Sublinear Distributed Product Checks on Replicated Secret-Shared Data over $\mathbb{Z}_{2^k}$ Without Ring Extensions

Yun Li
Ant Group
Beijing, China
Tsinghua University
Beijing, China
liyun24@antgroup.com

Daniel Escudero
J.P. Morgan AI Research & J.P.
Morgan AlgoCRYPT CoE
New York, USA
daniel.escudero@protonmail.com

Yufei Duan
Tsinghua University
Beijing, China
dyf23@mails.tsinghua.edu.cn

Zhicong Huang
Ant Group
Hangzhou, China
zhicong.hzc@antgroup.com

Cheng Hong
Ant Group
Beijing, China
vince.hc@antgroup.com

Chao Zhang
Tsinghua University
Beijing, China
chaoz@tsinghua.edu.cn

Yifan Song*
Tsinghua University
Beijing, China
Shanghai Qi Zhi Institute
Shanghai, China
yfsong@mail.tsinghua.edu.cn

Abstract

Multiple works have designed or used maliciously secure honest majority MPC protocols over $\mathbb{Z}_{2^k}$ using replicated secret sharing (*e.g.* Koti *et al.* USENIX'21). A recent trend in the design of such MPC protocols is to first execute a semi-honest protocol, and then use a check that verifies the correctness of the computation requiring only *sublinear* amount of communication in terms of the circuit size. The so-called *Galois ring extensions* are needed in order to execute such checks over $\mathbb{Z}_{2^k}$, but these rings incur incredibly high computation overheads, which completely undermine any potential benefits the ring $\mathbb{Z}_{2^k}$ had to begin with.

In this work we revisit the task of designing sublinear distributed product checks on replicated secret-shared data over $\mathbb{Z}_{2^k}$ among three parties with an honest majority. We present a novel technique for verifying the correctness of a set of multiplication (in fact, inner product) triples, involving a sublinear cost in terms of the number of multiplications. Most importantly, unlike previous works, our tools *do not rely on Galois ring extensions, which are computationally expensive*, and only require computation over rings of the form $\mathbb{Z}_{2^\ell}$. In terms of communication, our checks are $3 \sim 5\times$ lighter than existing checks using ring extensions, which is already quite remarkable. However, our most noticeable improvement is in terms of computation: our checks are $17.7 \sim 44.2\times$ better than previous approaches, for many parameter regimes of interest. Our experimental results show that checking a 10 million gate circuit with the 3PC protocol from Boyle *et al.* (CCS'19) takes about two minutes, while our approach takes only 2.82 seconds.

Finally, our techniques are not restricted to the three-party case, and we generalize them to replicated secret-sharing with an arbitrary number of parties n. Even though the share size in this scheme grows exponentially with n, prior works have used it for $n = 4$ or $n = 5$ — or even general n for feasibility results — and our distributed checks also represent improvements in these contexts.

CCS Concepts

• **Security and privacy → Cryptography**.

Keywords

Secure Computation, Malicious Security, Distributed Zero-knowledge Proofs

1 Introduction

With the recent emergence of large language models, machine learning has again demonstrated its remarkable ability to extract intricate patterns from massive training data, revolutionizing various fields from natural language processing to image recognition and beyond. Consequently, a huge amount of data (which may contain sensitive information) is collected and used for training these models, leading to growing concerns about privacy leakage issues. Privacy-Preserving Machine Learning (PPML) has been proposed in response to these concerns. By integrating privacy-enhanced techniques, PPML frameworks ensure that sensitive data remains protected during training and inference processes. Among all these techniques, Secure Multi-Party Computation (MPC) has shown to be a vital and promising cryptographic tool, standing out by its

[*] Corresponding author

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690260

relatively reduced overhead, strong privacy guarantees, as well as trustlessness of hardware.

Originating from Yao's millionaire problem [46], MPC has evolved as a typical and blooming cryptographic primitive for enhancing privacy, upon which many excellent PPML frameworks [15, 22, 38, 40, 42, 45] are built. Informally, it enables a set of n mutually distrustful parties to securely compute a given function on their individual private inputs, while revealing only the final output. In the passive adversarial model, an adversary who corrupts t out of the n involved parties obeys the protocol specification but tries to learn the honest parties' inputs. For honest majority (where $t < n/2$), a common paradigm for designing MPC protocols is using secret sharing, with Shamir secret sharing being a prime example. However, for a small number of parties, Replicated Secret Sharing [34] (RSS) has proven to be a much more suitable choice—the state-of-the-art RSS-based three-party computation protocol [6] only incurs a communication cost of one element per multiplication gate per party. Many popular PPML frameworks [15, 22, 38, 40, 42] have adopted such protocols as basic building blocks.

In the active security model, the adversary can arbitrarily deviate from the protocol specification, and thus may cause more serious harm. Ideally, a protocol would achieve *guaranteed output delivery* (GOD), meaning that these adversarial deviations cannot prevent honest parties from learning the output. However, occasionally the relaxed notion of *security with abort* is useful (as it is simpler and typically more efficient), dictating that active adversaries may learn the output while preventing the honest parties to do so, but are not able to learn any information besides the output about these parties' inputs. Generally, constructing actively secure computation protocols is more difficult than constructing passively secure ones, since extra checks are required to detect potentially malicious behavior. A common approach for active security over *fields* is using IT-MACs [17], which typically increases communication of the underlying passive protocol by at least 2×. Distributed product checks [11, 12, 32] on the other hand have been recently developed as an appealing alternative that, asymptotically, incurs no extra cost with respect to passive security.

Unfortunately, these techniques do not work well when computations are conducted over rings like $\mathbb{Z}_{2^k}$, which arises from the fact that these structures have undesirable properties such as existence of non-zero zero divisors or lack of invertibility, and thus further disable polynomial interpolation, the Schwartz-Zippel lemma[1], and other essential tools. In fact, even Shamir secret sharing with passive security over $\mathbb{Z}_{2^k}$ turns out to be challenging [3], as it highly relies on polynomial interpolation operations. Other secret sharing schemes, such as RSS, are more suitable for $\mathbb{Z}_{2^k}$, and this is the scheme we focus on in our work.

We note that the share size of RSS grows exponentially with the number of parties n, and hence our work is mostly suitable for small n, and in particular we target the relevant case of 3-party computation with 1 actively corrupted party (which is the minimum number of parties for honest majority). We remark that, for small n and for an honest majority, RSS is the preferred scheme for constructing MPC protocols: (1) RSS-based MPC protocols offer the optimal communication cost in this case (e.g., one element per multiplication gate per party for $n = 3$, not achievable with Shamir secret sharing), (2) RSS-based MPC protocols are lightweight computationally as they only involve simple addition and multiplication, without polynomial interpolations required by Shamir secret sharing; and (3) the fact that in RSS the same share is held by multiple parties enables simpler and more efficient protocols with guaranteed output delivery [11, 12]. Given the above, RSS-based schemes for small n have been well explored in previous works [14, 15, 33, 37, 38, 40, 42]. Other than the most common scenario of $n = 3$, settings like $n = 4$ or $n = 5$ also appear in the literature [15, 33, 37, 39] and are proved to be concretely efficient as well. Despite having a particular focus on the 3-party case, our techniques naturally extend to the general case of n parties and can potentially be used when n is larger (in fact, our presentation is for the n-party case, while we focus our experiments on $n = 3$).

