $\Pi_{\text{rec}}^{\text{king}}$ uses the approach from [25] to attain linear communication using two rounds. For load balancing, the role of $\mathcal{P}_{\text{king}}$ switches between all parties over all invocations.

Security: The subsimulator for Π_{rec} follows the protocol specification, given the (simulated) shares held by the honest parties. For a random sharing, i.e., all individual shares are random, real and ideal execution cannot be distinguished as no further inputs exist.

Still, a cheater may cause an honest party to receive an incorrect output. To prevent that, we build an actively secure reconstruction protocol $\Pi_{\text{rec}}^{\text{active}}$ (Fig. 1), utilizing that $\mathcal{D}$ has all shares of $\langle x \rangle$. We let it commit to x during setup, using a ROM commitment [39] with a random nonce r also additively shared between the parties. The parties can later verify that the value x they received is correct by reconstructing r and comparing it to the prior commit. Intuitively, the hiding property of ROM commitments keeps x private before opening as no subset of parties has any information on r. Additionally, the binding property ensures that the chance of incorrect x' being opened without `abort` is negligible. For a many openings, we can furthermore batch commit and verify by using a single $\mathsf{Com} = \mathsf{H}(x_1 || \dots || x_a || r)$.

Security: The reconstructed value x of an instance of the protocol is correct if the corresponding (batched) $\Pi_{\text{rec}}^{\text{active}}$.verify did accept. Otherwise, an honest party would receive $x' \neq x$ where $\mathsf{Com} = \mathsf{H}(\dots \|x\| \dots \|r) = \mathsf{H}(\dots \|x'| \dots \|r)$, i.e., there would be a hash collision which happens with negligible probability only. Also, after $\Pi_{\text{rec}}^{\text{active}}$.commit, the adversary is unable to derive any information on x. This stems from the fact that it lacks at least one random share of r, which hides r and, hence, by the hiding property of the ROM commitment, also x. Finally, the subsimulator for $\Pi_{\text{rec}}^{\text{active}}$ on a random sharing works as follows: For the commit phase, it samples $\mathsf{Com} \in_{\text{R}} \{0,1\}^{2\kappa}$ and hands it to the adversary on behalf of $\mathcal{D}$. For the open phase, it follows the specification of Π_{rec}. Finally, for the verify phase, it follows the specification of Π_{rec} to reconstruct r. For any following call of the adversary to ROM H, if the input is $\dots \|x\| \dots \|r$, it returns Com. Note that this works except for negligible probability, given the negligible probability of the adversary having invoked H earlier, using the same (then unknown) 3κ-bit value r. If any honest party receives a wrong x or r following the specification of Π_{rec}, the subsimulator simulates this party sending `abort` in the verification phase.

Subprotocol $\Pi_{\text{rec}}^{\text{active}}$

Input: $\langle x \rangle$ with $x \in \mathbb{Z}_{2^\ell}$ for some $\ell \in \mathbb{N}$, *Output:* x to all parties.

commit (batched):

1. For $1 \leq j \leq n$, $\mathcal{D}, \mathcal{P}_j$ sample shares $r^j \in_{\text{R}} \{0,1\}^{3\kappa}$ of $\langle r \rangle_{\text{bin}}$.
2. $\mathcal{D}$ computes $r = \bigoplus_{j=1}^{n} r^j$ and $\mathsf{Com} = \mathsf{H}(\dots \|x\| \dots ||r)$.
3. $\mathcal{D}$ sends Com to each party.

open:

4. $x = \Pi_{\text{rec}}(\langle x \rangle)$.

verify (batched):

5. $r = \Pi_{\text{rec}}(\langle r \rangle_{\text{bin}})$.
6. For $1 \leq j \leq n$, $\mathcal{P}_j$ sends `abort` if $\mathsf{Com} \neq \mathsf{H}(\dots \|x\| \dots ||r)$.

Figure 1: Opening an additive $\langle \cdot \rangle$-share to all parties.

We also require to reconstruct sharings $\langle x \rangle'$ where $\mathcal{D}$ has no knowledge of the shares. We allow here that an incorrect x' is being reconstructed as a cheater can simply provide an incorrect share or, as $\mathcal{P}_{\text{king}}$, can freely choose x'. Without any redundancy from $\mathcal{D}$, this cannot be detected. We will later show how to detect incorrect x' within the context of our protocol which is our main contribution. Yet, we need to ensure that all honest parties receive consistent values x'. To this end, we let each pair of parties exchange H of their outputs and `abort` in case of any mismatch. Using H allows to batch verification over many openings again. As we will later see, this batched verification is the only component of our overall protocol with total communication that is not linear in the number of parties n. We call this protocol $\Pi_{\text{rec}}^{\text{consistent}}$ (Fig. 2).

Subprotocol $\Pi_{\text{rec}}^{\text{consistent}}$

Input: $\langle x \rangle'$ with $x \in \mathbb{Z}_{2^\ell}$ for some $\ell \in \mathbb{N}$, *Output:* x to all parties.

open:

1. $x = \Pi_{\text{rec}}(\langle x \rangle')$.

verify (batched):

2. All pairs of parties $\mathcal{P}_i, \mathcal{P}_j$ exchange $h_i = \mathsf{H}(\dots \|x\| \dots)$, $h_j = \mathsf{H}(\dots \|x\| \dots)$ and send `abort` if $h_i \neq h_j$

Figure 2: Consistent opening of additive $\langle \cdot \rangle'$-shares to all parties.

Security: Each reconstructed value x is consistent across all honest parties if the corresponding (batched) $\Pi_{\text{rec}}^{\text{consistent}}$.verify did accept. Assume two honest parties with inconsistent outputs x_1, x_2. They exchange $\mathsf{H}(\dots \|x_1\| \dots), \mathsf{H}(\dots \|x_2\| \dots)$ which, except for a negligible probability of a collision, do not match. The subsimulator simply follows the specification of Π_{rec}.

Finally, for reconstruction value x of a sharing $[x]$, we need to reconstruct $\langle \lambda_x \rangle$. Then, we can set $x = \mathsf{m}_x + \lambda_x$. Yet, we need to rerandomize $\langle \lambda_x \rangle$ before applying $\Pi_{\mathsf{rec}}^{\mathsf{active}}$ using standard techniques. Details are provided in §C.2, and the corresponding protocol for rerandomizing and reconstructing $\langle \lambda_x \rangle$ is called $\Pi_{\mathsf{rec}}^{\mathsf{active,rerand}}$.

3.2 Multiplication

The subprotocol for multiplication is given in Fig. 3. We note that its general structure is well established in literature, e.g., for $n = 2$ it matches the passively secure protocol in ASTRA [20]. Given two sharings $[x], [y]$, a sharing of $z = xy$ can be computed by sampling the mask $\langle \lambda_z \rangle$ and then computing $\mathsf{m}_z = z - \lambda_z = (\mathsf{m}_x + \lambda_x)(\mathsf{m}_y + \lambda_y) - \lambda_z = \mathsf{m}_x\mathsf{m}_y + \mathsf{m}_x\lambda_y + \mathsf{m}_y\lambda_x + \lambda_x\lambda_y - \lambda_z$. To this end, we let the dealer $\mathcal{D}$ establish a sharing $\langle \gamma_{xy} \rangle$ for $\gamma_{xy} = \lambda_x\lambda_y$, the parties compute $\langle \mathsf{m}_z \rangle' = \mathsf{m}_x\mathsf{m}_y + \mathsf{m}_x\langle \lambda_y \rangle + \mathsf{m}_y\langle \lambda_x \rangle + \langle \gamma_{xy} \rangle - \langle \lambda_z \rangle$ by the linearity of additive secret sharing and then reconstruct this value, using $\Pi_{\mathsf{rec}}^{\mathsf{consistent}}$ (Fig. 2). Passive security immediately follows from the fact that by using a fresh random sharing $\langle \lambda_z \rangle$, $\langle \mathsf{m}_z \rangle'$ is random too, following the security arguments of Π_{rec}.

Subprotocol Π_{mult}

Input: $[x], [y]$ with $x, y \in \mathbb{Z}_{2^k}$, *Output:* $[z]$ where $z = xy \in \mathbb{Z}_{2^k}$.

1. $\mathcal{D}$ generates a $\langle \cdot \rangle$-sharing of $\gamma_{xy} = \lambda_x \cdot \lambda_y$ among parties: non-interactively sample shares with all but one party and send the last party its according share.
2. Parties and $\mathcal{D}$ locally sample random $\langle \lambda_z \rangle$.
3. Parties compute $\langle \mathsf{m}_z \rangle' = \mathsf{m}_x\mathsf{m}_y + \mathsf{m}_x\langle \lambda_y \rangle + \mathsf{m}_y\langle \lambda_x \rangle + \langle \gamma_{xy} \rangle - \langle \lambda_z \rangle$.
4. $\mathsf{m}_z = \Pi_{\mathsf{rec}}^{\mathsf{consistent}}(\langle \mathsf{m}_z \rangle')$.

Figure 3: Multiplication with Additive Errors.

Security: Regarding active security, $\Pi_{\mathsf{rec}}^{\mathsf{consistent}}$ guarantees consistency of the honest parties' result m'_z as soon as verify is run, but the adversary can cause it to be offset by an error δ: $\mathsf{m}'_z = \mathsf{m}_z + \delta$. The subsimulator begins by following the protocol specification. Then, the subsimulator runs the subsimulator for $\Pi_{\mathsf{rec}}^{\mathsf{consistent}}$ with input $\langle \mathsf{m}_z \rangle'$ random as described above. This randomness yields indistinguishability between real and ideal execution as per the considerations for Π_{rec}. Specifically, the randomness also prevents the adversary of choosing the error δ dependent on x, y.

3.3 Collective Sum Sharing

Our proof will later require to obtain a $[\cdot]$-sharing of the sum $\sum x^j$ over values x^j locally computed by the individual parties. While each party could $[\cdot]$-share its x^j after which the sum can be computed, this would result in $O(n^2)$ total communication per invocation. Instead, we propose subprotocol Π_{shareSum} (Fig. 4).

Subprotocol Π_{shareSum}

Input: $x^j \in \mathbb{Z}_{2^\ell}$ ($\ell \in \mathbb{N}$), provided by $\mathcal{P}_j$, for $1 \le j \le n$,
Output: $[x]$ where $x = \sum_{j=1}^{n} x^j$.

1. The parties and $\mathcal{D}$ non-interactively sample $\langle \lambda_x \rangle$
2. The parties compute $\langle \mathsf{m}_x \rangle'$ for $\mathsf{m}_x = x - \lambda_x$ by each $\mathcal{P}_j$ setting $\mathsf{m}_x^j = x^j - \lambda_x^j$.
3. $\mathsf{m}_x = \Pi_{\mathsf{rec}}^{\mathsf{consistent}}(\langle \mathsf{m}_x \rangle')$.

Figure 4: Π_{shareSum}: Computing sum of inputs from all parties.

Security: Regarding correctness, note that $\mathsf{m}_x + \lambda_x = \sum_{j=1}^{n}(x^j - \lambda_x^j) + \lambda_x = \sum_{j=1}^{n} x^j$. The subsimulator follows the protocol specification, using the subsimulator for $\Pi_{\mathsf{rec}}^{\mathsf{consistent}}$. Note that the input to $\Pi_{\mathsf{rec}}^{\mathsf{consistent}}$ is a random sharing as it uses freshly random $\langle \lambda_x \rangle$. As for multiplications, cheaters can cause honest parties to receive inconsistent outputs m_x, but this would be detected when running verify. Furthermore, any consistent error introduced is independent of each honest party's input as $\Pi_{\mathsf{rec}}^{\mathsf{consistent}}$ receives a random sharing as input, similar to the case in Π_{mult} (Fig. 3).

3.4 PRF-Based Random Sampling

To sample public random values, we let the parties recover a random sharing. For efficiency, we do this for a seed rs and then derive many values from that using a PRF. The subprotocol is given in Fig. 5.

Subprotocol Π_{rnd}

Input: Finite set A, $\ell \in \mathbb{N}$,
Output: For a random $\mathsf{rs} \in_R \{0,1\}^\kappa$: pseudorandom $r_1, \ldots, r_\ell \in A$ sampled using $\mathsf{PRF}_{\mathsf{rs}}$ to all parties and $\mathcal{D}$, rs to the adversary.

1. The parties and $\mathcal{D}$ non-interactively sample $\langle \mathsf{rs} \rangle$ for $\mathsf{rs} \in_R \{0,1\}^\kappa$
2. $\mathsf{rs} \leftarrow \Pi_{\mathsf{rec}}^{\mathsf{active}}(\langle \mathsf{rs} \rangle)$.
3. Use $\mathsf{PRF}_{\mathsf{rs}}$ to sample $r_1, \ldots, r_\ell \in_R A$.

Figure 5: Sampling public, random values using a PRF.

Security: Note that the parties do not just learn random values as in a coin flip protocol, but also the seed rs from which they were derived. Hence, we need to keep in mind that the adversary also learns rs. The subsimulator simply follows the protocol specification, utilizing the subsimulator of $\Pi_{\mathsf{rec}}^{\mathsf{active}}$ that receives a random sharing as input. We recall from §3.1 that after $\Pi_{\mathsf{rec}}^{\mathsf{active}}$.commit, the adversary is unable to learn anything about rs. Furthermore, only after successful $\Pi_{\mathsf{rec}}^{\mathsf{active}}$.verify, it is guaranteed that all honest parties have the correct rs and, hence, $r_1, \ldots, r_\ell$.

4 Actively Secure MPC

4.1 Technical Overview

Our actively secure protocol lets each party share its inputs (details in §C.1) and then optimistically evaluates the circuit, using for each multiplication Π_{mult} (§3.2), thus enabling malicious parties to introduce errors. Additions and multiplications by a constant are executed non-interactively, using the linearity of the underlying sharing schemes. Before revealing outputs using $\Pi_{\mathsf{rec}}^{\mathsf{active,rerand}}$ (§3.1), we run our novel DZK proof to detect any previously introduced errors, which is our main contribution explained in this section.

First, we introduce a high-level proof in §4.2.1 that follows ideas from [12, 46]. This proof will require to compute a sharing $[\Gamma]$, and proving that $[\Gamma]$ is correct constitutes the main challenge, which we begin to discuss in §4.2.2. In [12, 46], each party secret shares a locally computed value Γ^j and the sum over all $\left[\Gamma^j\right]$ yields $[\Gamma]$. Then, one zk-FLIOP is used for each $\left[\Gamma^j\right]$, letting its provider $\mathcal{P}_j$ prove that it has shared the correct value. Instead, we use only *one proof to rule them all* to check that the sum $[\Gamma]$ is correct. Departing from prior zk-FLIOP approaches, in our proof, the parties collectively act as prover. Similar to prior works, they then also simultaneously act as

verifiers, and the dealer $\mathcal{D}$ acts as an additional verifier. Using a single proof for all n parties cuts the overall complexity for the required zk-FLIOP(s) by factor $n\times$, achieving for the first time good scalability to many parties. Also, our protocol Π_{shareSum} (§3.3) is crucial to enable parties to efficiently compute $[\Gamma]$ without each party having to share a value separately. This is only enabled by the joint nature of our proof and is crucial to the efficiency of the following steps.

Our statement to be proven, like also the case in [12, 46], can be expressed as $\vec{\mathbf{x}} \cdot \langle \vec{\mathbf{y}} \rangle \stackrel{?}{=} [z]$. In our case of one single joint proof, vector $\vec{\mathbf{x}}$ is known to all parties, but not to $\mathcal{D}$, and vector $\vec{\mathbf{y}}$ is additively shared between the parties (with $\mathcal{D}$ having all shares and, hence, the cleartext value). In contrast to that, e.g. [12] would consider $\vec{\mathbf{y}}$ not to be secret shared, but known in the clear to the individual prover (in each of the n instances) and $\mathcal{D}$. Note that $\vec{\mathbf{x}}$ is known to all and, hence, also at least one honest party, while the trusted $\mathcal{D}$ has $\vec{\mathbf{y}}$. The same redundancy exists for $[z]$ as each party has m_z and $\mathcal{D}$ knows λ_z. Hence, each share is held by at least one honest party, preventing the adversary from changing any sharings without causing inconsistencies.

Our proof now iteratively decreases the size of the vectors, reducing the proof to proving $\vec{\mathbf{x}'} \cdot \langle \vec{\mathbf{y}'} \rangle \stackrel{?}{=} [z']$ over smaller vectors. While [12] approaches such a degree reduction relying on polynomial interpolation, this is computationally expensive, even amplified by now requiring Galois extension rings to even enable interpolation. Fortunately, [46], while working in another setting, provides a method that essentially splits the vectors into smaller segments and then computes a random linear combination on them to obtain $\vec{\mathbf{x}'}, \langle \vec{\mathbf{y}'} \rangle, [z']$. This step, which we explain in §4.2.3, is repeated recursively until the statement is sufficiently small for a direct proof, while maintaining communication cost sublinear in the original vector sizes. The final proof (§4.2.4) is similar to the degree reduction step, but computes a scalar triple $x' \cdot \langle y' \rangle \stackrel{?}{=} [z']$ which can then be opened and inspected for verification in the clear. The factor by which the vectors get smaller in each iteration is some parameter $\omega \geq 2$ which we call the *compression factor*.

The random linear combinations used throughout our proof are intended to cause any error δ in a triple, i.e., $z = \vec{\mathbf{x}} \cdot \vec{\mathbf{y}} + \delta$ or further errors introduced by corrupt parties in the proof to yield an error δ' in the resulting triple $\vec{\mathbf{x}'}, \langle \vec{\mathbf{y}'} \rangle, [z']$ too. This relies essentially on random linear combinations of values where at least one is not zero to also be non-zero except for negligible probability on fields, but this does unfortunately not hold for a ring $\mathbb{Z}_{2^k}$. Here, the SPD$\mathbb{Z}_{2^k}$-trick [22] enables to use a larger ring so that any input error modulo 2^k still yields an output error in the larger ring. This is exploited in [46] which iteratively computes random linear combinations, but does so on a ring $\mathbb{Z}_{2^{k+s}}$ with s sufficiently large so that after all iterations, any initial error modulo 2^k yields an error modulo 2^{k+s} except for negligible probability. Hence, in §4.2.2 we switch from computation on $\mathbb{Z}_{2^k}$ to $\mathbb{Z}_{2^{k+s}}$, noting that in contrast to SPD$\mathbb{Z}_{2^k}$ [22], this larger ring is only used for verification itself and its overhead amortizes to zero over many multiplications. Throughout our considerations regarding soundness, we will count the number of linear combinations, where each one enables errors to collect more factors of two while propagating through the proof. Should the number of two-factors, i.e., ℓ s.t. 2^ℓ is a divisor of the error, reach $k+s$, it overflows to zero, hence vanishing. Therefore, a careful analysis is required and later in §4.2.6 we choose s s.t. the chance of such an overflow becomes negligible.

Finally, we carefully design our protocol s.t. our dealer $\mathcal{D}$ is not needed in the online phase (and hence not a helper). $\mathcal{D}$ only works on $\vec{\mathbf{y}}$, masks λ_z of $[z]$, and random linear coefficients. The first two are already known during the offline phase, random λ_z are sampled there and $\vec{\mathbf{y}}$ is defined only over values from the offline phase (see its definition in §4.2.1). For random linear coefficients, our protocol Π_{rnd} (§3.4) is designed s.t. $\mathcal{D}$ knows them in advance which does not harm soundness as $\mathcal{D}$ is no prover.

4.2 Distributed Zero-Knowledge Verification

4.2.1 Starting Point: The High-Level Proof. We assume for now that the internal verification verify of the used subprotocols is immediately done for each instance and will later describe how it can be delayed for optimized batching in §4.2.5. Recall that while the multiplication subprotocol Π_{mult} (Fig. 3) ensures that its output is a consistent sharing of some value z, this is not necessarily the product of the inputs x, y. As $\langle \mathsf{m}_z \rangle'$ used by Π_{mult} has no redundancy, a corrupt party can simply change its share without causing any discrepancy in the sharing. Hence, we propose a verification protocol that ensures that the reconstructed m_z is correct in Fig. 6, and in the following, we describe its components step by step. The statement to verify can be phrased as follows: $\mathsf{m}_x\mathsf{m}_y + \mathsf{m}_x\lambda_y + \lambda_x\mathsf{m}_y + \gamma_{xy} - \lambda_z \stackrel{?}{=} \mathsf{m}_z$. To check that inexpensively for all m multiplications $[x_i] \cdot [y_i] = [z_i]$, we use a random linear combination check, following [12, 46]. We sample random public $r_1, \ldots, r_m$ and test for

$$\sum_{i=1}^{m} r_i \left(\mathsf{m}_{x_i}\mathsf{m}_{y_i} + \mathsf{m}_{x_i}\lambda_{y_i} + \lambda_{x_i}\mathsf{m}_{y_i} + \gamma_{x_i y_i} - \lambda_{z_i} \right) \stackrel{?}{=} \sum_{i=1}^{m} r_i \mathsf{m}_{z_i}.$$

Note that [12] uses fields while we consider the ring $\mathbb{Z}_{2^k}$ which drastically decreases the soundness of the check. Hence, we borrow the trick from [46] to repeat the verification σ (statistical security parameter) many times while also just considering random binary values for $r_1, \ldots, r_m$. One check detects incorrect m_{z_i} with probability of 0.5 and, hence, executing σ checks has a negligible probability of $2^{-\sigma}$ to not detect incorrect values. We rewrite the prior expression to $\Lambda + \Gamma \stackrel{?}{=} 0$ where

$$\Lambda := \sum_{i=1}^{m} r_i \left(\mathsf{m}_{x_i}\mathsf{m}_{y_i} - \mathsf{m}_{z_i} \right),$$

$$\Gamma := \sum_{i=1}^{m} r_i \left(\mathsf{m}_{x_i}\lambda_{y_i} + \lambda_{x_i}\mathsf{m}_{y_i} + \gamma_{x_i y_i} - \lambda_{z_i} \right).$$

The part Λ is known by all parties in the clear. In Fig. 6, its computation and the sampling of random bits for all σ checks can be found in steps 1 and 2. Also, note that in step 14, it is checked if $\Lambda + \Gamma \stackrel{?}{=} 0$ for all checks. Regarding Γ, we could let each party $\mathcal{P}_j$ compute $\sum_{i=1}^{m} r_i \left(\mathsf{m}_{x_i}\lambda_{y_i}^j + \lambda_{x_i}^j\mathsf{m}_{y_i} + \gamma_{x_i y_i}^j - \lambda_{z_i}^j \right)$, $[\cdot]$-share the result and then add up the parts provided by all parties to get $[\Gamma]$. The remaining challenge is to guarantee correct computation of $[\Gamma]$, given that the parts provided by the parties use m_v-values not known to $\mathcal{D}$ and λ_v^j-values known to only one party, yet again opening the potential for cheating.

Subprotocol Π_{verify}

Input for m multiplications: $[x_i], [y_i], [z_i]$ for multiplication inputs x_i, y_i and outputs $z_i \in \mathbb{Z}_{2^k}, 1 \le i \le m$,

Output: Accept if all triples are correct, Reject otherwise.

Prepare Top-Level Verification

1. $(r_i^\ell)_{1\le i\le m, 1\le \ell\le\sigma} = \Pi_{\mathsf{rnd}}(\{0,1\}, m\sigma)$
2. Parties compute $\Lambda^\ell = \sum_{1\le i\le m} r_i^\ell \left(\mathsf{m}_{x_i}\mathsf{m}_{y_i} - \mathsf{m}_{z_i}\right)$, $1 \le \ell \le \sigma$.

Prepare Dot-Product Proof

3. Parties define vectors $\vec{\mathbf{a}'}^\ell$ for $1 \le \ell \le \sigma$, and parties and $\mathcal{D}$ define vector $\vec{\mathbf{b}'}$, all of dimension $3m$ as follows:

$$\vec{\mathbf{a}'}^\ell = \begin{pmatrix} r_1^\ell \mathsf{m}_{x_1} \\ r_1^\ell \mathsf{m}_{y_1} \\ r_1^\ell \\ \vdots \\ r_m^\ell \mathsf{m}_{x_m} \\ r_m^\ell \mathsf{m}_{y_m} \\ r_m^\ell \end{pmatrix} \qquad \langle\vec{\mathbf{b}'}\rangle = \begin{pmatrix} \langle\lambda_{y_1}\rangle \\ \langle\lambda_{x_1}\rangle \\ \langle\gamma_{x_1 y_1} - \lambda_{z_1}\rangle \\ \vdots \\ \langle\lambda_{y_m}\rangle \\ \langle\lambda_{x_m}\rangle \\ \langle\gamma_{x_m y_m} - \lambda_{z_m}\rangle \end{pmatrix}$$

4. Parties lift $\vec{\mathbf{a}'}^\ell$ from $\mathbb{Z}_{2^k}$ to $\mathbb{Z}_{2^{k+s}}$ yielding $\vec{\mathbf{a}}^\ell$ (using zero-extension) for $1 \le \ell \le \sigma$.
5. Obtain $\langle\vec{\mathbf{b}}\rangle$ over $\mathbb{Z}_{2^{k+s}}$ from $\langle\vec{\mathbf{b}'}\rangle$ by $\mathcal{D}, \mathcal{P}_j$ lifting their respective share $\langle\vec{\mathbf{b}'}\rangle_j$ from $\mathbb{Z}_{2^k}$ to $\mathbb{Z}_{2^{k+s}}$ (using zero-extension).
6. Each party $\mathcal{P}_j$ computes $c_j^\ell = \vec{\mathbf{a}}^\ell \cdot \langle\vec{\mathbf{b}}\rangle_j$ for $1 \le \ell \le \sigma$.
7. Set $\left[c^\ell\right] \leftarrow \Pi_{\mathsf{shareSum}}(c_1^\ell, \ldots, c_n^\ell)$ for $1 \le \ell \le \sigma$.
8. Define $\left[\Gamma^\ell\right]$ by locally removing the s most significant bits from all shares of $\left[c^\ell\right]$ for $1 \le \ell \le \sigma$.
9. $(\tilde{r}^\ell)_{1\le\ell\le\sigma} = \Pi_{\mathsf{rnd}}(\mathbb{Z}_{2^{k+s}}, \sigma)$.
10. Compute $\vec{\mathbf{x}} = \sum_{\ell=1}^\sigma \tilde{r}^\ell \vec{\mathbf{a}}^\ell$, $\langle\vec{\mathbf{y}}\rangle = \langle\vec{\mathbf{b}}\rangle \in \mathbb{Z}_{2^{k+s}}^{3m}$.
11. Compute $[z] = \sum_{\ell=1}^\sigma \tilde{r}^\ell \cdot \left[c^\ell\right]$.

Run Dot-Product Proof

12. Repeat $\vec{\mathbf{x}}, \langle\vec{\mathbf{y}}\rangle, [z] \leftarrow \Pi_{\mathsf{reduceDeg}}(\vec{\mathbf{x}}, \langle\vec{\mathbf{y}}\rangle, [z])$ until $\vec{\mathbf{x}}, \vec{\mathbf{y}}$ have dimension $\le \omega$
13. $\Pi_{\mathsf{checkTriple}}(\vec{\mathbf{x}}, \langle\vec{\mathbf{y}}\rangle, [z])$

Finalize Top-Level Verification

14. For $1 \le \ell \le \sigma$, compute $\left[v^\ell\right] = \Lambda^\ell + \left[\Gamma^\ell\right]$ (by adding Λ^ℓ to m_{Γ^ℓ}) and set $v^\ell = \mathsf{m}_{v^\ell} + \lambda_{v^\ell}$ with $\lambda_{v^\ell} \leftarrow \Pi_{\mathsf{rec}}^{\mathsf{active,rerand}}(\langle\lambda_{v^\ell}\rangle)$. If all results are zero, Accept, otherwise, Reject.

Figure 6: Π_{verify}: Verification with a Distributed Zero-Knowledge Proof.

As discussed in §4.1, instead of n individual proofs to check that parties shared the correct values as in [12, 46], we only use one proof for $\Gamma \stackrel{?}{=} \sum_{i=1}^m r_i \left(\mathsf{m}_{x_i}\lambda_{y_i} + \lambda_{x_i}\mathsf{m}_{y_i} + \gamma_{x_i y_i} - \lambda_{z_i}\right)$. Furthermore, instead of letting each party share its part individually, we use Π_{shareSum} as we now are only interested in the sum over the values provided by the individual parties. This requires total communication of only $O(n)$ values instead of $O(n^2)$ if each party would individually share its part.

We express Γ by the following dot product:

$$\Gamma = \begin{pmatrix} r_1\mathsf{m}_{x_1} \\ r_1\mathsf{m}_{y_1} \\ r_1 \\ \vdots \\ r_m\mathsf{m}_{x_m} \\ r_m\mathsf{m}_{y_m} \\ r_m \end{pmatrix} \cdot \begin{pmatrix} \lambda_{y_1} \\ \lambda_{x_1} \\ \gamma_{x_1 y_1} - \lambda_{z_1} \\ \vdots \\ \lambda_{y_m} \\ \lambda_{x_m} \\ \gamma_{x_m y_m} - \lambda_{z_m} \end{pmatrix} \tag{1}$$

Note that in our protocol, the first vector is known to all parties and the second vector is $\langle\cdot\rangle$-shared. We name the first vector $\vec{\mathbf{a}'}$ and the second vector $\vec{\mathbf{b}'}$ so that $\vec{\mathbf{a}'} \cdot \vec{\mathbf{b}'} = \Gamma$ (step 3 of Fig. 6). Then, we use the linearity $\Gamma = \vec{\mathbf{a}'} \cdot \vec{\mathbf{b}'} = \vec{\mathbf{a}'} \cdot \sum_{j=1}^n \langle\vec{\mathbf{b}'}\rangle_j = \sum_{j=1}^n \vec{\mathbf{a}'} \cdot \langle\vec{\mathbf{b}'}\rangle_j$, enabling each party to compute a part of Γ (steps 6, 7 which are modified as discussed in the next section). After establishing a sharing $[\Gamma]$, to check its correctness we then let all parties collectively act as prover and verifier at the same time while $\mathcal{D}$ acts as an additional verifier.

4.2.2 Verifying $[\Gamma]$. Recall from §4.1 that our goal is to use a larger ring $\mathbb{Z}_{2^{k+s}}$ for our verification, utilizing the SPD$\mathbb{Z}_{2^k}$ trick [22]. Lifting all values to $\mathbb{Z}_{2^{k+s}}$ in §4.2.1 would unfortunately have yielded linear (w.r.t. m) overhead as the values from all multiplications had to be considered and hence would have to be lifted to the larger ring. Instead, we follow the idea from [46] to lift the dot product from Eq. 1 to $\mathbb{Z}_{2^{k+s}}$, which has communication independent of the dot product's dimension and, hence, m.

As the left vector $\vec{\mathbf{a}'}$ is held by all parties, they can simply lift each entry from $\mathbb{Z}_{2^k}$ to $\mathbb{Z}_{2^{k+s}}$ using zero-extension[3], obtaining a vector $\vec{\mathbf{a}}$. The second vector $\vec{\mathbf{b}'}$ from Eq. 1 is $\langle\cdot\rangle$-shared. We let the parties and dealer locally lift their shares $\langle\vec{\mathbf{b}'}\rangle_j$ to $\mathbb{Z}_{2^{k+s}}$, noting that this does not necessarily imply that the represented value $\vec{\mathbf{b}'}$ also is correctly lifted. We call the resulting shared vector $\langle\vec{\mathbf{b}}\rangle$. Finally, we let each party $\mathcal{P}_j$ compute $c_j := \vec{\mathbf{a}} \cdot (\langle\vec{\mathbf{b}}\rangle_j)$ and use Π_{shareSum} to compute $[c] := \sum_{j=1}^n c_j$, see steps 4-7 in Fig. 6.