Given that achieving active security over $\mathbb{Z}_{2^k}$ is not an easy task, it has become the topic of study of multiple works [3, 4, 11, 12]. The core techniques used in these works can be roughly divided into two. One is the "SPDZ2k trick" [4, 13, 19, 41], which adapts MACs to work over $\mathbb{Z}_{2^k}$ by making use of a larger ring $\mathbb{Z}_{2^{k+s}}$, where s is roughly the security parameter. While being lightweight in terms of computation, the SPDZ2k trick incurs ≥ 2× overheads in communication. The second approach is to use a Galois ring extension of $\mathbb{Z}_{2^k}$, which is a ring of the form $\mathbb{Z}_{2^k}[\mathtt{X}]/(f(\mathtt{X}))$, where $f(\mathtt{X})$ is a degree-d monic polynomial over $\mathbb{Z}_{2^k}$ that is irreducible when taken modulo 2. As observed initially in [3], Galois ring extensions serve as a fundamental tool for $\mathbb{Z}_{2^k}$-based MPC, because of the following reasons: (1) $\mathbb{Z}_{2^k}$ can be embedded in these rings, and (2) they behave in *almost* the same way as the *field* $\mathsf{GF}(2^d)$, which enables translating most field-based techniques to the ring case [10, 11, 26, 37, 38, 42]. Given these properties, Galois ring extensions have been used to adapt sublinear distributed product checks to $\mathbb{Z}_{2^k}$ [10, 11], with the intent of reducing the passive-to-active overhead to ≈ 1×, *asymptotically*. In fact, it is only through Galois rings that such checks over $\mathbb{Z}_{2^k}$ are known.

Unfortunately, Galois rings, albeit convenient and elegant, add several *concrete* overheads in terms of communication and computation. A Galois ring element is a polynomial of degree $< d$ with coefficients in $\mathbb{Z}_{2^k}$, and hence it is d times larger than a single element of $\mathbb{Z}_{2^k}$. As a result, representing an element of $\mathbb{Z}_{2^k}$ as an element in the Galois ring brings an overhead of d, which depending on the application at hand can be as large as the statistical security parameter.[2] Even more critically, the main practical drawback of Galois rings is that *computation* over these rings, even for a moderately large degree, is fairly expensive. Libraries such as ZEN[3] or NTL[4], which to the best of our knowledge can be considered the state-of-the-art for Galois ring computations, are experimentally shown in [22, Section 5] — and also in our work — to be insufficient for practical MPC.

[1] Here we mean the same Schwartz-Zippel lemma required in the field case does not work in the ring case. There is a variant of the Schwartz-Zippel lemma used in the previous work [43] that does hold over $\mathbb{Z}_{2^k}$, but it is not suitable for the interpolation-based checks traditionally required for sublinear verification.

[2] Works like [26] have made use of reverse multiplication-friendly embeddings (RMFEs) to reduce the overhead of this large size from d to a constant > 2. However, these techniques are mostly of theoretical interest at the moment.

[3] https://zenfact.sourceforge.net/

[4] https://libntl.org/

This leads to a very unfortunate situation: Sublinear distributed product checks over $\mathbb{Z}_{2^k}$ using Galois rings allow a transition from passive to active security essentially for free in communication (asymptotically in $|C|$). The computation overhead and its concrete communication, however, are too large to be practical. On the other hand, the SPDZ2k-based approach, while being computationally efficient, brings at least 2× overhead in communication. This sets the stage for the question:

> *Can we obtain concretely efficient sublinear distributed product checks over $\mathbb{Z}_{2^k}$? That is, can we obtain actively secure honest majority protocols over $\mathbb{Z}_{2^k}$ that (1) add no communication overhead (asymptotically) with respect to passive security and (2) achieve comparable concrete efficiency with protocols based on SPDZ2k trick?*

1.1 Our Contribution

In this work, we address the aforementioned question by making the following contributions:

Contribution 1. We design a novel sublinear distributed product check protocol which entirely avoids ring extensions and works natively over a ring of the form $\mathbb{Z}_{2^\ell}$. Our techniques are specifically tailored to the RSS scheme in the honest-majority setting, which is the preferred method for a small number of parties [11, 12, 33, 37, 38, 42]. It is *much cheaper* in computation than the ring-extension-based verification protocols used in [11, 12, 37, 38, 42], and meanwhile only incurs *sublinear* communication cost, and thus have the same amortized asympotical communication complexity as the underlying passively secure protocol.

Contribution 2. We fully implement our protocol in the MP-SPDZ framework [36] and benchmark its efficiency in the three-party case, upon the state-of-the-art RSS-based passively secure protocol [6]. We also implement in the same framework the ring extension-based sublinear distributed product check protocol from [11]. Experiment results show that in the LAN (and WAN) setting, our protocol achieves up to 44.2× (and 9.7× resp.) speedup than [11], with 1.22× more lightweight communication cost.

Our checks can be used for concretely efficient passive-to-active MPC compilation over $\mathbb{Z}_{2^k}$. In Appendix C, we describe an MPC protocol that uses our checks for security with abort, and extend it to GOD in Appendix F. As we mentioned, for $n = 3$ our protocol drastically improves both in communication and computation over [11], and makes sublinear checks over $\mathbb{Z}_{2^k}$ concretely practical. We report extensive experimental results in this setting, and we have made our MP-SPDZ source code available,[5] including the implementation of our protocol and that of [11], which has not been implemented to the best of our knowledge. Finally, distributed checks on RSS-shared multiplication triples is a core primitive that has been used (with expensive Galois ring extensions) in several actively secure protocols over $\mathbb{Z}_{2^k}$ [11, 12, 33, 37, 38, 42], for $n = 3$ and other small values of n too. Our efficient verification can be used as a drop-in replacement to improve the performance of these works and bring them closer to practical MPC, without any trade-offs or downsides.

Our result is obtained by a novel use of the SPDZ2k trick in the context of sublinear distributed product checks. Compared to Galois rings, the SPDZ2k trick is much more efficient both in terms of computation and representing ring elements: it only requires working over the ring $\mathbb{Z}_{2^{k+s}}$ where s is roughly the statistical security parameter, which for $k \approx s$ is only about twice the size as $\mathbb{Z}_{2^k}$ (whereas the element size of a Galois ring would be $O(k \cdot s)$), and furthermore computing over this ring is quite efficient as it is simple arithmetic modulo a power of two (instead of large-degree polynomials over these rings). Sadly, the SPDZ2k trick is far less flexible and way less algebraically elegant than using Galois rings: it is only useful for enabling a one-degree version of Schwartz-Zippel lemma (which is limited but is already handy, *e.g.*, for enabling MACs [19]), and it does not enable things like polynomial interpolation or general Schwartz-Zippel. Unfortunately, these properties are *essential* in the design of *all* existing sublinear distributed product checks [10–12]. In other words: it is not known how to obtain any form of sublinear checks *without* relying on polynomial interpolation, which over $\mathbb{Z}_{2^k}$ requires Galois rings.

Our main conceptual contribution lies in providing an alternative to the recursion tricks used in existing sublinear distributed product checks, without using polynomial interpolation. Instead, we leverage the SPDZ2k trick, avoiding expensive Galois ring extensions altogether. This turns out to be highly non-trivial given that *all* existing distributed checks use polynomial interpolation, and as we mentioned earlier the SPDZ2k trick is not that flexible nor "algebraically-friendly". We believe our ideas can be used in other contexts where sublinearity is required but standard techniques such as polynomial interpolation are expensive.

Remark 1. *In this work, our goal is to make sublinear checks over rings concretely efficient. Unfortunately, as we have argued above, the current use of Galois rings incurs large computational overheads. By avoiding Galois ring extensions and using lightweight tools, we achieve concrete efficiency as demonstrated by the experiments in Section 6. However, we do not discard some "hybrid" approach that uses Galois rings in a different way, and we leave this to future work.*

1.2 Technical Overview

Recall that our task is to verify the correctness of a set of inner products $\{(\llbracket \boldsymbol{a}_i \rrbracket_k, \llbracket \boldsymbol{b}_i \rrbracket_k, \llbracket c_i \rrbracket_k)\}_{i=1}^m$, where $\llbracket \cdot \rrbracket_k$ denotes RSS over $\mathbb{Z}_{2^k}$, introduced formally in Section 2.3 (here bold letters denote vectors of values and $\llbracket \boldsymbol{x} \rrbracket_k$ denotes a vector of replicated secret sharings over $\mathbb{Z}_{2^k}$, one for each value in $\boldsymbol{x}$). There are overall two broad steps in our verification protocol. First, note that the underlying vectors whose inner products we aim to check are secret-shared and, in general, they may be *secret*, that is, no single party may know these values.[6] The first step is then to turn the task of checking an inner product where the secrets are unknown to any party, to multiple instances where the values to check are actually known to some party. The second step is to perform the check in this new setting where there is a prover who knows the underlying secrets. This is done via some recursive check in which the prover—who knows the underlying secret—derives certain "compressed"

[5] Our code is available in https://github.com/AntCPLab/malicious_3pc_arithmetic.