Note that if no cheating occurs, $\vec{\mathbf{a}}, \vec{\mathbf{b}}, c'$ are equal modulo 2^k to the respective vectors $\vec{\mathbf{a}'}, \vec{\mathbf{b}'}$, and scalar Γ in Eq. 1 while the same is not necessarily the case modulo 2^{k+s}:

$$\vec{\mathbf{a}} \equiv_{2^k} \vec{\mathbf{a}'} \qquad \vec{\mathbf{b}} = \sum_{j=1}^n \langle\vec{\mathbf{b}}\rangle_j \equiv_{2^k} \sum_{j=1}^n \langle\vec{\mathbf{b}'}\rangle_j = \vec{\mathbf{b}'}$$

$$c = \sum_{j=1}^n c_j = \sum_{j=1}^n \vec{\mathbf{a}} \cdot \langle\vec{\mathbf{b}}\rangle_j \equiv_{2^k} \sum_{j=1}^n \vec{\mathbf{a}'} \cdot \langle\vec{\mathbf{b}'}\rangle_j = \vec{\mathbf{a}'} \cdot \vec{\mathbf{b}'} = \Gamma$$

Hence, we can reinterpret $[c]$ over $\mathbb{Z}_{2^{k+s}}$ as $[\Gamma]$ over $\mathbb{Z}_{2^k}$ by all parties locally discarding the s most significant bits of their shares of $[c]$ (step 8). Note that corrupt parties may only cheat by providing incorrect c_j to Π_{shareSum} or cheat during Π_{shareSum}. Yet, as we will show, if the obtained $[\Gamma]$ is incorrect, this will be detected.

From §4.2.1, recall that for sufficient soundness, the proof needs to be repeated for σ times, actually resulting in σ dot products $\vec{\mathbf{a}}^\ell \cdot \vec{\mathbf{b}} = \left[c^\ell\right], 1 \le \ell \le \sigma$ to be computed and verified.[4] We now follow [46] by merging these σ dot products into one. For that, we sample random $\tilde{r}^\ell \in_{\mathrm{R}} \mathbb{Z}_{2^{k+s}}, 1 \le \ell \le \sigma$ and non-interactively compute linear combinations $\vec{\mathbf{x}} = \sum_{\ell=1}^\sigma \tilde{r}^\ell \vec{\mathbf{a}}^\ell, z = \sum_{\ell=1}^\sigma \tilde{r}^\ell c^\ell$, and rename $\vec{\mathbf{y}} := \vec{\mathbf{b}}$ (steps 9, 10, 11), noting that $\vec{\mathbf{x}} \cdot \vec{\mathbf{y}} = \left(\sum_{\ell=1}^\sigma \tilde{r}^\ell \vec{\mathbf{a}}^\ell\right) \cdot \vec{\mathbf{b}} = \sum_{\ell=1}^\sigma \tilde{r}^\ell \vec{\mathbf{a}}^\ell \cdot \vec{\mathbf{b}} = \sum_{\ell=1}^\sigma \tilde{r}^\ell c^\ell = z$.[5] Hence, we merge σ dot products into one, resulting in a pair of vectors with dimension $3m$.

[3] Other extensions would also be possible as long as used consistently.

[4] $\vec{\mathbf{b}}$ is independent of the individual random r_i-values that vary between the σ iterations. Hence, $\vec{\mathbf{b}}$ remains constant across all dot products.

[5] We note that this approach is noted as possible optimization in [46], but not part of their original protocol.

Soundness: If verify succeeds for all prior operations, we can assume consistent sharings. Now, each corrupt party $\mathcal{P}_j$ may input incorrect c_j^ℓ ($1 \le \ell \le \sigma$) to Π_{shareSum} or cause Π_{shareSum} to output an incorrect value, which is consistent after running the corresponding verify. This results in $c^\ell = \vec{\mathbf{a}}^\ell \cdot \vec{\mathbf{b}} + \delta^\ell, 1 \le \ell \le \sigma$ for accumulated errors δ^ℓ chosen by the adversary which, by the properties of Π_{shareSum}, are independent of the other parties' inputs. Then,

$$z = \sum_{\ell=1}^{\sigma} \tilde{r}^\ell c^\ell = \sum_{\ell=1}^{\sigma} \tilde{r}^\ell \left(\vec{\mathbf{a}}^\ell \cdot \vec{\mathbf{b}} + \delta^\ell\right) = \sum_{\ell=1}^{\sigma} \left(\vec{\mathbf{x}} \cdot \vec{\mathbf{y}} + \tilde{r}^\ell \delta^\ell\right).$$

Hence, errors δ^ℓ in the c^ℓ result in an error $z = \vec{\mathbf{x}} \cdot \vec{\mathbf{y}} + \delta'$, where δ' is a random linear combination of the prior errors. Moreover, the adversary is unable to gain any information on the random coefficients by the properties of Π_{rnd} when choosing δ^ℓ which is before opening the coefficients.

4.2.3 Reducing the Vector Dimension. We want to efficiently verify correctness of the previously computed triple $(\vec{\mathbf{x}}, \vec{\mathbf{y}}, z)$. For this, we iteratively reduce the vector dimensions of the triple by factor ω (step 12 in Fig. 6) until it is sufficiently small to run one final check with communication linear in the reduced dimension (step 13), following the proof structure from [46]. The dimensions are reduced by the degree reduction protocol in Fig. 7.

Subprotocol $\Pi_{\mathsf{reduceDeg}}$

Input: Vectors $\vec{\mathbf{x}}, \langle\vec{\mathbf{y}}\rangle$ and scalar $[z], \vec{\mathbf{x}}, \vec{\mathbf{y}} \in \mathbb{Z}_{2^{k+s}}^d, z \in \mathbb{Z}_{2^{k+s}}, d > \omega$,

Output: Vectors $\vec{\mathbf{x}'}, \langle\vec{\mathbf{y}'}\rangle$ and scalar $[z'], \vec{\mathbf{x}'}, \vec{\mathbf{y}'} \in \mathbb{Z}_{2^{k+s}}^{\lceil d/\omega \rceil}, z' \in \mathbb{Z}_{2^{k+s}}$.

1. Split $\vec{\mathbf{x}}, \langle\vec{\mathbf{y}}\rangle$ into ω equally sized vectors $\vec{\mathbf{x}^1}, \ldots, \vec{\mathbf{x}^\omega}, \langle\vec{\mathbf{y}^1}\rangle, \ldots, \langle\vec{\mathbf{y}^\omega}\rangle$ of dimension $d' := \lceil d/\omega \rceil$ (pad with 0 entries if $\omega \nmid d$).
2. Each party $\mathcal{P}_i$ computes $z_i^{j,\ell} = \vec{\mathbf{x}^j} \cdot \langle\vec{\mathbf{y}^\ell}\rangle_i$ for all $1 \le j, \ell \le \omega$ except for $j = \ell = 1$.
3. For $1 \le j, \ell \le \omega$ except for $j = \ell = 1$, set $\left[z^{j,\ell}\right] \leftarrow \Pi_{\mathsf{shareSum}}(z_1^{j,\ell}, \ldots, z_n^{j,\ell})$.
4. Compute $\left[z^{1,1}\right] = [z] - \sum_{\ell=2}^{\omega} \left[z^{\ell,\ell}\right]$.
5. $(\alpha^\ell, \beta^\ell)_{1 \le \ell \le \omega} \leftarrow \Pi_{\mathsf{rnd}}(\mathbb{Z}_{2^{k+s}}, 2\omega)$.
6. Compute
$$\vec{\mathbf{x}'} = \sum_{\ell=1}^{\omega} \alpha^\ell \vec{\mathbf{x}^\ell} \quad \langle\vec{\mathbf{y}'}\rangle = \sum_{\ell=1}^{\omega} \beta^\ell \langle\vec{\mathbf{y}^\ell}\rangle \quad [z'] = \sum_{j=1}^{\omega}\sum_{\ell=1}^{\omega} \alpha^j \beta^\ell \left[z^{j,\ell}\right].$$

Figure 7: $\Pi_{\mathsf{reduceDeg}}$: Degree Reduction Step.

For $(\vec{\mathbf{x}}, \vec{\mathbf{y}}, z)$ with vector dimension d, we split the vectors into smaller vectors $\vec{\mathbf{x}^1}, \ldots, \vec{\mathbf{x}^\omega}$ and $\vec{\mathbf{y}^1}, \ldots, \vec{\mathbf{y}^\omega}$ of dimension $d' = d/\omega$ in step 1.[6] Later (step 5), $\Pi_{\mathsf{reduceDeg}}$ samples random coefficients α^ℓ, $\beta^\ell \in_R \mathbb{Z}_{2^{k+s}}$ for $1 \le \ell \le \omega$ to compute random linear combinations $\vec{\mathbf{x}'} = \sum_{\ell=1}^{\omega} \alpha^\ell \vec{\mathbf{x}^\ell}$ and $\vec{\mathbf{y}'} = \sum_{\ell=1}^{\omega} \beta^\ell \vec{\mathbf{y}^\ell}$ (step 6). We note that $z' := \vec{\mathbf{x}'} \cdot \vec{\mathbf{y}'} = \sum_{j=1}^{\omega}\sum_{\ell=1}^{\omega} \alpha^j \beta^\ell \cdot \vec{\mathbf{x}^j} \cdot \vec{\mathbf{y}^\ell}$.

While $\vec{\mathbf{x}'}$ and $\langle\vec{\mathbf{y}'}\rangle$ can be computed non-interactively, sharings $[\vec{\mathbf{x}^j} \cdot \vec{\mathbf{y}^\ell}]$ need to be established. Then, $[z']$ can be computed as linear combination on these values. Here, we use that $\vec{\mathbf{x}^j} \cdot \vec{\mathbf{y}^\ell} = \vec{\mathbf{x}^j} \cdot \sum_{i=1}^{n} \langle\vec{\mathbf{y}^\ell}\rangle_i = \sum_{i=1}^{n} (\vec{\mathbf{x}^j} \cdot \langle\vec{\mathbf{y}^\ell}\rangle_i) =: \sum_{i=1}^{n} z_i^{j,\ell}$, where each party $\mathcal{P}_j$ provides its locally computed $z_i^{j,\ell}$ (steps 2, 3). We also ensure that $\sum_{\ell=1}^{\omega} z^{\ell,\ell} = z$ by instead of letting parties provide values to compute $\left[z^{1,1}\right]$, deriving it from the other $\left[z^{\ell,\ell}\right]$ and $[z]$ (step 4).

[6] If needed, we pad the vectors with zeros s.t. d is a multiple of ω.

Soundness: Again, assume that verify succeeds for all prior operations, and we hence have consistent sharings. Each corrupt party $\mathcal{P}_i$ can contribute incorrect $z_i^{j,\ell}$ or cheat in Π_{shareSum}, yielding an incorrect output, consistent after running the corresponding verify. This results in $z^{j,\ell} = \vec{\mathbf{x}^j} \cdot \vec{\mathbf{y}^\ell} + \delta^{j,\ell}, 1 \le j, \ell \le \omega$ for accumulated errors $\delta^{j,\ell}$ chosen by the adversary, independent of the other parties' inputs. A special case is $\delta^{1,1} = \delta - \sum_{\ell=2}^{\omega} \delta^{\ell,\ell}$ where δ is the input error with $z = \vec{\mathbf{x}} \cdot \vec{\mathbf{y}} + \delta$, as $z^{1,1}$ is computed from z and the other $z^{\ell,\ell}$. It now holds that

$$\begin{aligned} z' = \sum_{j=1}^{\omega}\sum_{\ell=1}^{\omega} \alpha^j \beta^\ell z^{j,\ell} &= \sum_{j=1}^{\omega}\sum_{\ell=1}^{\omega} \alpha^j \beta^\ell \left(\vec{\mathbf{x}^j} \cdot \vec{\mathbf{y}^\ell} + \delta^{j,\ell}\right) \\ &= \vec{\mathbf{x}'} \cdot \vec{\mathbf{y}'} + \sum_{j=1}^{\omega} \alpha^j \sum_{\ell=1}^{\omega} \beta^\ell \delta^{j,\ell}. \end{aligned}$$

Hence, there is an error δ' in the output triple $z' = \vec{\mathbf{x}'} \cdot \vec{\mathbf{y}'} + \delta'$ with $\delta' = \sum_{j=1}^{\omega} \alpha^j \sum_{\ell=1}^{\omega} \beta^\ell \delta^{j,\ell}$. Each $\sum_{\ell=1}^{\omega} \beta^\ell \delta^{j,\ell}$ is a random linear combination and the outer $\sum_{j=1}^{\omega} \alpha^j \ldots$ then is a linear combination over the results. In addition, assuming an input error of $\delta \neq 0$ with t s.t. $2^t \mid \delta, 2^{t+1} \nmid \delta$, then as $\delta = \sum_{\ell=1}^{\omega} \delta^{\ell,\ell}$, there also exists one $\delta^{\ell,\ell}$ with $2^{t+1} \nmid \delta^{\ell,\ell}$ which goes into the random linear combination(s). We also note that all errors $\delta^{j,\ell}$ are independent of the random coefficients by the properties of Π_{rnd} and the fact that the coefficients are revealed after the errors were already chosen.

4.2.4 Final Triple Check. After repeating $\Pi_{\mathsf{reduceDeg}}$ until the vectors have a size of at most ω, we directly verify triple $(\vec{\mathbf{x}}, \vec{\mathbf{y}}, z)$ using subprotocol $\Pi_{\mathsf{checkTriple}}$ (Fig. 8). The idea resembles that of the degree reduction, following [46]. Instead of splitting the vectors into smaller subvectors, e.g, $\vec{\mathbf{x}^1}$, we now simply split them into their single entries, e.g., x_i. Then, instead of receiving a new vector triple, we get a scalar triple (x', y', z'). This is opened to all parties to verify that $z' = x'y'$ in the final step 7.

Subprotocol $\Pi_{\mathsf{checkTriple}}$

Input: Vectors $\vec{\mathbf{x}}, \langle\vec{\mathbf{y}}\rangle$ and scalar $[z], \vec{\mathbf{x}}, \vec{\mathbf{y}} \in \mathbb{Z}_{2^{k+s}}^d, z \in \mathbb{Z}_{2^{k+s}}$ with $1 \le d \le \omega$,

Output: `Accept` if dot product is correct, `Reject` otherwise.

1. Non-interactively sample a sharing $\langle y_0 \rangle$ by each party together with $\mathcal{D}$ sampling its share $\langle y_0 \rangle_i \in \mathbb{Z}_{2^{k+s}}$.
2. Each party $\mathcal{P}_i$ computes $z_i^{j,\ell} = x_j \cdot \langle y_\ell \rangle_i$ for all $1 \le j \le d, 0 \le \ell \le d$ except for $j = \ell = 1$.
3. For $1 \le j \le d, 0 \le \ell \le d$ except for $j = \ell = 1$, set $\left[z^{j,\ell}\right] \leftarrow \Pi_{\mathsf{shareSum}}(z_1^{j,\ell}, \ldots, z_n^{j,\ell})$.
4. Compute $\left[z^{1,1}\right] = [z] - \sum_{\ell=2}^{d} \left[z^{\ell,\ell}\right]$.
5. $(\alpha^\ell, \beta^\ell)_{1 \le \ell \le d} \leftarrow \Pi_{\mathsf{rnd}}(\mathbb{Z}_{2^{k+s}}, 2d)$, let $\beta^0 = 1$.
6. Compute
$$x' = \sum_{\ell=1}^{d} \alpha^\ell x_\ell \quad \langle y' \rangle = \sum_{\ell=0}^{d} \beta^\ell \langle y_\ell \rangle \quad [z'] = \sum_{j=1}^{d}\sum_{\ell=0}^{d} \alpha^j \beta^\ell \left[z^{j,\ell}\right]$$
7. The parties reconstruct $y' = \Pi_{\mathsf{rec}}^{\mathsf{active,rerand}}(\langle y' \rangle), z' = \mathsf{m}_{z'} + \lambda_{z'}$ with $\lambda_{z'} = \Pi_{\mathsf{rec}}^{\mathsf{active,rerand}}(\langle \lambda_{z'} \rangle)$ and output `Accept` if $z' = x'y'$, `Reject` otherwise.

Figure 8: $\Pi_{\mathsf{checkTriple}}$: Final Check for a Small Triple.