[6] This is the case for instance when verifying a passive MPC computation where the underlying secrets correspond to the intermediate wires in the computation, which are unknown to any individual party.

values which assist in the verification. This is where our biggest contributions are: instead of using interpolation-based "compression", which requires Galois rings and is unfortunately quite expensive computationally, we devise an interpolation-free method inspired by the SPDZ2k trick.

1.2.1 Reducing to Single-Prover Case. The first step, as in [12], is reducing the check of shared inner-product triples $\{(\llbracket \boldsymbol{a}_i \rrbracket_k, \llbracket \boldsymbol{b}_i \rrbracket_k, \llbracket c_i \rrbracket_k)\}_{i=1}^{m}$, where the secrets are unknown to any individual party, to the check of new shared inner-product triples $\{(\llbracket \boldsymbol{\mu}_i \rrbracket_k, \llbracket \boldsymbol{\nu}_i \rrbracket_k, \llbracket w_i \rrbracket_k)\}_{i=1}^{p}$ where all the secrets are known to a single party, which we refer to as the *prover*. Following similar ideas from [12], this works by letting the parties first take a random linear combination of the original triples, reducing the check to a single inner product, and then using a multiplication protocol on replicated shares to turn this inner product into multiple products where there is a prover who knows the underlying shares.

The key to the above reduction is, by using the multiplication protocol on replicated shares, each party can locally compute an additive share of the multiplication result (via an inner-product computation). This further enables that each single party (acting as a prover) who owns an additive share of the multiplication result, can share his additive share with the replicated secret sharing scheme to the other parties (acting as verifiers); then through local conversions, the parties can obtain replicated secret sharings of all the values involved in the local inner-product computation conducted by the prover for his additive share, thus forming a new distributed inner-product triple where all the secrets are known by the single prover. (Apart from the correctness of the new distributed triple, the consistency between the replicated shares distributed by the prover and the shares of the original multiplication results also needs to be checked; this can be handled with ease. See Section 3.)

The same approach works in our case, except that [12] uses Galois ring extensions to reduce to a *single* inner-product check per prover to achieve negligible soundness error. This is the first place where the large-degree ring extensions are needed: to ensure a random linear combination achieves negligible soundness error.

In our case, we repeat the random linear combinations (over $\mathbb{Z}_2$) λ times to achieve negligible soundness error $2^{-\lambda}$ (where λ is the security parameter) and still work over $\mathbb{Z}_{2^k}$. And each prover now needs to prove λ inner-product triples over $\mathbb{Z}_{2^k}$.

After the reduction, each inner-product triple is of the same dimension d that is the summation of the dimensions of the original triples as we conduct a linear combination. The number of inner-product triples p is the security parameter λ as we repeat λ times.

1.2.2 Review Sublinear Distributed Checks for Single Prover. Once in the single-prover setting, works such as [10–12] execute a *recursive* check that takes $O(\log d)$ rounds, where each round includes two main steps: (1) a step that splits the inner-product triple into several shorter inner-product triples of equal dimension, and (2) a step that merges the shorter-dimension inner-product triples into one. In this way, each round, the dimension of the inner-product to be checked is reduced by a constant factor. After a logarithmic number of rounds, the task is reduced to verifying a single multiplication triple which can be done efficiently.

More concretely, the first step is done by asking the prover to share the result of each shorter inner-product triple, and the second step is approached by polynomial interpolation with Schwartz-Zippel lemma. This is the second place where the large-degree ring extensions are needed: to perform polynomial interpolation and guarantee low cheating probability with Schwartz-Zippel.

1.2.3 Our Solution for Single Prover. Trying to avoid the heavy computational cost of Galois ring extensions, we resort to an entirely different approach. In essence, we still keep the recursive idea with two steps in each iteration: one that splits the inner-product triple into triples of smaller dimensions, and one that merges the shorter-dimension triples into one. But we perform these steps in a fundamentally different way, by using ideas reminiscent of the SPDZ2k work [19] which only require a ring of the form $\mathbb{Z}_{2^{k+s}}$, instead of an inefficient large-degree ring extension.

First, before starting with the recursion, the parties "lift" the inner-product triples that are defined over $\mathbb{Z}_{2^k}$ to a ring $\mathbb{Z}_{2^{k+s}}$, by simply interpreting the shares over $\mathbb{Z}_{2^k}$ as elements of $\mathbb{Z}_{2^{k+s}}$. (This is exactly the property we need from RSS; the same property does not hold for Shamir's.) However, this leads to shares of an incorrect secret, since an error of the form $2^k q$ may be introduced due to the wrap-around of the shares. Fortunately, we use the crucial observation that the prover knows all shares and hence, knows what this wrap-around is. This enables the prover to "correct" these mod 2^{k+s} sharings by distributing shares of this overflow, hence reducing the check from $\mathbb{Z}_{2^k}$ to $\mathbb{Z}_{2^{k+s}}$. The communication cost of this step is distributing p replicated secret sharings, one for each inner-product triple. Recall that p is the number of input inner-product triples, which equals to the security parameter. Thus the cost is sublinear in the circuit size.

Now, the advantage of having the check over $\mathbb{Z}_{2^{k+s}}$ is that we can use the SPDZ2k trick: the potential original errors in the lower k bits we want to catch will w.h.p. remain within the top $k+s$ bits after taking a random linear combination over $\mathbb{Z}_{2^{k+s}}$. This resolves the first place of [10] that requires a large-degree extension ring for doing random linear combinations.

After lifting to $\mathbb{Z}_{2^{k+s}}$, we compute a random linear combination of the p inner-product triples and obtain a single inner-product triple of dimension $p \cdot d$, denoted by $(\llbracket \boldsymbol{x} \rrbracket_{k+s}, \llbracket \boldsymbol{y} \rrbracket_{k+s}, \llbracket z \rrbracket_{k+s})$. Now we try to mimic the sublinear distributed product check in [10] to recursively reduce the dimension of the triple we need to check. We still split the inner-product triples into triples of smaller dimensions as in [10]. But when we try to merge the triples, the polynomial trick and the Schwartz-Zippel lemma fail since we work in $\mathbb{Z}_{2^{k+s}}$ rather than a large-degree ring extension.

For a compression parameter q, our goal is to reduce the dimension by a factor of q, i.e., $p \cdot d/q$. We first follow the previous approach and split the triple of dimension $p \cdot d$ to q smaller triples $\{(\llbracket \boldsymbol{x}_i \rrbracket_{k+s}, \llbracket \boldsymbol{y}_i \rrbracket_{k+s}, \llbracket z_i \rrbracket_{k+s})\}_{i=1}^{q}$, each of dimension $p \cdot d/q$. To merge these q triples, we consider computing a new target inner-product where the first input is a *random linear combination* of $\{\llbracket \boldsymbol{x}_i \rrbracket_{k+s}\}_{i=1}^{q}$ and the second input is *another random linear combination* of $\{\llbracket \boldsymbol{y}_i \rrbracket_{k+s}\}_{i=1}^{q}$. To this end, we ask the prover to also share $z_{i,j} = \boldsymbol{x}_i \cdot \boldsymbol{y}_j$ for all i, j. So one may view that we obtain q^2 inner-product triples $\{(\llbracket \boldsymbol{x}_i \rrbracket_{k+s}, \llbracket \boldsymbol{y}_j \rrbracket_{k+s}, \llbracket z_{i,j} \rrbracket_{k+s})\}_{i,j=1}^{q}$.

Now the computation towards the target inner-product triple can be viewed as two random linear combinations of all inner-product triples: one time over the index i (for the first input) and one time over the index j (for the second input). Thus, relying on the SPDZ2k trick again, we may argue that the error remains in $\mathbb{Z}_{2^{k+s}}$ with overwhelming probability. This resolves the second place of [10] that requires a large-degree extension ring.

1.2.4 Soundness Analysis. A naive security analysis based on SPDZ2k shows that, to ensure negligible soundness error, s needs to be as large as $O(\lambda \cdot \log(pd))$ where λ is the security parameter. This in general is too large to be practical. We note that the analysis based on SPDZ2k is very pessimistic. To obtain a better bound on s, we measure the error by the number of 2-factors and design a security game that mimics the random linear combination and the dimension reduction steps in our protocol. We give a fine-grained security analysis of this game and show that $s = \lambda + O(\log \lambda \cdot \log(pd))$ is sufficient to achieve λ-bit security. (See Lemma 5.1 and Section 5.1.) We also show how to boost the soundness of our protocol by doing a light-weight repeating. With this optimization, we can achieve $\lambda = 40$ bit security by setting $s = 64$. (See Section 5.2.)

2 Preliminaries

2.1 Basic Notation

Let $\mathcal{P} = \{P_1, \ldots, P_n\}$ be a set of n parties, and t a threshold s.t. $n = 2t + 1$. We denote by $[n]$ the set $\{1, \cdots, n\}$. We use $\mathbb{Z}_{2^k}$ for the ring modulo 2^k, and sometimes use $\equiv_k$ to explicitly represent congruence modulo 2^k. We use bold letters to denote vectors of values. Let λ be the statistical security parameter, and σ the computational security parameter. We use PRG_K for pseudo-random generators with a key $K \in \{0,1\}^\sigma$ to generate randomness, where the target domain will be clear from context.

For positive integers k, s, we define a function $\mathsf{Po2} : \mathbb{Z}_{2^{k+s}} \rightarrow \{0, 1, \ldots, k+s\}$ that maps a given input to the number of 2-factors in its prime decomposition: $\mathsf{Po2}(0) = k+s$, and for all $x \in \mathbb{Z}_{2^{k+s}} \setminus \{0\}$, $\mathsf{Po2}(x)$ is the largest ℓ s.t. 2^ℓ divides x.

2.2 Security Model

In this work, we focus on multi-party setting in an honest majority against a malicious adversary controlling up to $t = (n-1)/2$ corrupted parties. We assume that every two of the parties are connected via a secure (private and authentic) synchronous channel. We focus on security with (selective) *abort*, where the adversary can instruct all functionalities to send an abort signal to (some of) the parties, which then halt. We use the client-server model, and only consider the case where the adversary controls exactly $t = (n-1)/2$ parties (see Appendix B for more details).

2.3 Replicated Secret Sharing

A t-out-of-n replicated secret sharing (RSS) scheme [12, 34] in the setting of $n = 2t + 1$ consists of the following two procedures:

- $\mathsf{share}(x, D)$: This procedure allows a dealer D to distribute a secret x to the parties, and each party gets a sharing of the secret x. Specifically, to share a secret x, the dealer samples random additive share x_T for every set $T \subseteq \mathcal{P}$ with $|T| = t + 1$, that is, $x = \sum_{T \subset \mathcal{P}:|T|=t+1} x_T$. Then for each share x_T, the dealer hands it to the parties in T.
- $\mathsf{reconstruct}([\![x]\!], D)$: This procedure allows the parties to reveal the secret x to the party D. To do this, for each $T \subseteq \mathcal{P}$ with $|T| = t + 1$ and $D \notin T$,[7] one party in T (say, the one with smallest index) sends its share x_T to D. Then each party sends a *hash* of his share of $[\![x]\!]$ to D. After receiving these messages from all parties, D checks the consistency of the hashes for $[\![x]\!]$. If any inconsistency is detected, it aborts the procedure; otherwise it reconstructs the secret x by computing $x = \sum_{T \subset \mathcal{P}:|T|=t+1} x_T$.
  To reconstruct N secrets each party must receive $\binom{n-1}{t+1} \cdot N$ missing shares in total, which corresponds to the number of sets of size $t + 1$ that this party does not belong to, and also $n - 1$ hash digests.

Below we use $[\![\cdot]\!]_k$ to denote the RSS scheme over $\mathbb{Z}_{2^k}$. When k is clear from context or talking about general cases, we simply write $[\![x]\!]$. For a vector $\boldsymbol{x}$, we use $[\![\boldsymbol{x}]\!]$ (w.r.t. $[\![\boldsymbol{x}]\!]_k$) to denote a vector of replicated secret sharings, one for each value in $\boldsymbol{x}$.

RSS satisfies several useful properties we will make use of throughout our work. We list them below, and refer the reader to Appendix A for details.

- *Pairwise Consistency.* It is possible for the parties to check that they receive *consistent* shares, meaning each party in a set T receives the same term x_T.
- *Linear Operations.* It is possible to perform affine operations on secret-shared data locally. For adding a public value, only a set of parties T_0 of size $t + 1$ needs to know the value.
- *Local Multiplication.* It is possible to locally multiply two sharings $[\![x]\!]_k, [\![y]\!]_k$ to obtain additive sharings of the product. We denote this operation by $\langle x \cdot y \rangle = [\![x]\!]_k \cdot [\![y]\!]_k$.
- *Local conversion.* Given a sharing $[\![x]\!]$, the parties can *locally* obtain sharings $[\![x_S]\!]$ for every $S \subseteq \mathcal{P}$ with $|S| = t + 1$.
- *Modulo reduction.* For a secret sharing $[\![x]\!]_{k+s}$ where $x \in \mathbb{Z}_{2^{k+s}}$ where s is a positive integer, the parties can locally obtain $[\![x \bmod 2^k]\!]_k$.

2.4 Some Ideal Functionalities

As in [11], we assume instantiations for some functionalities in order to sample shares of random values, sample public coins, and also distribute shared inputs. The functionalities are described at a high level below. For instantiations we refer the reader to Appendix C.1.

- $\mathcal{F}_{\text{rand}}$: This functionality samples a random $r \in \mathbb{Z}_{2^k}$, and distributes a sharing $[\![r]\!]$. This can be instantiated with the help of a shared key setup and a PRG non-interactively.
- $\mathcal{F}_{\text{coin}}$: This functionality samples a random $r \in \mathbb{Z}_{2^k}$, and distributes the public value r.
- $\mathcal{F}_{\text{input}}$: Here, a party P_i provides as input a value $x \in \mathbb{Z}_{2^k}$, and the functionality distributes shares $[\![x]\!]$ to the parties.

[7] In particular, D could be a client and in this case $D \notin T$ for every set T.

2.5 Actively Secure MPC from Product Verification

Using the RSS (or in fact, any linear secret sharing) scheme, a general template to design an MPC protocol for a given arithmetic circuit is to (1) distribute shares of the inputs, (2) use linearity to handle addition gates, (3) use some actively secure multiplication protocol to handle multiplication gates, and (4) reconstruct the output at the end of the computation. As in [12], we consider an instantiation of the multiplication by first using *any* passively secure multiplication protocol that preserves privacy under the presence of a malicious adversary, followed by a sublinear check that ensures that all the products were executed correctly, while involving a communication that is sublinear in the number of multiplication gates. We describe these different components below.

Passive Multiplication Functionality. We let $\mathcal{F}_{\mathrm{mult}}$ be a functionality that takes as input consistent sharings $\llbracket x \rrbracket$, $\llbracket y \rrbracket$, and outputs consistent sharings $\llbracket x \cdot y + \epsilon \rrbracket$, where ϵ is some additive error chosen by the adversary. This can be instantiated in multiple ways, and our techniques are agnostic to the underlying implementation. We discuss multiple instantiations in Appendix C.2.

Remark 2 (On inner products). *It is common for many applications (such as these in the context of machine learning) to make use of inner products. Instantiations of $\mathcal{F}_{mult}$ such as the ones mentioned in Appendix C.2 can be easily generalized to handle inner products $\llbracket \boldsymbol{x} \cdot \boldsymbol{y} \rrbracket \leftarrow \llbracket \boldsymbol{x} \rrbracket \cdot \llbracket \boldsymbol{y} \rrbracket$ involving the same communication as a single multiplication. Our verification techniques accommodate for this case, and we present them in this more general setting.*

Inner-Product Checking Functionality. We let $\mathcal{F}_{\mathrm{VrfySSIP}}$[8] be a functionality that, on input a series of secret-shared inner products $\{(\llbracket \boldsymbol{a}_i \rrbracket_k, \llbracket \boldsymbol{b}_i \rrbracket_k, \llbracket c_i \rrbracket_k)\}_{i=1}^m$, and indicates the parties whether the inner products are correct or not. See Section 3.1 for more details.