Note that revealing the resulting triple might result in leakage. Contrasting [46], x' is already known to all parties. On the other hand, y' depends on $\vec{\mathbf{y}}$ which is secret-shared. To ensure security, we mask it by introducing a new private random y_0 which we use

as a mask in y'. The corresponding coefficient β^0 is statically set to 1. Finally, $z' = x'y'$ if no cheating occurs, so the reconstruction of z' can be simulated. If cheating occurs, there will be an offset, but as we will see in the soundness, this does not leak anything.

Soundness: Soundness is as in §4.2.3 with the difference that subvectors are replaced by scalars and that y_0 is used as a mask. Assume that $z^{j,\ell} = x_j \cdot y_\ell + \delta^{j,\ell}$ for accumulated errors $\delta^{j,\ell}$ chosen by the adversary. It now holds that

$$z' = \sum_{j=1}^{d}\sum_{\ell=0}^{d} \alpha^j \beta^\ell z^{j,\ell} = x' \cdot y' + \sum_{j=1}^{d} \alpha^j \sum_{\ell=0}^{d} \beta^\ell \delta^{j,\ell}.$$

Hence, there is an error δ' in the output triple $z' = x' \cdot y' + \delta'$ with $\delta' = \sum_{j=1}^{d} \alpha^j \sum_{\ell=0}^{d} \beta^\ell \delta^{j,\ell}$. The outer $\sum_{j=1}^{d} \alpha^j \ldots$ again is a random linear combination. The inner term can be rewritten to $\sum_{\ell=1}^{d} \beta^\ell \delta^{j,\ell} + \delta^{j,0}$, a random linear combination plus an error chosen by the adversary. The additional error was overlooked in [46], but this does not harm soundness as we show in §E.1. Following the same arguments as in §4.2.3, we also have that an input error of $\delta \neq 0$ (i.e., $z = \vec{\mathbf{x}} \cdot \vec{\mathbf{y}} + \delta$) with t s.t. $2^t \mid \delta, 2^{t+1} \nmid \delta$ causes some $\delta^{\ell,\ell}$ with $2^{t+1} \nmid \delta^{\ell,\ell}$ to go into the inner linear combination(s). Also, note that the error in the output triple only depends on public randomness and the errors chosen by the adversary. Hence, it does not leak additional information.

4.2.5 Delayed Verifications of Subprotocols. Recall that for reconstructing shared values (§3.1), a verify-step is required to ensure consistency (for $\Pi_{\text{rec}}^{\text{consistent}}$) or even correctness (for $\Pi_{\text{rec}}^{\text{active}}$) of the output. To fully leverage the verifications' amortization, we delay them until later in the computation. Our key intuition here is that most communication is sufficiently masked, independent of the verify steps. No masks are used in instances of Π_{rnd}, but they reveal random data, also independent of any private data. Still, other occurrences of reconstructing secret-shared data may leak additional information. E.g., we argue that reconstruction of z' in $\Pi_{\text{checkTriple}}$ does not leak information in §4.2.4, assuming that all parties have consistent sharings at the start of the soundness considerations. Hence, we always run subprotocol verifications before reconstructing values y', z' in step 7 of $\Pi_{\text{checkTriple}}$ (Fig. 8). Note that this includes the consistency check for all multiplications during the optimistic circuit evaluation. We do the necessary communication during the prior interactive step 5. After that, the only interactions are in step 7 of $\Pi_{\text{checkTriple}}$ (Fig. 8), in step 14 of Π_{verify} (Fig. 6) and for revealing the final output values after a successful verification. In all three cases, we directly run the verify-step in parallel. Details on the validity of these partially delayed verifications are provided in the full proof in §E.2.

4.2.6 Security. Our resulting protocol Π_{MPC} runs all multiplications optimistically, then uses Π_{verify} (Fig. 6), and if it succeeds, reveals the output values to the parties (§4.1). Simulation of the protocol works by composing the subsimulators that we have defined alongside all required subprotocols before, and we provide the full formal details in §E.2. The main challenge is to show soundness of our proof and to set parameter s accordingly. Recall that the steps of our verification iteratively compute random linear combinations, where we show that each step's output has an error that also is a random linear combination of all input errors. The finally resulting triple is opened and the protocol aborts if it is incorrect. Hence, it remains to be shown that with the right choice of parameter s, the chance that errors introduced by an adversary vanish over $\mathbb{Z}_{2^{k+s}}$ while iteratively doing random linear combinations is negligible. Fortunately, a thorough analysis for selection of s is already given in [46]. Still, additional care is required to account for the use of PRFs (§3.4) instead of truly random coefficients in [46], as [46] left this open, leading to higher cost. We defer the details to §E.1. With the analysis regarding s and the simulation proof, we get:

Theorem 1. *Let C be a circuit, m be the number of multiplications in the circuit, and $\omega \in \mathbb{N}$ be some compression factor. For $T = 2 \cdot \lceil \log_\omega(3m) \rceil + 1$ and $s \geq \max(3T, \sigma + T(\frac{1}{2} + \log_2(\frac{5}{2} + \frac{3\sigma}{T})))$, if $T \leq \sigma$, then the following holds: Π_{MPC} computes C on the inputs provided by all parties with security with abort in the presence of an active adversary corrupting any subset of parties $\mathcal{P}_j$.*

4.2.7 Complexity. If we instantiate all instances of Π_{rec} using $\Pi_{\text{rec}}^{\text{king}}$, i.e., with the $\mathcal{P}_{\text{king}}$-strategy (§3.1), then in the semi-honest base protocol, each multiplication costs only one ring ($\mathbb{Z}_{2^k}$) element communication offline and $2n-2$ ring elements online, so ≈ 2 elements per party. Thorough analysis of our verification Π_{verify} (Fig. 6) yields an asymptotic complexity of $O(\sigma k + s + n\kappa)$ bits offline and $O(\log m \cdot (\omega^2 \cdot (k+s) + \kappa) + \sigma(k+s) + n\kappa)$ bits online per party. The majority stems from $O(\log m)$ iterations for reducing the vector dimensions, each using $O(\omega^2)$ instances of Π_{shareSum} and one Π_{rnd}. Note that only the cost of batched instances of verify grows with the number of parties n, where $\mathcal{D}$ distributes a constant number of commitments offline, whereas online, parties need to do a pairwise exchange of hashes. Discarding σ, κ as constants yields $O(k+s+n)$ bits offline and $O(n + \log m \cdot \omega^2 \cdot (k+s))$ bits online per party. Using $s \in O(\log m)$ (Theorem 1), we have zero amortized communication per multiplication. Computation is dominated by $O(\omega^2)$ dot products per iteration, where computation (not communication) scales with the vector dimension. This yields $(3m + 3m/\omega + 3m/\omega^2 + \ldots) \cdot O(\omega^2) = O(m \cdot \omega^2)$ multiplications over $\mathbb{Z}_{2^{k+s}}$.

Note that [12] also supports a constant round proof by choosing $\omega \in O(\sqrt{m})$, yielding higher communication $O(\sqrt{m})$ instead of $O(\log m)$. This is not possible for us due to the ω^2-term, which would render our communication linear in m, as in [46].

5 Evaluation

5.1 Analytical Comparison with [12, 46]

As seen in §4.2.7, our communication complexity is $O(n + \log m \cdot \omega^2 \cdot (k+s))$ bits per party, for m multiplications, compression factor ω, bit-width k, extended bit-width $k+s$, and n parties. If instead of our novel proof, we would just translate the proof of [46] to our setting, online communication per party would grow to $O(n \cdot \log m \cdot \omega^2 \cdot (k+s))$ bits. This is as [46] would let each party separately prove that it contributes the correct part to $[\Gamma]$ in §4.2.2, using n proofs instead of our single, collective one. Furthermore, [46] samples $O(\log m \cdot \omega^2)$ public random values separately, whereas we use a PRF (§5), reducing this to $O(\log m)$ many seeds.

Using [12] on rings is possible, while not tailored to that. Yet, this relies on *Galois ring extensions*, i.e., calculation on the ring $\mathbb{Z}_{2^k}[X]$ modulo a polynomial of degree $d = \sigma + O(\log m)$ for statistical security σ. This yields an online communication of $O(n \cdot \log m \cdot \omega \cdot$

kd) bits per party, again suffering from n separate proofs and the extension factor d, whereas complexity now only scales linearly in the compression factor ω instead of quadratically.

In Fig. 9, we compare the concrete communication between our verification and those from [46] (translated to our setting) and [12] for different numbers of parties n. We consider $m = 10^6$ multiplications on 32-bit integers ($k = 32$), statistical security parameter $\sigma = 40$, computational security parameter $\kappa = 128$, and minimal sufficient s. For [12], we use the optimal compression factor $\omega = 2$. For our approach and [46], we use $\omega = 3$, yielding higher communication but being required for security to ensure $T \leq \sigma$ in §4.2.6.

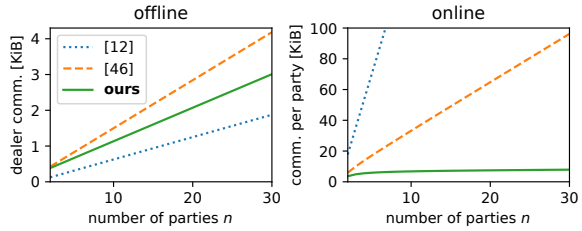


Figure 9: Verification communication overhead (per party) for a circuit with $m = 10^6$ multiplications.

Our online communication per party is already better than both other approaches for $n = 2$, but especially scales significantly better with more parties, given a low coefficient for the only linear term, yielding almost constant communication. Both other approaches scale visibly linearly, with [12] requiring > 497 KiB communication per party for $n = 30$. Offline communication, i.e., the communication complexity of the dealer $\mathcal{D}$, scales linear with n, but is substantially smaller than the online communication per party. The semi-honest circuit computation phase costs 3.8 MiB offline and 3.8–7.4 MiB online communication per party. Hence, the additional verification communication still amortizes well for all verification approaches.

5.2 Experimental Evaluation

5.2.1 Implementation. We implement our protocol in C++, using the code of [38] as a basis for circuit representation, communication layer, PRFs, etc., and make it publicly available at https://encrypto.de/code/TheOneProof. This, to the best of our knowledge, is the first public implementation of zk-FLIOPS in the $(n + 1)$-party setting with arbitrary n. Our implementation enables building arbitrary arithmetic circuits, on which it then runs the semi-honest circuit computation phase, and our novel verification. We also support to run the semi-honest phase without our checks, yielding the passively secure baseline protocol that we will compare to. We integrate optional parallelization using OpenMP. For $\mathbb{Z}_{2^{k+s}}$, we also utilize the Clang/GCC extensions for 128-bit integers. For heuristical instantiation of the ROM H, we use SHA256.

5.2.2 Environment and Parameters. We benchmark on a single server with an Intel Core i9-7960X CPU @ 2.8 GHz with 32 threads, 128 GB of DDR4 RAM @ 2666 MHz.[7] On that, we locally simulate a network with up to 30+1 machines, based on the network emulation of NEON [41]. We consider a LAN setting (1 ms RTT, 1 GBit/s bandwidth) and a WAN setting (100 ms RTT, 100 MBit/s bandwidth).

As security parameters, we use $\sigma = 40$ (statistical) and $\kappa = 128$ (computational). We use $\mathbb{Z}_{2^k} = \mathbb{Z}_{2^{32}}$ as computation domain because 32-bit integers are a popular choice in classical, non-secure computer programs. For $\mathbb{Z}_{2^{k+s}}$, we use $\mathbb{Z}_{2^{128}}$, i.e., $s = 96$ which is sufficiently large for all benchmarks. We mostly focus on a low number of parties $n = 3$; as will be shown in §5.2.6, the observed behavior for $n = 3$ is similar to that for larger values n.

Our benchmarks vary in the number of multiplications m in a circuit of a certain multiplicative depth. We use a synthetic circuit in which we evenly distribute the m multiplications across different layers according to the desired depth. Then, after each layer of multiplications, we add the same amount of additions to achieve a balanced distribution between interactive and non-interactive operations. We do not include input and output phases in our experiments, given that our focus is to investigate the performance of our verification for multiplications which only concerns the circuit evaluation itself. By the passively secure baseline, we refer to the protocol obtained by using the main circuit evaluation phase of our protocol while removing all verification steps. We average over 10 iterations per experiment. Our verification iteratively decreases vector sizes $\log_\omega(3m)$ times while each iteration's communication scales quadratically with ω. We experimentally establish $\omega = 20$ as a good tradeoff between communication and rounds.

5.2.3 Optimizations. Recall from §3.1 that shared values can be reconstructed using either the $\mathcal{P}_{\text{king}}$-strategy, requiring two rounds but constant communication per party, or using broadcasting, requiring a single round but communication linear in n per party. This is a tradeoff between communication and round complexity. For $n = 2$, both variants require sending 2 ring-elements. Hence, we always use broadcasting there. Otherwise, we observe that some steps of the verification (§4.2), especially instances of Π_{rnd} to sample random coefficients, require low, constant communication. In such cases, we use broadcasting to save one communication round. Otherwise, we use the $\mathcal{P}_{\text{king}}$-strategy. Compared to exclusively using the $\mathcal{P}_{\text{king}}$-strategy, this causes only a minor increase in communication.

5.2.4 Overhead over Passively-Secure Computation. Our core question is how much overhead our verification adds on top of the passively secure baseline protocol. In Tab. 3, we provide our experimental results for $m = 1$ million multiplications with different circuit depths. As can be seen from this table, the communication overhead is very small at only 0.7%. Yet, the iterative nature of our verification yields a noticeable overhead of 25 additional communication rounds which, depending on the circuit depth, amortizes well only for deeper circuits. Investigating the tradeoff between communication, rounds, and the required computation, we proceed to analyze the resulting run times. Local computation, if not parallelized, is the bottleneck in a LAN setting, yielding an up to 4× increase of total run time. This can be explained by our verification, while not using expensive field operations, still handling large dot products with computation scaling linearly with the number of multiplications. This is similar as in [46], reporting an increase of run time by even 8.4× in their respective setting. By parallelizing over 8 threads, we are able to decrease the verification overhead to +149%, yielding actively secure computation at less than 3× the

[7] All experiments allocated < 5.4 GiB memory per party, and for 10^6 multiplications, < 1.1 GiB were allocated per party.

Table 3: Comparison between the passively secure baseline and our actively secure protocol, using our novel verification, with the verification's total overhead over the baseline (for communication, regarding the total value across all n parties and the dealer). Benchmark for $m = 10^6$ multiplications and $n = 3$ parties. Communication and run times provided as offline + online. Offline communication is what is sent by the dealer, online communication is reported per party.

circuit depth		communication (MiB)	online rounds	run time (s) — 1 thread		run time (s) — 8 threads	
				LAN	WAN	LAN	WAN
10	passive	3.81 + 5.09	20	0.13 + 0.11	0.91 + 1.89	0.14 + 0.09	0.91 + 1.80
	active	3.82 + 5.13	45	0.27 + 0.61	1.07 + 3.78	0.30 + 0.28	1.07 + 3.27
	overhead	+0.7%	+125.0%	+268.7%	+73.4%	+149.2%	+60.2%
30	passive	3.81 + 5.09	60	0.12 + 0.11	0.93 + 3.71	0.14 + 0.08	0.94 + 3.64
	active	3.82 + 5.13	85	0.26 + 0.60	1.07 + 5.58	0.29 + 0.27	1.06 + 5.09
	overhead	+0.7%	+ 41.7%	+273.7%	+43.2%	+147.9%	+34.3%
100	passive	3.81 + 5.09	200	0.13 + 0.18	0.92 + 10.77	0.14 + 0.14	0.92 + 10.69
	active	3.82 + 5.13	225	0.26 + 0.67	1.06 + 12.64	0.29 + 0.33	1.06 + 12.14
	overhead	+0.7%	+ 12.5%	+204.2%	+17.2%	+118.6%	+13.7%

run time of passively secure MPC.[8] In a WAN setting, the impact of local computation decreases so that our verification comes at only +73% overhead for circuit depth 10, decreasing down to +17% for depth 100. Using 8 threads decreases this further, down to below +14% for depth 100. In conclusion, our verification enables achieving active security at a low cost, especially well suited for minimizing communication volume, or overall run time in a WAN.