3 Reducing Distributed Prover to Single Prover Checks

The ability to check secret-shared inner-products $\{(\llbracket \boldsymbol{a}_i \rrbracket_k, \llbracket \boldsymbol{b}_i \rrbracket_k, \llbracket c_i \rrbracket_k)\}_{i=1}^m$ lies at the core of the full MPC protocol described in Appendix C, and it is the central component that enables compiling passive-to-active security with only sublinear — in fact logarithmic — communication overhead. We can interpret this as a proof that these tuples lie in a given language (*i.e.* the language of correct inner-products), where the input is distributed (*i.e.* secret-shared) among multiple *provers* (i.e., the distributed-prover case).

It turns out that having a single prover who knows the underlying secrets (i.e., the single-prover case) greatly helps in designing an efficient instantiation, and the bulk of our work will focus on this particular scenario. In this section, we show that a verification protocol for the single-prover case can be used to design a verification protocol for the distributed-prover case. First, we define in Section 3.1 the corresponding functionalities: $\mathcal{F}_{\mathrm{VrfySSIP}}$ for the distributed-prover case, and $\mathcal{F}_{\mathrm{VrfyIP}}$ for the single-prover scenario. Then, in Section 3.3 we provide the concrete instantiation of $\mathcal{F}_{\mathrm{VrfySSIP}}$ in the $\mathcal{F}_{\mathrm{VrfyIP}}$-Hybrid Model (other functionalities like $\mathcal{F}_{\mathrm{coin}}$ are used in this instantiation, as we will see).

FUNCTIONALITY 3.0.1. *($\mathcal{F}_{VrfySSIP}$ - Verifying Secret-Shared Inner-Product Triples).*

Let $\mathcal{S}$ be the ideal world adversary.

1. $\mathcal{F}_{\mathrm{VrfySSIP}}$ receives m from all parties. Then for all $i \in [m]$, $\mathcal{F}_{\mathrm{VrfySSIP}}$ receives honest parties' shares of $(\llbracket \boldsymbol{a}_i \rrbracket_k, \llbracket \boldsymbol{b}_i \rrbracket_k, \llbracket c_i \rrbracket_k)$. For each replicated secret sharing, $\mathcal{F}_{\mathrm{VrfySSIP}}$ checks whether honest parties' shares satisfy pairwise consistency.
   - If any inconsistency is detected, $\mathcal{F}_{\mathrm{VrfySSIP}}$ sends honest parties' shares to $\mathcal{S}$. Then for each honest party, $\mathcal{F}_{\mathrm{VrfyIP}}$ receives an output from $\mathcal{S}$ and passes it to the honest party as the output of the functionality. (In this case, we essentially give up the security of honest parties. See Remark 3.)
   - Otherwise, $\mathcal{F}_{\mathrm{VrfySSIP}}$ reconstructs the whole sharings $(\llbracket \boldsymbol{a}_i \rrbracket_k, \llbracket \boldsymbol{b}_i \rrbracket_k, \llbracket c_i \rrbracket_k)$ for all $i \in [m]$, and sends the shares of corrupted parties to $\mathcal{S}$. In addition, $\mathcal{F}_{\mathrm{VrfySSIP}}$ computes $\epsilon_i \equiv_k c_i - \boldsymbol{a}_i \cdot \boldsymbol{b}_i$ and sends ϵ_i to $\mathcal{S}$. Then $\mathcal{F}_{\mathrm{VrfySSIP}}$ goes to the next step.
2. $\mathcal{F}_{\mathrm{VrfySSIP}}$ checks if the equation $c_i \equiv_k \boldsymbol{a}_i \cdot \boldsymbol{b}_i$ holds for all $i \in [m]$.
   - If it doesn't hold for some $i \in [m]$, $\mathcal{F}_{\mathrm{VrfySSIP}}$ sends abort to all honest parties and $\mathcal{S}$.
   - Otherwise, $\mathcal{F}_{\mathrm{VrfySSIP}}$ receives a command out $\in$ {accept, abort} from $\mathcal{S}$ and sends out to all honest parties.

3.1 Functionalities for Inner-Product Verification

3.1.1 Distributed Prover. First, we formalize $\mathcal{F}_{\mathrm{VrfySSIP}}$ as Functionality 3.0.1, which corresponds to the functionality that checks for the correctness of the inner-products in the case where the underlying secrets are not necessarily known to any particular party. We assume these m triples are computed by an inner-product protocol that is secure up to additive attacks (see Appendix C.2) and thus revealing the additive error of each inner-product triple to the ideal adversary is allowed.

3.1.2 Single Prover. Recall that our goal is to instantiate $\mathcal{F}_{\mathrm{VrfySSIP}}$ by first reducing it to a check in which the underlying secrets are known to a particular party. The corresponding functionality, which we denote by $\mathcal{F}_{\mathrm{VrfyIP}}$, appears formally as Functionality 3.0.2.

Remark 3. *Note that in functionalities $\mathcal{F}_{VrfySSIP}$ and $\mathcal{F}_{VrfyIP}$, privacy is given up if the input sharings are not pairwise consistent. This must be guaranteed via extra checks by the outside protocol that calls these functionalities (see Appendix A). This is a common pattern in MPC protocols. See Appendix C for more details.*

3.2 Recap of the Approach in [12]

In order to instantiate the distributed-prover check $\mathcal{F}_{\mathrm{VrfySSIP}}$ based on the single-prover one $\mathcal{F}_{\mathrm{VrfySSIP}}$, we follow a similar approach as in [12]. Consider m secret-shared inner-products (of potentially different dimensions) $\{(\llbracket \boldsymbol{a}_i \rrbracket_k, \llbracket \boldsymbol{b}_i \rrbracket_k, \llbracket c_i \rrbracket_k)\}_{i=1}^m$, where $|\boldsymbol{a}_i| = |\boldsymbol{b}_i| =$

[8] This stands for *verify secret-shared inner products.*

FUNCTIONALITY 3.0.2. *($\mathcal{F}_{VrfyIP}$ - Verifying Secret-Shared Inner-Product Triples Known by A Single Party).*

Let $\mathcal{S}$ be the ideal world adversary.

(1) $\mathcal{F}_{\text{VrfyIP}}$ receives the prover's identity j, a parameter p, and honest parties' shares of $\{([\![\boldsymbol{\mu}_i]\!]_k, [\![\boldsymbol{\nu}_i]\!]_k, [\![w_i]\!]_k)\}_{i=1}^{p}$. For each replicated secret sharing, $\mathcal{F}_{\text{VrfyIP}}$ checks whether honest parties' shares satisfy pairwise consistency.
   - If any inconsistency is detected, $\mathcal{F}_{\text{VrfyIP}}$ sends the identity j and honest parties' shares to $\mathcal{S}$. Then for each honest party, $\mathcal{F}_{\text{VrfyIP}}$ receives an output from $\mathcal{S}$ and passes it to the honest party as the output of the functionality. (In this case, we essentially give up the security of honest parties. See Remark 3.)
   - Otherwise, $\mathcal{F}_{\text{VrfyIP}}$ reconstructs the whole sharings $([\![\boldsymbol{\mu}_i]\!]_k, [\![\boldsymbol{\nu}_i]\!]_k, [\![w_i]\!]_k)$ for all $i \in [p]$, and sends the identity j and the shares of corrupted parties to $\mathcal{S}$. In addition, if P_j is corrupted, $\mathcal{F}_{\text{VrfyIP}}$ also sends the whole sharings $\{([\![\boldsymbol{\mu}_i]\!]_k, [\![\boldsymbol{\nu}_i]\!]_k, [\![w_i]\!]_k)\}_{i=1}^{p}$ to $\mathcal{S}$. Then $\mathcal{F}_{\text{VrfyIP}}$ goes to the next step.