5.2.5 Scalability w.r.t. Circuit Size. Scalability with the number of multiplications m is investigated in Tab. 4 for circuit depth 30. The communication overhead of our verification decreases for increasing m, amortizing away the additional cost per multiplication. When used over sufficiently many m, this overhead becomes insignificant, amounting to only 0.7% for $m = 1$M or even 0.1% for $m = 10$M multiplications. The round complexity slightly increases as it scales linearly with $\log_\omega m$. Regarding run times, the verification adds less overhead for fewer multiplications in the LAN setting, primarily given the decrease in local computation workload. In a WAN, we observe that the low amortized communication overhead also leads to lower overhead on very big circuits, reaching active security for $m = 10$M multiplications at only 26% run time overhead.

5.2.6 Scalability w.r.t. the Number of Parties n. In Fig. 10, we investigate scalability with n. Communication per party and dealer remain mostly constant, with the verification adding negligible overhead over the passively secure baseline. Regarding low n, one passively secure multiplication costs on average $(2n-2)/n$ ring elements per party, explaining an initial increase up to some constant bound. The discontinuity in run time for $n = 2$ is due to each reconstruction using 1 instead of 2 rounds there (§5.2.3). While we have seen in §5.1 that the dealer's communication for verification scales linearly in n, this is unnoticeable here as the passively secure part is dominant. Run time still increases for more parties. We attribute that to a higher complexity of handling incoming and outgoing traffic, given that this needs to account for an increasing number of network connections, while local computation specific to the protocol remains the same. The verification overhead remains comparable for different n, being 211%–274% in LAN and 43%–57% in WAN.

[8] Our code parallelizes the passively secure baseline and the verification. Yet, the impact on verification is higher as this has a significantly higher computation workload.

Table 4: Actively secure protocol vs. the passive baseline for m multiplications. Results include offline and online. Communication is over all parties and dealer. Benchmark for circuit depth 30 and $n = 3$ parties, with 8 threads each.

m		comm. (MiB)	online rounds	run time (s)	
				LAN	WAN
1K	passive	0.02	60	0.04	3.19
	active	0.08	79	0.05	4.17
	overhead	+298.9%	+31.7%	+ 37.9%	+30.9%
10K	passive	0.19	60	0.04	3.19
	active	0.27	82	0.06	4.34
	overhead	+ 41.6%	+36.7%	+ 49.7%	+35.8%
100K	passive	1.91	60	0.05	3.55
	active	2.01	85	0.10	4.89
	overhead	+ 5.4%	+41.7%	+ 93.3%	+37.8%
1M	passive	19.07	60	0.22	4.58
	active	19.20	85	0.56	6.15
	overhead	+ 0.7%	+41.7%	+147.9%	+34.3%
10M	passive	190.74	60	1.99	13.87
	active	190.87	88	5.17	17.51
	overhead	+ 0.1%	+46.7%	+159.7%	+26.2%

5.2.7 Remark on zk-FLIOPS with Galois Ring Extensions. While [12] considers the same $(n+1)$-party setting as our work, it uses Galois ring extensions to support $\mathbb{Z}_{2^k}$ while its construction is designed primarily for finite fields, and provides no implementation. To the best of our knowledge, there also exists no other implementation of MPC using zk-FLIOPS in the $n+1$-party setting. Still, besides more expensive verification, especially w.r.t. n as discussed in §5.1,

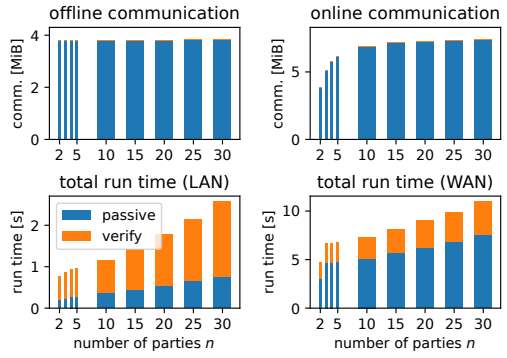


Figure 10: Scalability for varying number of parties n, communication split into offline (sent by $\mathcal{D}$) and online (sent per party). Bars are split into the passively secure baseline (blue) and verification overhead (orange). Benchmark for $m = 10^6$ multiplications, circuit depth 30, and 1 thread per party.

computation on a Galois ring extension adds significant overhead as observed in, e.g., [23, 46]. In a three-party honest majority setting, with $m = 10^6$ and depth 10, [46] observed a run time overhead of 43.4× in LAN and 4.5× in WAN over a zk-FLIOP tailored for rings.

5.2.8 Comparison with Asterisk [38]. In Tab. 5, we compare our protocol with Asterisk [38]. For fairness, we benchmark the implementation of [38] in our environment, using 40 bit values as [38] requires $k \geq \sigma$. Our protocol improves over Asterisk in all metrics. We improve the offline communication by ≈ 4× and online communication by 1.2×, because we do not use MACs, but a verification that amortizes well with the number of multiplications. Regarding run time, while the verification of Asterisk runs in constant rounds, we have faster local computation as we do not require field arithmetic. We improve offline run time by ≈ 30× in the computation-bound LAN and ≈ 9× in a WAN, and improve online run time by ≈ 3.5× in LAN and ≈ 1.3× in WAN. While Asterisk could also be translated to rings, using the SPD$\mathbb{Z}_{2^k}$ trick [22] to decrease computational workload, this would require carrying out each individual multiplication over $\mathbb{Z}_{2^{k+\sigma}}$. That would further increase communication by another 80% here, including the online phase.

5.2.9 Additional Benchmarks and Comparisons. In the appendix, we provide further benchmarks. A generalization of our protocol to dot products is introduced and benchmarked in §F. We compare our protocol to [46] and SPD$\mathbb{Z}_{2^k}$ [22] in §G.

6 Conclusion

Our work shows how to efficiently achieve active security for a dishonest majority, assisted by a trusted dealer, with a very little communication overhead over passive security. We unify and optimize previous approaches that were either for dishonest majority or optimized for rings $\mathbb{Z}_{2^k}$, but not both. Hence, we create the first protocol for dishonest majority with a dealer that is optimized for rings. Our core technical contribution is a new single distributed zero-knowledge (DZK) proof that verifies correctness of the computation for all parties, reaching good scalability to many parties for the first time in the domain of DZK for actively secure MPC.

Table 5: Comparison of our protocol with Asterisk [38], offline + online phase (communication per party) for $n = 3$ parties and $m = 10^6$ multiplications. We use 1 thread per party as the code of Asterisk has no parallelization.

circ. depth	protocol	communication (MiB)	run time (s) LAN	run time (s) WAN
10	[38]	15.26 + 6.36	8.25 + 2.20	9.91 + 5.12
	ours	3.82 + 5.13	0.27 + 0.61	1.07 + 3.78
	impr.	4.0× + 1.2×	30.1× + 3.6×	9.2× + 1.4×
30	[38]	14.62 + 6.36	8.03 + 2.19	9.67 + 7.23
	ours	3.82 + 5.13	0.26 + 0.60	1.07 + 5.58
	impr.	3.8× + 1.2×	31.1× + 3.6×	9.0× + 1.3×
100	[38]	14.40 + 6.36	7.89 + 2.12	9.53 + 16.33
	ours	3.82 + 5.13	0.26 + 0.67	1.06 + 12.64
	impr.	3.8× + 1.2×	29.8× + 3.2×	9.0× + 1.3×

We also provide the first DZK implementation for arbitrary many parties and showcase practicality of our protocol, improving over state-of-the-art solutions.

Acknowledgments

The authors thank Nora Khayata for providing helpful references for arithmetic circuit compilers.

This project received funding from the European Research Council (ERC) under the European Union's research and innovation programs Horizon Europe (PRIVTOOLS/101124778) and Horizon 2020 (PSOTI/850990). It was co-funded by the Deutsche Forschungsgemeinschaft (DFG) within SFB 1119 CROSSING/236615297. It was supported by the German Federal Ministry of Research, Technology and Space (BMFTR), and the Hessian Ministry of Science and Research, Arts and Culture (HMWK) within the National Research Center for Applied Cybersecurity ATHENE.

References

[1] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. 2011. How to Garble Arithmetic Circuits. In *FOCS*. IEEE.
[2] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. 2016. High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority. In *CCS*. ACM.
[3] Gilad Asharov, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Ariel Nof, Benny Pinkas, Katsumi Takahashi, and Junichi Tomida. 2022. Efficient Secure Three-Party Sorting with Applications to Data Analysis and Heavy Hitters. In *CCS*. ACM.
[4] Donald Beaver. 1991. Efficient Multiparty Protocols Using Circuit Randomization. In *CRYPTO*. Springer.
[5] Mihir Bellare and Phillip Rogaway. 1993. Random Oracles are Practical: A Paradigm for Designing Efficient Protocols. In *CCS*. ACM.
[6] Aner Ben-Efraim, Michael Nielsen, and Eran Omri. 2019. Turbospeedz: Double Your Online SPDZ! Improving SPDZ Using Function Dependent Preprocessing. In *ACNS*. ACM.
[7] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. 2019. Zero-Knowledge Proofs on Secret-Shared Data via Fully Linear PCPs. In *CRYPTO*. Springer.
[8] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. 2020. Efficient Pseudorandom Correlation Generators from Ring-LPN. In *CRYPTO*. Springer.
[9] Elette Boyle, Niv Gilboa, Matan Hamilis, Yuval Ishai, and Ariel Nof. 2025. Preprocessing for Life: Dishonest-Majority MPC with a Trusted or Untrusted Dealer. In *S&P*. IEEE.
[10] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. 2019. Practical Fully Secure Three-Party Computation via Sublinear Distributed Zero-Knowledge Proofs. In

CCS. ACM.
[11] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. 2020. Efficient Fully Secure Computation via Distributed Zero-Knowledge Proofs. In *ASIACRYPT*. Springer.
[12] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. 2021. Sublinear GMW-Style Compiler for MPC with Preprocessing. In *CRYPTO*. Springer.
[13] Lennart Braun, Daniel Demmler, Thomas Schneider, and Oleksandr Tkachenko. 2022. MOTION – A Framework for Mixed-Protocol Multi-Party Computation. *ACM TOPS* (2022).
[14] Lennart Braun, Moritz Huppert, Nora Khayata, Thomas Schneider, and Oleksandr Tkachenko. 2023. FUSE – Flexible File Format and Intermediate Representation for Secure Multi-Party Computation. In *ASIACCS*. ACM.
[15] Andreas Brüggemann and Thomas Schneider. 2026. When Trying to Catch Cheaters Breaks the MPC: Breaking and Fixing Delayed Consistency Checks in Trident, Fantastic Four, SWIFT, and Quad. In *EUROCRYPT*. Springer.
[16] Andreas Brüggemann, Ariel Nof, and Thomas Schneider. 2026. One Proof to Rule Them All: Practical, Sublinear Verification for Actively Secure MPC on $\mathbb{Z}_{2^k}$ with Dishonest Majority and a Dealer. In *CCS*. ACM.
[17] Andreas Brüggemann, Oliver Schick, Thomas Schneider, Ajith Suresh, and Hossein Yalame. 2024. Don't Eject the Impostor: Fast Three-Party Computation With a Known Cheater. In *S&P*. IEEE.
[18] Niklas Büscher, Daniel Demmler, Stefan Katzenbeisser, David Kretzmer, and Thomas Schneider. 2018. HyCC: Compilation of Hybrid Protocols for Practical Secure Computation. In *CCS*. ACM.
[19] Ran Canetti. 2000. Security and Composition of Multiparty Cryptographic Protocols. *Journal of Cryptology* 13, 1 (2000).
[20] Harsh Chaudhari, Ashish Choudhury, Arpita Patra, and Ajith Suresh. 2019. ASTRA: High Throughput 3PC over Rings with Application to Secure Prediction. In *CCSW@CCS*. ACM.
[21] Edward Chen, Jinhao Zhu, Alex Ozdemir, Riad S. Wahby, Fraser Brown, and Wenting Zheng. 2023. Silph: A Framework for Scalable and Accurate Generation of Hybrid MPC Protocols. In *S&P*. IEEE.
[22] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. 2018. SPDZ2k: Efficient MPC mod 2^k for Dishonest Majority. In *CRYPTO*. Springer.
[23] Anders Dalskov, Daniel Escudero, and Marcel Keller. 2021. Fantastic Four: Honest-Majority Four-Party Secure Computation with Malicious Security. In *USENIX Security*.
[24] Ivan Damgård, Daniel Escudero, Tore Kasper Frederiksen, Marcel Keller, Peter Scholl, and Nikolaj Volgushev. 2019. New Primitives for Actively-Secure MPC over Rings with Applications to Private Machine Learning. In *S&P*. IEEE.
[25] Ivan Damgård and Jesper Buus Nielsen. 2007. Scalable and Unconditionally Secure Multiparty Computation. In *CRYPTO*. Springer.
[26] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. 2012. Multiparty Computation from Somewhat Homomorphic Encryption. In *CRYPTO*. Springer.
[27] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2014. Ad-Hoc Secure Two-Party Computation on Mobile Devices using Hardware Tokens. In *USENIX Security*.
[28] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY – A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In *NDSS*.
[29] Henri Dohmen, Robin William Hundt, Nora Khayata, and Thomas Schneider. 2025. SEEC: Memory Safety Meets Efficiency in Secure Two-Party Computation. In *ASIACCS*. ACM.
[30] Radhika Garg and Xiao Wang. 2025. Smaug: Modular Augmentation of LLVM for MPC. In *S&P*. IEEE.
[31] Radhika Garg, Kang Yang, Jonathan Katz, and Xiao Wang. 2024. Scalable Mixed-Mode MPC. In *S&P*. IEEE.
[32] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *STOC*. ACM.
[33] Vipul Goyal, Hanjun Li, Rafail Ostrovsky, Antigoni Polychroniadou, and Yifan Song. 2021. ATLAS: Efficient and Scalable MPC in the Honest Majority Setting. In *CRYPTO*. Springer.
[34] Vipul Goyal, Yifan Song, and Chenzhi Zhu. 2020. Guaranteed Output Delivery Comes Free in Honest Majority MPC. In *CRYPTO*. Springer.
[35] Tim Heldmann, Thomas Schneider, Oleksandr Tkachenko, Christian Weinert, and Hossein Yalame. 2021. LLVM-Based Circuit Compilation for Practical Secure Computation. In *ACNS*. Springer.
[36] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. 2010. TASTY: Tool for Automating Secure Two-partY computations. In *CCS*. ACM.
[37] Y. Huang. 2012. *Practical Secure Two-Party Computation*. PhD dissertation. University of Virginia. http://yhuangpress.files.wordpress.com/2014/02/dissertation.pdf
[38] Banashri Karmakar, Nishat Koti, Arpita Patra, Sikhar Patranabis, Protik Paul, and Divya Ravi. 2024. Asterisk: Super-fast MPC with a Friend. In *S&P*. IEEE.
[39] Jonathan Katz and Yehuda Lindell. 2014. *Introduction to Modern Cryptography* (second ed.). CRC Press.
[40] Marcel Keller. 2020. MP-SPDZ: A Versatile Framework for Multi-Party Computation. In *CCS*. ACM.
[41] Andreas Klinger, Vincent Ehrmanntraut, and Ulrike Meyer. 2024. Estimating the Runtime and Global Network Traffic of SMPC Protocols. In *CODASPY*. ACM.
[42] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. 2021. SWIFT: Superfast and Robust Privacy-Preserving Machine Learning. In *USENIX Security*.
[43] Benjamin Levy, Muhammad Ishaq, Benjamin Sherman, Lindsey Kennard, Ana L. Milanova, and Vassilis Zikas. 2023. COMBINE: COMpilation and Backend-INdependent vEctorization for Multi-Party Computation. In *CCS*. ACM.
[44] Chongrong Li, Pengfei Zhu, Yun Li, Zhanpeng Guo, Jingyu Li, Yuncong Hu, Zhicong Huang, and Cheng Hong. 2026. Secure Lookup Tables: Faster, Leaner, and More General. In *S&P*. IEEE.
[45] Yun Li, Daniel Escudero, Yufei Duan, Zhicong Huang, Cheng Hong, Chao Zhang, and Yifan Song. 2024. Sublinear Distributed Product Checks on Replicated Secret-Shared Data over $\mathbb{Z}_{2^k}$ Without Ring Extensions. Cryptology ePrint Archive, Paper 2024/700.
[46] Yun Li, Daniel Escudero, Yufei Duan, Zhicong Huang, Cheng Hong, Chao Zhang, and Yifan Song. 2024. Sublinear Distributed Product Checks on Replicated Secret-Shared Data over Z2k Without Ring Extensions. In *CCS*. ACM.
[47] Jake Loftus and Nigel P. Smart. 2011. Secure Outsourced Computation. In *AFRICACRYPT*. Springer.
[48] Jianqiao Cambridge Mo, Karthik Garimella, Austin Ebel, and Brandon Reagen. 2025. ABLE: Optimizing Mixed Arithmetic and Boolean Garbled Circuit. Cryptology ePrint Archive, Paper 2025/048.
[49] Payman Mohassel and Peter Rindal. 2018. ABY3: A Mixed Protocol Framework for Machine Learning. In *CCS*. ACM.
[50] Philipp Muth and Stefan Katzenbeisser. 2022. Assisted MPC. Cryptology ePrint Archive, Paper 2022/1453.
[51] Alex Ozdemir, Fraser Brown, and Riad S. Wahby. 2022. CirC: Compiler Infrastructure for Proof Systems, Software Verification, and More. In *S&P*. IEEE.

A Open Science

All artifacts of this paper are publicly available under the MIT license on GitHub (https://encrypto.de/code/TheOneProof) and, alternatively, Zenodo (https://doi.org/10.5281/zenodo.22011889). These repositories both contain the following parts:

- Implementation of the passively secure baseline protocol and our new actively secure variant, including the novel zk-FLIOP implementation, and protocol variants from §F supporting dot products.
- Scripts to locally emulate realistic network behavior, based on those provided by [41].
- Benchmark implementations.
- Documentation on building, executing, and benchmarking the protocols.
- An optional Dockerfile for testing in a container.
- Scripts to re-benchmark the original implementations of related works [22, 38, 46] for the comparisons in §5.2.8 and §G, including small changes and fixes in the code to ensure a fair comparison (e.g., change the integer resolution and apply our network emulation tooling).

Our benchmarks use synthetic circuits as introduced in §5.2.2. All inputs are set to synthetic values for benchmarking, because MPC protocols run oblivious of concrete input values as they need to be data-oblivious, yielding consistent performance on any input.

B Note on Ethical Considerations

This work raises no ethical concerns. It introduces a novel MPC protocol to evaluate generic arithmetic circuits. When running computation on sensitive data of multiple parties, our protocol provides the opportunity to preserve privacy of the data. For our evaluation, we consider synthetic circuits without proposing a concrete use for a certain application. Furthermore, benchmarks are run on fully synthetic data. Overall, we identify no ethical concerns.

C Additional Building Blocks

C.1 Inputs

A party $\mathcal{P}_j$ can $[\cdot]$-share its private input x as follows, using a subprotocol Π_{input}: First, $\mathcal{D}, \mathcal{P}_j$ sample $\lambda_x^j \in_R \mathbb{Z}_{2^k}$. Shares λ_x^ℓ for $\ell \neq j$ are set to 0 by the corresponding parties. Then, $\mathcal{P}_j$ sends $\mathsf{m}_x = x - \lambda_x^j$ to all parties. Finally, all pairs of parties exchange H of their m_x and `abort` in case of a mismatch. This is batched over many instances.
Security: The subsimulator is given a value x to share if $\mathcal{P}_j$ is not corrupted and then follows the protocol specification. Note that an adversary is unable to distinguish between subsimulation/protocol execution with different x as the transmitted value is masked by λ_x^j, unknown to the adversary. If $\mathcal{P}_j$ is corrupted, the subsimulator also follows the protocol specification. In this case, note that if $\mathcal{P}_j$ provides inconsistent m_x, the hash-based check will lead to an `abort` except for negligible probability. As an additional property, note that from consistent m_x and knowledge of λ_x^i, the subsimulator can derive the plaintext value x.

C.2 Outputs

For reconstruction value x of a sharing $[x]$, the intuition is to simply use $\Pi_{\mathsf{rec}}^{\mathsf{active}}$ on $\langle \lambda_x \rangle$ to learn λ_x. Then, we can set $x = \mathsf{m}_x + \lambda_x$. Yet, for security, we need to rerandomize $\langle \lambda_x \rangle$ before. Generally, to reconstruct some $\langle a \rangle$, we proceed as follows: $\mathcal{P}_j, \mathcal{D}$ for $1 \leq j < n$ non-interactively sample random shares $\langle r \rangle_j$. Then, $\mathcal{D}$ computes and sends to $\mathcal{P}_n$ share $\langle r \rangle_n$ s.t. the sum over all shares is $r = 0$. Hence, $\langle r \rangle$ is a random sharing subject to $r = 0$. We simply compute $\langle a + r \rangle$, reconstruct this instead of $\langle a \rangle$ using $\Pi_{\mathsf{rec}}^{\mathsf{active}}$, and we call the resulting protocol $\Pi_{\mathsf{rec}}^{\mathsf{active,rerand}}$.
Security: Security is similar as for $\Pi_{\mathsf{rec}}^{\mathsf{active}}$ (§3.1). Yet, we let the subsimulator here receive as input a' which is the anticipated output and matches the real world execution (it is up to the context of the simulator calling this subsimulator to determine a'). The subsimulator uses $\langle a \rangle$ to derive the actually represented value a', having access to all shares through $\mathcal{D}$. Then, it simulates the computation of $\langle r \rangle$, following the description above. It identifies one honest party $\mathcal{P}_h$ and modifies $\langle r \rangle_h$ by adding $a' - a$, yielding $\langle r' \rangle$. Then, it proceeds to follow the protocol specification. Regarding simulation of $\Pi_{\mathsf{rec}}^{\mathsf{active}}$, note that it runs on $\langle a + \langle r' \rangle \rangle$ which is not a sharing where all shares are random. Still, we observe that the shares are random subject to summing up to $a + r' = a'$ which suffices to render the reconstruction secure. Depending on the instantiation of Π_{rec}, corrupted parties only receive a' for an honest $\mathcal{P}_{\mathsf{king}}$ or $\langle a \rangle_j + \langle r' \rangle_j$ for all honest parties $\mathcal{P}_j$. The first case is the same between real and ideal execution. For the second case, through $\langle r' \rangle$ only the sum over all values is not random, but fully determined by a' and the shares held by the corrupted parties, again yielding indistinguishability between real and ideal execution.

D Instantiating the Dealer: Possible Directions

Our work uses a trusted dealer. The straightforward instantiation, also used in our implementation, is to use one additional semi-honest and non-colluding party without inputs or outputs to act as the dealer. As an extension, the dealer may be implemented using trusted hardware. We now briefly present alternatives and their expected costs.

Untrusted Dealer. If we used an untrusted dealer instead, such dealer can send incorrect correlations as in [9, 50]. We stress that such a model does still not allow any collusion between the dealer and any of the online parties. The dealer provides two kinds of correlations:

Multiplication correlations: Observe that the dealer can provide incorrect $\langle \gamma_{xy} \rangle$ in Π_{mult} (§3.2) while we need that $\gamma_{xy} = \lambda_x \lambda_y$. This requirement forms a degree-2 statement, so it can be verified using a DZK-based protocol with sublinear communication, where the dealer is the prover and the online parties are the verifiers [9, 12]. For $n = 2$ parties, this is done by Auxiliator [17] for semi-honest parties using the exact same baseline protocol, so both protocol designs could simply be merged.

Verification correlations: These include all the data sent by the dealer as a verifier for our proof. A crucial point here is that malformed correlations can only cause online parties to abort a correct computation. Nevertheless, in this case, the parties do not gain any information. Moreover, from inspecting the protocol, we see that the result of the verification is independent of private inputs, and so a selective-failure attack is not possible. We thus conclude that it is not required to verify these correlations.

Distributing the Dealer. This is naturally significantly more expensive, as run an MPC protocol for computing the dealer functionality. For example, the dealer could be represented as a circuit, and then any MPC protocol to compute this circuit is run. The cost of such a protocol can be estimated by the number of multiplications in the dealer's circuit. In our case, there are m triples to compute, and further ones in the dot products in the different stages of our verification protocol, so the dealer computes additional $O(m)$ multiplications over $\mathbb{Z}_{2^{k+s}}$. Hence, the overall size of the circuit is $O(m)$. We leave optimization and the implementation of such a distributed dealer as future work.

E Additional Proofs

E.1 Soundness Analysis

For the soundness analysis, [46] introduces the interactive game $\mathcal{G}ame(k, s, T)$ shown in Fig. 11. It is parametrized by the number of bits k, additional bits s used for the extended ring $\mathbb{Z}_{2^{k+s}}$, and some number of game rounds $T \in \mathbb{N}$. The game uses notation $\mathrm{Po2}(x)$ which for $x \in \mathbb{Z}_{2^{k+s}}$ and $x \neq 0$ is the number of 2-factors in x, i.e., the largest integer u s.t. $2^u | x$. Furthermore, $\mathrm{Po2}(0) := k + s$. It is used to analyze how an error introduced by the adversary in the protocol propagates throughout the iterative random linear combination steps. If an error is introduced, by additional errors and multiplication by constant coefficients, its number of 2-factors may increase to $k + s$, i.e., it overflows out of $\mathbb{Z}_{2^{k+s}}$. Yet, for soundness we require that the final triple (x', y', z') in $\Pi_{\mathsf{checkTriple}}$ is incorrect in $\mathbb{Z}_{2^{k+s}}$, given that an error was introduced at some point of time. We need to prove that the likelihood of having a correct final triple while an error modulo 2^k was introduced is negligible in σ. In [46], it is proven that an adversary for their protocol with advantage p of causing honest parties to wrongfully accept during verification can be reduced to an adversary $\mathcal{A}_g$ with advantage p if $T = 2 \log_\omega(3m) + 1$.[9] As [46] also elaborates, the random selection in step 4 of $\mathcal{G}ame$

[9] Parameters changed according to our exact statement to prove given that the initial vectors have dimension m.

$\mathcal{G}ame(k, s, T)$ [46]

$E_0 = k - 1$
for $i = 1, \ldots, T$ **do**
  $e_i, c_i \leftarrow \mathcal{A}_g$ s.t. $e_i, c_i \in \mathbb{Z}_{2^{k+s}}, \mathsf{Po2}(e_i) \leq E_{i-1}$
  $r_i \in_R \mathbb{Z}_{2^{k+s}}$, hand r_i to $\mathcal{A}_g$
  $E_i = \mathsf{Po2}(r_i \cdot e_i + c_i)$
return win if $E_T = k + s$ **else return fail**

$\mathcal{G}ame_0'(k, s, T, \sigma, \omega, d)$

$E_0 = k - 1$
for $i = 1$ **do**
  $(e_i^j)_{1 \leq j \leq \sigma} \leftarrow \mathcal{A}_g$ s.t. $e_i^j \in \mathbb{Z}_{2^{k+s}} \; \forall\, 1 \leq j \leq \sigma$,
    $\exists j' : \mathsf{Po2}(e_i^{j'}) \leq E_{i-1}$
  $\mathsf{rs}_i \in_R \{0,1\}^\kappa, r_i^j = \mathsf{PRF}_{\mathsf{rs}_i}(j) \; \forall\, 1 \leq j \leq \sigma$
  hand $\mathsf{rs}_i, r_i^1, \ldots, r_i^\sigma$ to $\mathcal{A}_g$
  $E_i = \mathsf{Po2}(\sum_{j=1}^{\sigma} r_i^j e_i^j)$
for $i = 2, 4, \ldots, T - 3$ **do**
  $(e_i^j)_{1 \leq j \leq \omega} \leftarrow \mathcal{A}_g$ s.t. $e_i^j \in \mathbb{Z}_{2^{k+s}} \; \forall\, 1 \leq j \leq \omega$,
    $\exists j' : \mathsf{Po2}(e_i^{j'}) \leq E_{i-1}$
  $\mathsf{rs}_i \in_R \{0,1\}^\kappa, r_i^j = \mathsf{PRF}_{\mathsf{rs}_i}(j) \; \forall\, 1 \leq j \leq \omega$
  hand $r_i^1, \ldots, r_i^\omega$ to $\mathcal{A}_g$
  $E_i = \mathsf{Po2}(\sum_{j=1}^{\omega} r_i^j e_i^j)$
  $(e_{i+1}^j)_{1 \leq j \leq \omega} \leftarrow \mathcal{A}_g$ s.t. $e_{i+1}^j \in \mathbb{Z}_{2^{k+s}} \; \forall\, 1 \leq j \leq \omega$,
    $\exists j' : \mathsf{Po2}(e_{i+1}^{j'}) \leq E_i$
  $r_{i+1}^j = \mathsf{PRF}_{\mathsf{rs}_{i+1}}(\omega + j) \; \forall\, 1 \leq j \leq \omega$
  hand $\mathsf{rs}_{i+1}, r_{i+1}^j$ to $\mathcal{A}_g$
  $E_{i+1} = \mathsf{Po2}(\sum_{j=1}^{\omega} r_{i+1}^j e_{i+1}^j)$
for $i = T - 1$ **do**
  $(e_i^j)_{1 \leq j \leq d} \leftarrow \mathcal{A}_g$ s.t. $e_i^j \in \mathbb{Z}_{2^{k+s}} \; \forall\, 1 \leq j \leq d$,
    $\exists j' : \mathsf{Po2}(e_i^{j'}) \leq E_{i-1}$
  $e_i^0 \leftarrow \mathcal{A}_g$ s.t. $e_i^0 \in \mathbb{Z}_{2^{k+s}}$
  $\mathsf{rs}_i \in_R \{0,1\}^\kappa, r_i^j = \mathsf{PRF}_{\mathsf{rs}_i}(j) \; \forall\, 1 \leq j \leq d$
  hand $r_i^1, \ldots, r_i^d$ to $\mathcal{A}_g$
  $E_i = \mathsf{Po2}(\sum_{j=1}^{d} r_i^j e_i^j + e_i^0)$
  $(e_{i+1}^j)_{1 \leq j \leq d} \leftarrow \mathcal{A}_g$ s.t. $e_{i+1}^j \in \mathbb{Z}_{2^{k+s}} \; \forall\, 1 \leq j \leq d$,
    $\exists j' : \mathsf{Po2}(e_{i+1}^{j'}) \leq E_i$
  $r_{i+1}^j = \mathsf{PRF}_{\mathsf{rs}_{i+1}}(d + j) \; \forall\, 1 \leq j \leq d$
  hand $\mathsf{rs}_{i+1}, r_{i+1}^j$ to $\mathcal{A}_g$
  $E_{i+1} = \mathsf{Po2}(\sum_{j=1}^{d} r_{i+1}^j e_{i+1}^j)$
return win if $E_T = k + s$ **else return fail**

Figure 11: Security games used for the soundness proof.

contrasts a choice of random numbers using a PRF, requiring a less efficient random sampling for each required random number for the reduction to work.

Hence, we introduce a modified $\mathcal{G}ame_0'(k, s, T, \sigma, \omega, d)$ (Fig. 11) that closer aligns with our specific proof from §4.2. As additional parameters, we have $\sigma, \omega, d \in \mathbb{N}$. Value d here represents the vector dimension in $\Pi_{\mathsf{checkTriple}}$ which results from repeatedly dividing $3m$ by ω and rounding up the result until $d \leq \omega$. The game first does derive its random challenges e_i^j from a PRF that we need for our more efficient instantiation. Accordingly, they are sampled from a random seed which is also handed to the adversary. In addition, we use the actual linear combinations (of errors) from the protocol, whereas $\mathcal{G}ame$ abstracts away all but one value into c_i. During degree reductions ($\Pi_{\mathsf{reduceDeg}}$), one PRF seed is used for two linear combinations, which is why pairs of iterations need to be grouped to use the same seed. Finally, the last pair of iterations (from $\Pi_{\mathsf{checkTriple}}$) additionally allows the adversary to add a value to the random linear combination. This technicality stems from the fact that as observed in §4.2.4, regarding soundness, the first of two random linear combinations of errors gets extended by an additional error without a random coefficient.