(2) $\mathcal{F}_{\text{VrfyIP}}$ checks if $w_i \equiv_k \boldsymbol{\mu}_i \cdot \boldsymbol{\nu}_i$ holds for all $i \in [p]$.
   - If it doesn't hold for some $i \in [p]$, $\mathcal{F}_{\text{VrfyIP}}$ sends abort to all honest parties and $\mathcal{S}$.
   - Otherwise, $\mathcal{F}_{\text{VrfyIP}}$ receives a command out $\in$ {accept, abort} from $\mathcal{S}$ and sends out to all honest parties.

δ_i, $[\![\boldsymbol{a}_i]\!]_k = ([\![a_{i,1}]\!]_k, \ldots, [\![a_{i,\delta_i}]\!]_k)$, $[\![\boldsymbol{b}_i]\!]_k = ([\![b_{i,1}]\!]_k, \ldots, [\![b_{i,\delta_i}]\!]_k)$. Our goal is to verify that for all $i \in [m]$, $c_i \equiv_k \boldsymbol{a}_i \cdot \boldsymbol{b}_i \equiv_k \sum_{j=1}^{\delta_i} a_{i,j} \cdot b_{i,j}$. In [12], this is approached by first letting all parties compute a random linear combination of all inner-product triples so that the verification task is reduced to verifying a single inner-product triple $([\![\boldsymbol{a}]\!], [\![\boldsymbol{b}]\!], [\![c]\!])$ of dimension $\delta = \sum_{i=1}^{m} \delta_i$. Then by the property of the replicated secret sharing scheme, all parties can locally compute an additive sharing $\langle c \rangle := \langle \boldsymbol{a} \cdot \boldsymbol{b} \rangle = [\![\boldsymbol{a}]\!] \cdot [\![\boldsymbol{b}]\!]$ (see Appendix A). Let $c^{(j)}$ denote the j-th share of $\langle c \rangle$, which is held by party P_j. Now, each party P_j distributes replicated shares of $c^{(j)}$, as $[\![c^{(j)}]\!]$. Then all parties check that

(1) Each party P_j correctly computes and shares $[\![c^{(j)}]\!]$.
(2) The secret of $[\![c]\!]$ is identical to the secret of $\sum_{j=1}^{n} [\![c^{(j)}]\!]$.

However, this approach only works over a large enough ring extension (or a large enough finite field), and does not work over the ring $\mathbb{Z}_{2^k}$. When we work over a (large enough) ring extension, a random linear combination of all inner-product triples satisfies that if one of the inner-product triples is incorrect, then the resulting inner-product triple is also incorrect with overwhelming probability (to be more concrete, the failure probability is roughly the inverse of the so-called *Lenstra constant* of the ring extension). Thus, checking the resulting inner-product triple is sufficient. However, recall that the main goal in our work is to operate over the ring $\mathbb{Z}_{2^k}$ *directly*, and if we attempt to follow the template in [12] in this case, the failure probability can be as large as 1/2: consider an example where there is a single inner-product triple with additive error $\epsilon = 2^{k-1}$. Then as long as the random coefficient is a multiple of 2, the additive error will vanish in the resulting triple.

3.3 Instantiating $\mathcal{F}_{\text{VrfySSIP}}$ in the $\mathcal{F}_{\text{VrfyIP}}$-Hybrid Model

Our idea is to boost the 1/2 soundness that stems from using $\mathbb{Z}_{2^k}$ by repeating the approach above λ times. This ensures that, if one of the m inner-product triples is incorrect, then one of the λ resulting inner-product triples is also incorrect with probability $1 - 2^{-\lambda}$. We note that it is sufficient to use random coefficients over $\mathbb{Z}_2$ rather than $\mathbb{Z}_{2^k}$ to achieve the same effect.[9] The idea of using random bits as coefficients is also used in several previous works such as [39]. To generate the random coefficients, we let all parties jointly sample a random PRG seed which is expanded locally by each party.

After reducing the original verification task to the one of verifying λ inner-product triples $\{([\![\boldsymbol{a}'_i]\!]_k, [\![\boldsymbol{b}'_i]\!]_k, [\![c'_i]\!]_k)\}_{i=1}^{\lambda}$, all parties locally compute an additive sharing of c'_i over $\mathbb{Z}_{2^k}$, denoted by

$$\langle c'_i \rangle_k = ({c'}_i^{(1)}, \ldots, {c'}_i^{(n)}) = [\![\boldsymbol{a}'_i]\!]_k \cdot [\![\boldsymbol{b}'_i]\!]_k \qquad (1)$$

where ${c'}_i^{(j)}$ is the local additive share of c'_i held by party P_j for all $i \in [\lambda]$. Then each party P_j shares his additive share ${c'}_i^{(j)}$ as $[\![{c'}_i^{(j)}]\!]_k$, for each $i \in [\lambda]$. We will check that

(1) For all $i \in [\lambda]$, each party P_j correctly computes and shares their additive share $[\![{c'}_i^{(j)}]\!]_k$.
(2) For all $i \in [\lambda]$, the secret of $[\![c'_i]\!]_k$ is identical to the secret of $\sum_{j=1}^{n} [\![{c'}_i^{(j)}]\!]_k$.

For the first task, as noted in [12], ${c'}_i^{(j)}$ can be computed as an inner product of the j-th shares of $[\![\boldsymbol{a}'_i]\!]_k, [\![\boldsymbol{b}'_i]\!]_k$. Also, all parties can locally convert $[\![\boldsymbol{a}'_i]\!]_k, [\![\boldsymbol{b}'_i]\!]_k$ to replicated secret sharings of the j-th shares of $[\![\boldsymbol{a}'_i]\!]_k, [\![\boldsymbol{b}'_i]\!]_k$. At this point, we can make use of the functionality $\mathcal{F}_{\text{VrfyIP}}$ (described in Functionality 3.0.2), which allows a single prover to prove the correctness of λ inner-product triples that are shared all parties. For the second task, for each $i \in [\lambda]$, we simply compute $[\![o_i]\!]_k = [\![c'_i]\!]_k - \sum_{j=1}^{n} [\![{c'}_i^{(j)}]\!]_k$ and reconstruct o_i to check whether $o_i \equiv_k 0$. We summarize these ideas in protocol Π_{VrfySSIP} (Protocol 3.3.1).

LEMMA 3.1. *Let λ be the statistical security parameter and σ be the computational security parameter. Assume that G is a pseudorandom generator. The protocol $\Pi_{VrfySSIP}$ securely computes $\mathcal{F}_{VrfySSIP}$ with abort in the $\{\mathcal{F}_{coin}, \mathcal{F}_{input}, \mathcal{F}_{VrfyIP}\}$-hybrid model against a malicious adversary controlling $t = \frac{n-1}{2}$ corrupted parties.*

The proof of Lemma 3.1 is given in Appendix G.4.

[9] It is important to note that, computationally, using ring extensions is not substantially different from taking multiple linear combinations. Using finite fields as an example: taking a linear combination of values over $\mathbb{F}_2$ with coefficients over $\mathbb{F}_{2^\lambda}$ is the same as taking λ linear combinations over $\mathbb{F}_2$. We use bits for the linear combination but a similar optimization can be done over the ring extension. However, the main benefit of writing such computation directly over $\mathbb{Z}_{2^k}$ is that subsequent computations — in particular our instantiation of $\mathcal{F}_{\text{VrfyIP}}$ — can be designed entirely over $\mathbb{Z}_{2^k}$ instead of a computationally expensive extension.