Lemma 1 ($\mathcal{G}AME_0'(k, s, T, \sigma, \omega, d) \leq \mathcal{G}AME(k, s, T)$). *If the winning probability of any $\mathcal{A}_g$ for $\mathcal{G}ame$ is negligible in σ, so is the winning probability of any $\mathcal{A}_g$ for $\mathcal{G}ame_0'$.*

Proof. We introduce intermediate games $\mathcal{G}ame_1'(k, s, T, \sigma, \omega, d)$ and $\mathcal{G}ame_2'(k, s, T, \sigma, \omega, d)$ to bridge the gap between $\mathcal{G}ame_0'$ and $\mathcal{G}ame$. $\mathcal{G}ame_1'$ only differs from $\mathcal{G}ame_0'$ in that it does not hand the random seeds rs_i to $\mathcal{A}_g$. $\mathcal{G}ame_2'$ differs from $\mathcal{G}ame_0'$ by instead of sampling random seeds rs_i to derive pseudorandom values from, it directly randomly samples these values without use of a PRF and handing any seed to $\mathcal{A}_g$. We proceed to first reduce $\mathcal{G}ame_0'$ to $\mathcal{G}ame_1'$, then reduce $\mathcal{G}ame_1'$ to $\mathcal{G}ame_2'$, and finally reduce $\mathcal{G}ame_2'$ to $\mathcal{G}ame$.

For the first step, assume an adversary $\mathcal{A}_g^{\mathcal{G}ame_0'}$ for $\mathcal{G}ame_0'$ with advantage p. Then, there is an adversary $\mathcal{A}_g^{\mathcal{G}ame_1'}$ for $\mathcal{G}ame_1'$ with at least the same advantage. This holds as when $\mathcal{A}_g^{\mathcal{G}ame_0'}$ learns a seed rs_i, the derived randomness is only used in conjunction with e_i^j-values that $\mathcal{A}_g^{\mathcal{G}ame_0'}$ has chosen before learning the seed. It could use additional knowledge of the seed in further steps to select, e.g., e_{i+1}^j-values. Yet, future steps do, besides the adversary's actions, only depend on E_i and future random values which are derived from seeds independent to rs_i. So there is no benefit for $\mathcal{A}_g^{\mathcal{G}ame_0'}$ to utilize knowledge of rs_i. Hence, there is an adversary of at least the same advantage not using rs_i, and this adversary then can be used as $\mathcal{A}_g^{\mathcal{G}ame_1'}$ with the same advantage.

For the second step, consider now an algorithm D that takes as input all random challenges r_i^j from $\mathcal{G}ame_1'/\mathcal{G}ame_2'$ and otherwise behaves like these games, outputting 1 iff $\mathcal{A}_g$ wins. We group the random challenges into partitions which, in $\mathcal{G}ame_1'$, use the same seed. If for each segment, we provide truly random challenges as input to D, then D behaves like $\mathcal{G}ame_2'$. If instead, for each partition we sample a random seed and derive the challenges from

that, providing them as input to D, then D behaves like $\mathcal{G}ame_1^{'}$. We now do game hops where in each hop, we change the inputs for one partition of pseudorandom to truly random challenges to get from $\mathcal{G}ame_1^{'}$ to $\mathcal{G}ame_2^{'}$. In each of these hops, D behaves the same (except for some negligible difference) for both sets of inputs. Otherwise, it would distinguish between the PRF outputs and truly random values, which would contradict the security of the used PRF which we assume to be secure. Hence, the winning probabilities for $\mathcal{G}ame_1^{'}, \mathcal{G}ame_2^{'}$ can only differ by a negligible amount.

Finally, to reduce $\mathcal{G}ame_2^{'}$ to $\mathcal{G}ame$, we need to group some parts of the random linear combinations. For an adversary $\mathcal{A}_g^{\mathcal{G}ame_2^{'}}$, we construct an adversary $\mathcal{A}_g^{\mathcal{G}ame}$ with equal advantage. For $i = 1$, we set e_i (from $\mathcal{G}ame$) to $e_i^{j'}$ (from $\mathcal{G}ame_2^{'}$) as both satisfy that their Po2 is at most E_{i-1}. Furthermore, we pick $r_i^j \in_R \mathbb{Z}_{2^{k+s}}$ for $1 \leq j \leq \sigma, j \neq j'$, and then define c_i (from $\mathcal{G}ame$) as $\sum_{1 \leq j \leq \sigma, j \neq j'} r_i^j e_i^j$. We interpret challenge r_i (from $\mathcal{G}ame$) as $r_i^{j'}$ (from $\mathcal{G}ame_2^{'}$) as both are sampled uniformly at random. E_1 is then defined as usual. Iterations $i = 2, 4, \ldots, T-3$ follow the exact same structure, noting that the only difference in the individual steps is the size of the random linear combinations. For the last iteration $i = T-1$, there is one difference: the selection of e_i^0 by the adversary, added to the first random linear combination. Yet, the definition of $\mathcal{G}ame$ makes this easy. We again set $e_i = e_i^{j'}$ but use $c_i = \sum_{1 \leq j \leq d, j \neq j'} r_i^j e_i^j + e_i^0$, including the final offset. It is easy to see that the resulting adversary for $\mathcal{G}ame$ wins with the same probability as the one for $\mathcal{G}ame_2^{'}$. □

Lemma 2 (Reducing the Protocol Adversary to the Game $\mathcal{G}_{AME_0^{'}}(k, s, T, \sigma, \omega, d)$). *If the winning probability of any $\mathcal{A}_g$ for $\mathcal{G}ame_0^{'}$ is negligible in σ, so is the probability that any protocol adversary that chooses for each multiplication $1 \leq i \leq m$ errors $\delta_i \in \mathbb{Z}_{2^k}$ s.t. $z_i = x_i y_i + \delta_i$ can make Π_{verify} accept if any of the errors was $\neq 0$.*

Proof. Assume that there exists an i' s.t. $\delta_{i'} \neq 0$ in $\mathbb{Z}_{2^k}$ and that Π_{verify} accepts. Note that $z_i = x_i y_i + \delta_i$ implies that $\mathsf{m}_{z_i} = \mathsf{m}_{x_i}\mathsf{m}_{y_i} + \mathsf{m}_{x_i}\lambda_{y_i} + \lambda_{x_i}\mathsf{m}_{y_i} + \gamma_{x_i y_i} - \lambda_{z_i} + \delta_i$. As Π_{verify} accepts, we can use that all instances of verify succeeded, i.e., $\langle\cdot\rangle$-shares were opened correctly and all sharings are consistent. First, we assume that each Γ^ℓ as computed in the protocol is correct, i.e., $\Gamma^\ell := \sum_{i=1}^m r_i^\ell(\mathsf{m}_{x_i}\lambda_{y_i} + \lambda_{x_i}\mathsf{m}_{y_i} + \gamma_{x_i y_i} - \lambda_{z_i})$. Then, each honest party receives

$$v^\ell = \Lambda^\ell + \Gamma^\ell = \sum_{i=1}^m r_i^\ell \left(\mathsf{m}_{x_i}\mathsf{m}_{y_i} - \mathsf{m}_{z_i}\right)$$

$$+ \sum_{i=1}^m r_i^\ell(\mathsf{m}_{x_i}\lambda_{y_i} + \lambda_{x_i}\mathsf{m}_{y_i} + \gamma_{x_i y_i} - \lambda_{z_i}) = \sum_{i=1}^m r_i^\ell \delta_i.$$

Note that all $r_i^\ell \in \{0, 1\}$ are obtained from Π_{rnd}, so they are unknown to the adversary on selection of the δ_i. In addition, they are derived from a random seed rs using a PRF, so we can also assume them to be truly random for this proof, given that any non-negligible difference in soundness would contradict security of the PRF.[10] Now, $v^\ell = 0 \implies r_{i'}^\ell \delta_{i'} = -\sum_{1 \leq i \leq m, i \neq i'} r_i^\ell \delta_i$. With randomly chosen $r_{i'}^\ell \in \{0, 1\}$, this only holds with probability 0.5. As the verification only accepts if $v^\ell = 0$ for all $1 \leq \ell \leq \sigma$, it only accepts with a probability $\leq 2^{-\sigma}$.

Now, we assume that at least one Γ^ℓ is incorrect, i.e., $\Gamma^\ell = \vec{\mathbf{a}'}^\ell \cdot \vec{\mathbf{b}'}^\ell + \delta'_\ell$ where there exists at least one ℓ' with $\delta'_{\ell'} \neq 0$ in $\mathbb{Z}_{2^k}$. Recall from §4.2.2 that the protocol computes triples $c^\ell = \vec{\mathbf{a}}^\ell \cdot \vec{\mathbf{b}}^\ell + \delta_\ell$, including potentially introduced errors δ_ℓ, over $\mathbb{Z}_{2^{k+s}}$. As $\Gamma^\ell, \vec{\mathbf{a}'}^\ell, \vec{\mathbf{b}'}^\ell$ are their counterparts truncated down to k bits, we have that $\delta_{\ell'} \equiv_{2^k} \delta'_{\ell'} \not\equiv_{2^k} 0$. Then, $\mathsf{Po2}(\delta_{\ell'}) \leq k - 1 = E_0$. We let $\mathcal{A}_g$ pick $e_1^j = \delta_j$ for $1 \leq j \leq \sigma$, one of them having Po2 at most E_0 as required. Now, we interpret challenge $r_1^1, \ldots, r_1^\sigma$ from the game as the random coefficients from step 9 of Π_{verify}, noting that in the game and the protocol, the adversary also learns the fitting, random seed. While the seed is sampled by $\mathcal{D}$ beforehand in our protocol, the hiding property of $\Pi_{\mathsf{rec}}^{\mathsf{active}}$ ensures that the seed is unknown to all parties in advance. Then, $E_1 = \mathsf{Po2}(\sum_{j=1}^\sigma r_i^j e_i^j) = \mathsf{Po2}(\delta')$ where δ' is the error of the new triple $z = \vec{\mathbf{x}} \cdot \vec{\mathbf{y}} + \delta'$, using the soundness considerations from §4.2.2.

Each instance of $\Pi_{\mathsf{reduceDeg}}$ in the protocol corresponds to one of the iterations $i = 2, 4, \ldots, T-3$ in $\mathcal{G}ame_0^{'}$. As per the considerations from §4.2.3, an input error $z = \vec{\mathbf{x}} \cdot \vec{\mathbf{y}} + \delta$ leads to an output error $z' = \vec{\mathbf{x}'} \cdot \vec{\mathbf{y}'} + \delta'$ which is the random linear combination of random linear combinations. Using as invariant that $E_{i-1} = \mathsf{Po2}(\delta)$, as per §4.2.3 there exists a ℓ' s.t. $\mathsf{Po2}(\delta^{\ell',\ell'}) \leq E_{i-1}$ with $\delta^{\ell',\ell'}$ as in §4.2.3. We set in the game $e_i^j = \delta^{j,\ell'}$ from the protocol for $1 \leq j \leq \omega$. Then, we interpret the new challenge from the game as half of the pseudorandom coefficients from step 5 of $\Pi_{\mathsf{reduceDeg}}$ as before, but without revealing the seed. Then, $E_i = \mathsf{Po2}(\sum_{j=1}^\omega r_i^j e_i^j)$. We do a second step where in the game, we set $e_{i+1}^j = \sum_{\ell=1}^\omega \beta^\ell \delta^{j,\ell}$ for $1 \leq j \leq \omega$ with the sums as in §4.2.3. Note that one of the values equals $\sum_{j=1}^\omega r_i^j e_i^j$, hence, the requirement regarding one value with Po2 of at most E_i is satisfied. We interpret the new challenge from the game as the second half of the pseudorandom coefficients from step 5 of $\Pi_{\mathsf{reduceDeg}}$, this time also revealing the seed. This yields E_{i+1} as defined by the game.

The final iteration $i = T - 1$ works like each prior dimension reduction. In particular, we again have an error $\delta^{\ell',\ell'}$ with $\mathsf{Po2}(\delta^{\ell',\ell'}) \leq E_{i-1}$. The only difference is that in the protocol, as per §4.2.4, the inner random linear combinations on the errors are extended by adding $\delta^{j,0}$ without a random coefficient. Similarly, in the game the final iteration enables $\mathcal{A}_g$ to choose such a value, so we simply set $e_i^0 = \delta^{\ell',0}$. Otherwise following all steps such as for $\Pi_{\mathsf{reduceDeg}}$ finally yields E_T.

Finally, note that if the protocol adversary wins, then it must hold that $z' \equiv_{2^{k+s}} x'y'$ in $\Pi_{\mathsf{checkTriple}}$. Hence, for error δ' with $z' = x'y' + \delta'$, $\mathsf{Po2}(\delta') = k + s$. Note that E_T exactly is computed as $\mathsf{Po2}(\delta')$ by our reduction. Hence, the protocol adversary wins iff $E_T = k + s$, i.e., $\mathcal{A}_g$ wins $\mathcal{G}ame_0^{'}$. □

Lemma 3. *Let $k, T \in \mathbb{N}$ with $T \leq \sigma$. For $s \geq \max(3T, \sigma + T(\frac{1}{2} + \log_2(\frac{5}{2} + \frac{3\sigma}{T})))$, any $\mathcal{A}_g$ wins $\mathcal{G}ame(k, s, T)$ with probability of at most $2^{-\sigma}$.*

[10] Note that while the adversary also learns rs, it does so in conjunction with the other random values while we only require that it has no knowledge of it when selecting the errors δ_i.

Proof. This simply is the combination of [46, Lemma 5.1] and [45, Appendix G.6.1 (full version of [46])] proven in [45]. □

E.2 Security

We proceed to prove the security of our protocol Π_{MPC}, which, for the sake of completeness, we also formally define here in Fig. 12.

Protocol Π_{MPC}

Inputs: $x_i^1, \ldots, x_i^{k_i} \in \mathbb{Z}_{2^k}$ from $\mathcal{P}_i$ with k_i inputs, for $1 \leq i \leq n$,

Output: $C(x_1^1, \ldots, x_1^{k_1}, \ldots, x_n^1, \ldots, x_n^{k_n})$.

1. $\mathcal{D}$ samples a random pre-shared seed for each party $\mathcal{P}_i$ and hands it to this party.
2. Obtain $\left[x_i^j\right]$ for all $1 \leq i \leq n, 1 \leq j \leq k_i$ using Π_{input}.
3. Evaluate the circuit, using the linearity of $[\cdot]$ for all linear operations and Π_{mult} for all multiplications.
4. Run Π_{verify} on the data from all multiplications and abort if it returns Reject.
5. Otherwise, reconstruct all circuit outputs $[y]$ by reconstructing $\langle \lambda_y \rangle$ using $\Pi_{\mathsf{rec}}^{\mathsf{active,rerand}}$ and then computing $y = \mathsf{m}_y + \lambda_y$.

Figure 12: Π_{MPC}: MPC protocol for evaluating an arithmetic circuit C.

Proof of Theorem 1 (§4.2.6). For a corrupted subset of parties, we provide a simulator in Fig. 13.

Simulator $\mathcal{SIM}_{\mathsf{MPC}}$

Using $\mathcal{P}_h$ as an arbitrary, but fixed (given the set of corrupted parties) party.

1. Sample random pre-shared seeds and simulate them being handed to corrupted parties by $\mathcal{D}$.
2. Simulate all instances of $\Pi_{\mathsf{rec}}^{\mathsf{active}}$.commit and the preprocessing of all instances of Π_{mult} with the corresponding sub-simulators.
3. For each input by an honest party, follow the protocol specification using 0 as the input value, using the subsimulator for Π_{input}.
4. For each input by a corrupt party, again use the subsimulator for Π_{input}. Note that this enables extraction of the corrupt party's input.
5. For linear gates, follow the protocol specification.
6. For each multiplication, run the subsimulator for the online phase of Π_{mult}. Use m_v-values of $\mathcal{P}_h$ and λ_v-values as known to $\mathcal{D}$ to derive any potential error introduced by the adversary.
7. Follow steps 1–11 of Π_{verify}, using the corresponding subsimulators for $\Pi_{\mathsf{rnd}}, \Pi_{\mathsf{shareSum}}$.
8. For step 12 of Π_{verify}, simulate all iterations of $\Pi_{\mathsf{reduceDeg}}$ using the corresponding subsimulators for $\Pi_{\mathsf{rnd}}, \Pi_{\mathsf{shareSum}}$.
9. For step 13 of Π_{verify}, simulate $\Pi_{\mathsf{checkTriple}}$ as follows:
   a. Simulate steps 1–4, using the subsimulator for Π_{shareSum}. Use m_v-values of $\mathcal{P}_h$ and λ_v-values as known to $\mathcal{D}$ to derive $z^{j,\ell}$ for all $1 \leq j \leq d, 0 \leq \ell \leq d$. Also, use that $\mathcal{P}_h$ knows $\vec{\mathbf{x}}$ and that $\mathcal{D}$ knows all $\vec{\mathbf{y}}_i$ to compute the errors $\delta^{j,\ell}$ on all $z^{j,\ell}$ as defined in the soundness part of Fig. 4.2.4.
   b. For step 5, simulate Π_{rnd}, $\Pi_{\mathsf{rec}}^{\mathsf{consistent}}$.verify, $\Pi_{\mathsf{rec}}^{\mathsf{active}}$.verify, as subprotocol verification steps are delayed until this step. The only interaction of Π_{rnd} is a call to $\Pi_{\mathsf{rec}}^{\mathsf{active}}$. We run the subsimulators for $\Pi_{\mathsf{rec}}^{\mathsf{active}}$.open, $\Pi_{\mathsf{rec}}^{\mathsf{active}}$.verify, $\Pi_{\mathsf{rec}}^{\mathsf{consistent}}$.verify. Inspection of the subsimulators confirms that this is possible concurrently, given that the required communication is done on independent values, at least from the honest parties. Synchronization of aborts is simulated by following the protocol specification where each party sends a bit to each other party, indicating if it did abort. Then, simulate each honest party that did send or receive an abort aborting the protocol execution. Send an early abort to the ideal functionality. Otherwise, compute δ' with $z' = x' \cdot y' + \delta'$ as described in the soundness in §4.2.4, using the errors from step 9.a and the random values revealed by Π_{rnd}.
   c. Follow the protocol specification for step 6 of $\Pi_{\mathsf{checkTriple}}$.
   d. For step 7, simulate $\Pi_{\mathsf{rec}}^{\mathsf{active,rerand}}$, including its verify, using the corresponding subsimulator. For $\langle y' \rangle$, hand input y' to the subsimulator which is known to the simulator as it is known to $\mathcal{D}$. For $\langle z' \rangle$, hand input $\lambda_{z''} = x' \cdot y' + \delta' - \mathsf{m}_{z'}$ to the subsimulator, using δ' computed in step 9.b. Synchronization of aborts is done as before, now including the check if the triple is correct.
10. Simulate step 14 of Π_{verify} with the corresponding subsimulator for $\Pi_{\mathsf{rec}}^{\mathsf{active,rerand}}$, including its verify. Recall from the proof of Lemma 2 how each v^ℓ is a linear combination of random bits (which the simulator learns in step 7) and errors extracted in step 6. From these and m_{v^ℓ}, compute fitting λ_{v^ℓ} and hand these to the subsimulator for $\Pi_{\mathsf{rec}}^{\mathsf{active,rerand}}$. Synchronization of aborts is done as before, now including the check if all v^ℓ are zero.
11. Send the inputs extracted in step 4 to the ideal functionality, computing circuit C, and receive the adversary's outputs.
12. For each output y' to the adversary, reconstruct the actual represented value y and then run the subsimulator for $\Pi_{\mathsf{rec}}^{\mathsf{active,rerand}}$ on input $y' - \mathsf{m}_y$ with m_y from $\mathcal{P}_h$. Include the verify steps. Follow the protocol specification for all other parties' outputs. If an abort happens, send a late abort to the ideal functionality. Finally, output what the adversary outputs.

Figure 13: $\mathcal{SIM}_{\mathsf{MPC}}$: Simulator for Π_{MPC}

Indistinguishability between real and ideal world is trivial up to step 9.d, following the security considerations for the used subprotocols and the fact that all instances of $\Pi_{\mathsf{rec}}^{\mathsf{consistent}}, \Pi_{\mathsf{rec}}^{\mathsf{active}}$ are on independent random sharings. Then, in step 9.d, indistinguishability follows the security of $\Pi_{\mathsf{rec}}^{\mathsf{active,rerand}}$, from the fact y' is masked by random y_0, and by the considerations regarding the error in z' from §4.2.4. For step 10, also note that the revealed v^k are indistinguishable as they are determined by the random coefficients and introduced errors, as discussed in Lemma 2. Steps 11 and 12 exactly reconstruct the expected values which, together with the security of $\Pi_{\mathsf{rec}}^{\mathsf{active,rerand}}$, again yields indistinguishability.

As an additional remark, note that for both instantiations of Π_{rec} (§3.1), while a cheater might cause honest parties to receive incorrect or inconsistent outputs, the following holds: the adversary always knows the outputs received by all honest parties. For $\Pi_{\mathsf{rec}}^{\mathsf{king}}$ and an honest $\mathcal{P}_{\mathsf{king}}$, $\mathcal{P}_{\mathsf{king}}$ distributes a consistent value to all parties. If a corrupted party provides an incorrect share, this offsets the resulting value for all parties by some value trivially known to the adversary. If $\mathcal{P}_{\mathsf{king}}$ is corrupted, it hands the result to all parties. The adversary chooses and, hence, also knows all parties' outputs. For $\Pi_{\mathsf{rec}}^{\mathsf{bc}}$, the adversary learns the shares of all honest parties. Together with the shares it sends to each of the honest parties, it then has all shares handed and held by each honest party and can derive its output. Hence, the verify-steps does not enable selective failure attacks, the adversary is already aware of its output in advance.

The indistinguishability between the real and ideal world as justified above relies on the protocol aborting if any output wire of the circuit holds an incorrect value, i.e., on the soundness of our verification. Using Lemma 3, the choice of s guarantees that the winning probability of any $\mathcal{A}_g$ in $\mathcal{G}ame(k, s, T)$ is negligible. Then, so is the winning probability regarding $\mathcal{G}ame_0^{'}(k, s, T, \sigma, \omega, d)$, following Lemma 1. Finally, Lemma 2 yields that the probability of any protocol adversary breaking soundness is negligible. □

F Support for Dot Products

For certain applications, e.g., in privacy-preserving machine learning, an efficient evaluation of dot products is crucial. As demonstrated in [42] or, specifically for rings in [46], zk-FLIOPs can also

be used for efficient verification of dot product evaluation. Here, we shortly line out how similarly, our verification from §4.2 can be extended to support dot products.

Passively secure multiplication using function-dependent preprocessing can easily be extended to dot products at no communication overhead, replacing all input scalars by vectors [20, 38]. For our multiplication protocol Π_{mult} (Fig. 3), we simply need to replace $\langle \mathsf{m}_z \rangle' = \mathsf{m}_x \mathsf{m}_y + \mathsf{m}_x \langle \lambda_y \rangle + \mathsf{m}_y \langle \lambda_x \rangle + \langle \gamma_{xy} \rangle - \langle \lambda_z \rangle$ by $\langle \mathsf{m}_z \rangle' = \vec{\mathsf{m}}_{\mathbf{x}} \vec{\mathsf{m}}_{\mathbf{y}} + \vec{\mathsf{m}}_{\mathbf{x}} \langle \vec{\lambda}_{\mathbf{y}} \rangle + \vec{\mathsf{m}}_{\mathbf{y}} \langle \vec{\lambda}_{\mathbf{x}} \rangle + \langle \gamma_{xy} \rangle - \langle \lambda_z \rangle$, using $\gamma_{xy} = \vec{\lambda}_{\mathbf{x}} \vec{\lambda}_{\mathbf{y}}$. Hence, our verification's goal now is to prove $\vec{\mathsf{m}}_{\mathbf{x}} \vec{\mathsf{m}}_{\mathbf{y}} + \vec{\mathsf{m}}_{\mathbf{x}} \vec{\lambda}_{\mathbf{y}} + \vec{\lambda}_{\mathbf{x}} \vec{\mathsf{m}}_{\mathbf{y}} + \gamma_{xy} - \lambda_z \stackrel{?}{=} \mathsf{m}_z$. Terms Λ, Γ in our proof are changed accordingly: $\Lambda := \sum_{i=1}^{m} r_i \left(\vec{\mathsf{m}}_{\mathbf{x}_i} \vec{\mathsf{m}}_{\mathbf{y}_i} - \mathsf{m}_{z_i} \right)$ and $\Gamma := \sum_{i=1}^{m} r_i (\vec{\mathsf{m}}_{\mathbf{x}_i} \vec{\lambda}_{\mathbf{y}_i} + \vec{\lambda}_{\mathbf{x}_i} \vec{\mathsf{m}}_{\mathbf{y}_i} + \gamma_{x_i y_i} - \lambda_{z_i})$. Then, we redefine

$$\Gamma = \begin{pmatrix} r_1 \vec{\mathsf{m}}_{\mathbf{x}_1} \\ r_1 \vec{\mathsf{m}}_{\mathbf{y}_1} \\ r_1 \\ \vdots \\ r_m \vec{\mathsf{m}}_{\mathbf{x}_m} \\ r_m \vec{\mathsf{m}}_{\mathbf{y}_m} \\ r_m \end{pmatrix} \cdot \begin{pmatrix} \vec{\lambda}_{\mathbf{y}_1} \\ \vec{\lambda}_{\mathbf{x}_1} \\ \gamma_{x_1 y_1} - \lambda_{z_1} \\ \vdots \\ \vec{\lambda}_{\mathbf{y}_m} \\ \vec{\lambda}_{\mathbf{x}_m} \\ \gamma_{x_m y_m} - \lambda_{z_m} \end{pmatrix}.$$

We continue to carry out verification as described in §4.2. Note that the new Γ does allow different dot products to have different dimensions. Furthermore, as a scalar multiplication can be interpreted as dot product with dimension 1, it is possible to mix verification of scalar multiplication and dot products. What changes is the dimension of inputs for computing Γ, now being $2\times$ the sum over all dot products' dimensions plus the number of dot products, whereas it is $3m$ when just computing multiplications on scalar values. This increases the verification complexity, but it remains sublinear in the number of multiplications/dot products.

We implement this extension and provide benchmark results in Tab. 6. Note that our verification's communication overhead is lower when we have more and smaller dot products, given our amortization properties and the fact that larger dot products increase the size of the statement to be proven. E.g., the verification for 10K dot products of dimension 100 yields almost the same verification complexity as for 1M scalar multiplications. The overhead regarding run time is significantly larger compared to scalar products, given the computational complexity of the now larger proof.

G Additional Empirical Comparisons

In this section, we provide additional comparison between our protocol and the protocols from [46] and [22]. We use the implementation of [46] that is based on MP-SPDZ [40]. After discovering that it partly ignores the ring sizes set through the appropriate interfaces of MP-SPDZ, we adapted the implementation to the best of our knowledge to support computation for $k = 32$ to match the setting that we evaluated our protocol in. Regarding [22], we use its implementation in MP-SPDZ [40]. MP-SPDZ does not support running the protocol with a dealer, so we only benchmark the online phase, noting that the offline phase without a dealer would be substantially more expensive. For the offline phase, the dealer would need to provide one authenticated triple per multiplication,

Table 6: Performance of our protocol on dot products of different dimensions. Reported numbers include offline and online. Communication is over all parties and dealer. Annotated in brackets is the verification overhead over the passively secure baseline protocol. Benchmark for circuit depth 1, $n = 3$ parties, and 8 threads per party.

#dot prod.	dim.	total comm. (MiB)	online rounds	run time (s)	
				LAN	WAN
10K	10	0.29 (+54.0%)	27	0.04 (× 6.0)	1.57 (×6.1)
10K	100	0.30 (+59.6%)	27	0.12 (×12.9)	1.67 (×6.3)
10K	1000	0.32 (+68.4%)	30	0.96 (×33.0)	2.62 (×9.2)
100K	10	2.02 (+ 6.1%)	27	0.18 (× 5.1)	2.27 (×3.3)
100K	100	2.04 (+ 6.8%)	30	1.06 (×16.7)	3.20 (×4.2)
1M	10	19.20 (+ 0.7%)	30	1.58 (× 6.1)	4.96 (×2.0)

which can be achieved by sending 4 elements of $\mathbb{Z}_{2^{k+\sigma}}$ and sampling other shares using pre-shared keys. For [22], we consider $\sigma = 32$ which enables to represent elements of $\mathbb{Z}_{2^{k+\sigma}}$ by standard 64-bit integers for better efficiency. Hence, we here use a lower security parameter for more efficient data representation, which only biases the benchmarks in favor of [22].

Comparison to Honest-Majority Three-Party Computation. For $n = 2$ parties and a dealer, our protocol has a similar setting to honest-majority three-party computation. Both feature three participants to the MPC who do not collude. Yet, our protocol has the advantage that one participant only is a dealer (which we here assume to be semi-honest in contrast to the other malicious parties), i.e., it only needs to provide preprocessing material at some point before inputs to the computation are provided, possibly for many protocol executions, and then go offline. This again results in the opportunity to distribute the dealer (cf. §D) while keeping the online phase efficient. Honest majority three-party computation has three full-fledged parties available in the online phase, which is a stronger requirement but provides more possibilities for optimization. In Tab. 7, we compare our protocol to the three-party protocol from [46] using zk-FLIOPs on $\mathbb{Z}_{2^k}$, which our protocol design is partly based on. Note that [46] uses no offline phase. Our protocol has lower communication, which we partly attribute to our protocol optimizations and the fact that we assume the dealer not to deviate from the protocol. Yet, as expected, it has higher run time. We note that the majority of run time overhead stems from requiring an offline phase, leading to the dealer acting in advance instead of all parties being equally active throughout the computation, resulting in lower parallelization compared to [46]. Our online phase is just 24.4% slower in LAN and 3.8% slower in WAN. To conclude, [46] has lower run time in this specific setting, whereas our protocol provides the freedom of just having two parties available for the online phase, and we also have good scalability for many parties while the $(2 + 1)$-party case is just one special case of our protocol.

Comparison to SPD$\mathbb{Z}_{2^k}$. We also compare to SPD$\mathbb{Z}_{2^k}$ [22] with a dealer, sharing our $(n + 1)$-party setting while relying on MACs instead of distributed zero-knowledge. For $n = 3$ parties, our communication is better by a factor of almost $5\times$. While our verification has

Table 7: Empirical comparison between our protocol, the zk-FLIOP protocol from [46] on honest-majority three-party computation, and SPD$\mathbb{Z}_{2^k}$ [22] with a dealer. Benchmarks for $m = 10^6$ multiplications across 30 layers. Communication and run times provided as offline + online (per party). Decreases and increases of communication w.r.t. the total communication across all parties and dealer.

	setting	total comm. (MiB)	online rounds	run time (s)	
				LAN	WAN
RSS zk-FLIOP on $\mathbb{Z}_{2^k}$ [46]	3 non-colluding parties	5.12	46	0.41	3.45
our protocol	2 parties + 1 non-colluding dealer	3.82 + 3.85	48	0.27 + 0.51	1.13 + 3.58
decrease/increase		−25.0%	+4.3%	+90.8%	+36.3%
SPD$\mathbb{Z}_{2^k}$ [22]+dealer	3 parties + 1 non-colluding dealer*	30.52* + 21.33	69	?* + 0.88	?* + 10.67
our protocol	3 parties + 1 non-colluding dealer	3.82 + 5.13	85	0.26 + 0.60	1.07 + 5.58
decrease/increase		−79.7%*	+23.2%	at least −1.8%*	at least −37.7%*

*: **The SPD$\mathbb{Z}_{2^k}$-implementation of MP-SPDZ [40] has no support for a dealer. Hence, we only run the online phase, manually calculate the expected offline communication, and can provide no offline run times.**

a higher round complexity, our online run time is lower than that of [22] by 32% for LAN and 48% for WAN. This indicates that the savings in communication from not requiring authenticated triples and not requiring the complete calculation to be over a larger ring $\mathbb{Z}_{2^{k+\sigma}}$ surpass potential computational overheads from our proof. Yet, it also is to be noted that [22] does not utilize function-dependent preprocessing to optimize performance as in our protocol, but we are not aware of any publicly available implementation of a variant for function-dependent preprocessing.