PROTOCOL 3.3.1. (Π_{VrfySSIP} - *Verifying Secret-Shared Inner-Product Triples).*

1. All parties agree on a PRG G with seed length σ.
2. All parties invoke $\mathcal{F}_{\text{coin}}$ to generate a random seed of size σ. All parties locally expand the seed and obtain random binary coefficients $\boldsymbol{\gamma}_1, \ldots, \boldsymbol{\gamma}_\lambda \in \{0,1\}^m$.
3. For all $i \in [\lambda]$, all parties set
$$\begin{aligned} \llbracket \boldsymbol{a}'_i \rrbracket_k &= (\gamma_{i,1} \cdot \llbracket \boldsymbol{a}_1 \rrbracket_k, \ldots, \gamma_{i,m} \cdot \llbracket \boldsymbol{a}_m \rrbracket_k), \\ \llbracket \boldsymbol{b}'_i \rrbracket_k &= (\llbracket \boldsymbol{b}_1 \rrbracket_k, \ldots, \llbracket \boldsymbol{b}_m \rrbracket_k), \\ \llbracket c'_i \rrbracket_k &= \sum_{j=1}^{m} \gamma_{i,j} \cdot \llbracket c_j \rrbracket_k. \end{aligned}$$
4. For all $i \in [\lambda]$, all parties locally compute an additive sharing $\langle c'_i \rangle_k = ({c'}_i^{(1)}, \ldots, {c'}_i^{(n)}) = \llbracket \boldsymbol{a}'_i \rrbracket_k \cdot \llbracket \boldsymbol{b}'_i \rrbracket_k$. Each party P_j uses $\mathcal{F}_{\text{input}}$ to share ${c'}_i^{(j)}$.
5. **Checking Correctness of Computation**: For each party P_j and for all $i \in [\lambda]$, let $\boldsymbol{\mu}_i^{(j)}$ and $\boldsymbol{\nu}_i^{(j)}$ be vectors deduced from the j-th shares of $\llbracket \boldsymbol{a}'_i \rrbracket_k$ and $\llbracket \boldsymbol{b}'_i \rrbracket_k$ respectively s.t. $c_i'^{(j)} = \boldsymbol{\mu}_i^{(j)} \cdot \boldsymbol{\nu}_i^{(j)}$. All parties locally convert $\llbracket \boldsymbol{a}'_i \rrbracket_k$ and $\llbracket \boldsymbol{b}'_i \rrbracket_k$ to $\llbracket \boldsymbol{\mu}_i^{(j)} \rrbracket_k$ and $\llbracket \boldsymbol{\nu}_i^{(j)} \rrbracket_k$. All parties invoke $\mathcal{F}_{\text{VrfyIP}}$ with input $(j, \lambda, \{(\llbracket \boldsymbol{\mu}_i^{(j)} \rrbracket_k, \llbracket \boldsymbol{\nu}_i^{(j)} \rrbracket_k, \llbracket {c'}_i^{(j)} \rrbracket_k)\}_{i=1}^{\lambda})$.
6. **Checking Zero**: For all $i \in [\lambda]$, all parties locally compute $\llbracket o_i \rrbracket_k = \llbracket c'_i \rrbracket_k - \sum_{j=1}^{n} \llbracket {c'}_i^{(j)} \rrbracket_k$. Then all parties reconstruct the secret o_i using the procedure $\mathsf{reconstruct}(\llbracket o_i \rrbracket_k, P_j)$ for $j \in [n]$.
7. If $\mathcal{F}_{\text{VrfyIP}}$ returns abort or there exists $i \in [\lambda]$ s.t. $o_i \not\equiv_k 0$, all parties abort, otherwise output accept.

4 Instantiating $\mathcal{F}_{\text{VrfyIP}}$ – Verifying Inner-Product Triples with a Single Prover

We focus now on the task of instantiating $\mathcal{F}_{\text{VrfyIP}}$, which takes as input a series of secret-shared inner-product triples $\{(\llbracket \boldsymbol{\mu}_i \rrbracket_k, \llbracket \boldsymbol{\nu}_i \rrbracket_k, \llbracket w_i \rrbracket_k)\}_{i=1}^{p}$ each of dimension d, and outputs if all the tuples are correct. Here, unlike $\mathcal{F}_{\text{VrfySSIP}}$, there is a single prover P_j who knows all the shares. As we have discussed in Section 3, Protocol Π_{VrfySSIP} from Section 3 allows us to reduce the task of verifying multiple secret-shared inner-product triples $\{(\llbracket \boldsymbol{a}_i \rrbracket_k, \llbracket \boldsymbol{b}_i \rrbracket_k, \llbracket c_i \rrbracket_k)\}_{i=1}^{m}$ having a total number of $\delta \geq m$ products, where the underlying secrets are not known by any party (which is precisely the setting that appears when verifying passive MPC computations), to n instances of $\mathcal{F}_{\text{VrfyIP}}$ with $p = \lambda$ and $d = \delta \cdot \binom{n-1}{t}^2$. To be more specific, the p triples $\{(\llbracket \boldsymbol{\mu}_i \rrbracket_k, \llbracket \boldsymbol{\nu}_i \rrbracket_k, \llbracket w_i \rrbracket_k)\}_{i=1}^{p}$ (where $p = \lambda$) to be verified by $\mathcal{F}_{\text{VrfyIP}}$ come from the λ inner-products (each of dimension $d = \delta \cdot \binom{n-1}{t}^2$) computed by each party P_j to obtain his local additive share ${c'}_i^{(j)}$ of c'_i for each $i \in [\lambda]$ (see Equation 1).

In this section, we show how to realize $\mathcal{F}_{\text{VrfyIP}}$ *without using any ring extension*, by only making use of direct operations over a ring of the form $\mathbb{Z}_{2^k}$, and gaining substantial benefits in practice. We will provide a detailed overview of our techniques in what follows before presenting the formal protocols, but in a few sentences, it is worth mentioning that our ideas are achieved by replacing the interpolation-based recursive proofs in [10–12], which require large-degree extensions, by a different recursion approach that still achieves sublinearity without making use of polynomial interpolation. Moving away from the traditional checks using interpolation incurs in a small communication loss due to lack of good properties such as the maximum-distance separability and low dimension of square code, attainable by Reed Solomon codes. However, the cost incurred by ring extensions turns out to be higher, both computationally and also in terms of communication.

We note that for our recursion, we rely on taking linear combinations *à la* SPDZ2k: we use rings of the form $\mathbb{Z}_{2^{k+s}}$ to ensure soundness somewhat proportional to 2^{-s}. We point out that our work is the first in considering the bridge between more “coding theory” techniques, such as the one used for distributed zero-knowledge proofs, and more “number theory” tools such as the SPDZ2k trick.

4.1 Construction of Our Verification Protocol

Protocol Overview. Let $\{(\llbracket \boldsymbol{\mu}_i \rrbracket_k, \llbracket \boldsymbol{\nu}_i \rrbracket_k, \llbracket w_i \rrbracket_k)\}_{i=1}^{p}$ be the set of p tuples to be verified by $\mathcal{F}_{\text{VrfyIP}}$, where each vector has dimension d. Let P_j be the prover, who knows the underlying shares. At a very high level, our verification protocol works as follows. First, we lift each of p input inner-product triples we want to verify from $\mathbb{Z}_{2^k}$ to $\mathbb{Z}_{2^{k+s}}$ for a large enough s such that if the input inner-product triple is incorrect, then after lifting it to $\mathbb{Z}_{2^{k+s}}$, the new triple is still incorrect *when modulo* 2^k. Recall that for the input inner-product triple, the additive error can be as large as 2^{k-1}. Then when we multiply a random coefficient on both sides of the relation, the error will vanish when modular 2^k if the coefficient is even, which happens with probability $1/2$. On the other hand, when working over the ring $\mathbb{Z}_{2^{k+s}}$, the error will vanish only if the coefficient is a multiple of 2^s, which happens with probability 2^{-s}.[10]

In the second step, we compute a random linear combination of the p inner-product triples after lifting so that the task is reduced to verifying a single inner-product triple. As we mentioned above, if there is an incorrect inner-product triple, then the resulting inner-product triple is still incorrect (when modulo 2^{k+s}) with overwhelming probability. Recall that d denotes the dimension of each input inner-product triple of Π_{VrfyIP}. Then the dimension of the inner-product triple obtained in Step 2 is $p \cdot d$. In the third step, we try to adapt the recursion trick in [12]. Concretely, we want to reduce the dimension of the inner-product triple we need to verify by a factor of q (for some parameter q) each round. Then after $\log_q(pd)$ rounds, the task is reduced to verifying a single *multiplication* triple, which can be done efficiently.

However, the approach in [12] heavily relies on the property of finite fields (or ring extensions). In particular,

- In each round, the dimension reduction is achieved relying on techniques related to polynomials. However, polynomial interpolation does not work over the ring $\mathbb{Z}_{2^{k+s}}$.

[10] The idea of using additional s bits was introduced in [19], and it has been used in subsequent works such as [3, 13, 16, 41]. As we will see, however, the security analysis in our case is much more involved than in these previous works, partly stemming from the fact that we use this idea *recursively*.

- A key lemma required in [12] is that if the inner-product triple is incorrect at the beginning of Round i, then after reducing the dimension in this round, the new inner-product triple (whose dimension is reduced by q) is still incorrect with overwhelming probability. This lemma again relies on the property of finite fields (or ring extensions) and does not hold over the ring $\mathbb{Z}_{2^{k+s}}$.

In the following, we will show how we tackle these issues while keeping the communication cost to be sublinear in the size of the input inner-product triples. Let $s \in \mathbb{N}$ be an integer. As we will see later, s will be determined by the security parameter λ. In the following, we will measure the additive error ϵ of an inner-product triple by using $\mathsf{Po2}(\epsilon)$ (defined in Section 2).

4.1.1 Step 1: Lifting Inner-Product Triples to $\mathbb{Z}_{2^{k+s}}$. In the first step, our main observation is that a replicated secret sharing $[\![x]\!]_k$ over $\mathbb{Z}_{2^k}$ can be naturally viewed as a replicated secret sharing $[\![x']\!]_{k+s}$ over $\mathbb{Z}_{2^{k+s}}$, where $x' \in \mathbb{Z}_{2^{k+s}}$ is such that $x' \equiv_k x$. Recall that $[\![x]\!]_k$ is defined by $\{x_T\}_{T\subset\mathcal{P},|T|=t+1}$ and $x \equiv_k \sum_{T\subset\mathcal{P},|T|=t+1} x_T$. Let $x' \equiv_{k+s} \sum_{T\subset\mathcal{P},|T|=t+1} x_T$. Thus the *same* set of shares defines a replicated secret sharing $[\![x']\!]_{k+s}$. In particular, we have $x' \equiv_k x$.

Now, for each $i \in [p]$, all parties view $([\![\boldsymbol{\mu}_i]\!]_k, [\![\boldsymbol{\nu}_i]\!]_k, [\![w_i]\!]_k)$ as replicated secret sharings over $\mathbb{Z}_{2^{k+s}}$: $([\![\boldsymbol{\mu}'_i]\!]_{k+s}, [\![\boldsymbol{\nu}'_i]\!]_{k+s}, [\![w'_i]\!]_{k+s})$. Note that after lifting to $\mathbb{Z}_{2^{k+s}}$, the inner-product triple may not be correct anymore. However, here we make the following crucial observation: the prover party P_j can compute $\boldsymbol{\mu}'_i, \boldsymbol{\nu}'_i, w'_i$ since this party knows the whole sharings $([\![\boldsymbol{\mu}_i]\!]_k, [\![\boldsymbol{\nu}_i]\!]_k, [\![w_i]\!]_k)$ in the clear. This allows P_j to "correct" the inner-products so that they are correct modulo 2^{k+s}. More precisely:

1. P_j computes $h_i \equiv_{k+s} \boldsymbol{\mu}'_i \cdot \boldsymbol{\nu}'_i - w'_i$, which should be a multiple of 2^k if P_j is honest.
2. P_j shares $h_i/2^k$ using $\mathcal{F}_{\text{input}}$ to all parties using the replicated secret sharing over $\mathbb{Z}_{2^s}$. Then all parties locally multiply their shares of $[\![h_i/2^k]\!]_s$ by 2^k modulo 2^{k+s}. In this way, all parties can obtain a replicated secret sharing $[\![h_i]\!]_{k+s}$ such that h_i is a multiple of 2^k. A central observation is that, since the parties are multiplying by 2^k, a malicious prover cannot "correct" the lower-bit error of the inner-product triple by secret-sharing an incorrect $[\![h_i/2^k]\!]_s$.
3. All parties using $[\![h_i]\!]_{k+s}$ to correct the inner-product triple over $\mathbb{Z}_{2^{k+s}}$ by setting $[\![\tilde{w}'_i]\!]_{k+s} := [\![w'_i]\!]_{k+s} + [\![h_i]\!]_{k+s}$.

Note that if the prover P_j is an honest party, then $([\![\boldsymbol{\mu}'_i]\!]_{k+s}, [\![\boldsymbol{\nu}'_i]\!]_{k+s}, [\![\tilde{w}'_i]\!]_{k+s})$ is a correct inner-product triple over $\mathbb{Z}_{2^{k+s}}$. On the other hand, if P_j is a corrupted party and $([\![\boldsymbol{\mu}_i]\!]_k, [\![\boldsymbol{\nu}_i]\!]_k, [\![w_i]\!]_k)$ is incorrect modulo 2^k, then $([\![\boldsymbol{\mu}'_i]\!]_{k+s}, [\![\boldsymbol{\nu}'_i]\!]_{k+s}, [\![\tilde{w}'_i]\!]_{k+s})$ is still incorrect not only modulo 2^{k+s}, but modulo 2^k. In particular, $\tilde{w}'_i - \boldsymbol{\mu}'_i \cdot \boldsymbol{\nu}'_i \equiv_k w_i - \boldsymbol{\mu}_i \cdot \boldsymbol{\nu}_i \not\equiv_k 0$. This means that the additive error $\epsilon_i := \tilde{w}'_i - \boldsymbol{\mu}'_i \cdot \boldsymbol{\nu}'_i$ satisfies that $\mathsf{Po2}(\epsilon_i) \leq k - 1$.

4.1.2 Step 2: Merging into One Inner-Product Triple. In the second step, we reduce the p inner-product triples over $\mathbb{Z}_{2^{k+s}}$ to a single inner-product triple of dimension $p \cdot d$. This is done by generating p random coefficients in $\mathbb{Z}_{2^{k+s}}$ then computing a linear combination of the p triples with these random coefficients. Concretely,

1. All parties invoke $\mathcal{F}_{\text{coin}}$ to prepare p random coefficients $\theta_1, \ldots, \theta_p \in \mathbb{Z}_{2^{k+s}}$.
2. All parties turn to verify $\sum_{i=1}^{p} \theta_i \cdot \boldsymbol{\mu}'_i \cdot \boldsymbol{\nu}'_i \equiv_{k+s} \sum_{i=1}^{p} \theta_i \cdot \tilde{w}'_i$. To this end, all parties set $[\![\boldsymbol{x}]\!]_{k+s} = (\theta_1 \cdot [\![\boldsymbol{\mu}'_1]\!]_{k+s}, \ldots, \theta_p \cdot [\![\boldsymbol{\mu}'_p]\!]_{k+s})$, $[\![\boldsymbol{y}]\!]_{k+s} = ([\![\boldsymbol{\nu}'_1]\!]_{k+s}, \ldots, [\![\boldsymbol{\nu}'_p]\!]_{k+s})$ and $[\![z]\!]_{k+s} = \sum_{i=1}^{p} \theta_i \cdot [\![\tilde{w}'_i]\!]_{k+s}$.

Here we use $([\![\boldsymbol{x}]\!]_{k+s}, [\![\boldsymbol{y}]\!]_{k+s}, [\![z]\!]_{k+s})$ to denote the single triple obtained from merging p triples $\{\boldsymbol{\mu}'_i, \boldsymbol{\nu}'_i, w'_i\}_{i=1}^{p}$. Note that if the prover party P_j is honest, then $([\![\boldsymbol{x}]\!]_{k+s}, [\![\boldsymbol{y}]\!]_{k+s}, [\![z]\!]_{k+s})$ is a correct inner-product triple over $\mathbb{Z}_{2^{k+s}}$. On the other hand, if P_j is corrupted, let $\epsilon_i := \tilde{w}'_i - \boldsymbol{\mu}'_i \cdot \boldsymbol{\nu}'_i$ be the additive error of the i-th inner-product triple. If there exists $i^\star$ such that $\mathsf{Po2}(\epsilon_{i^\star}) \leq k - 1$, then with overwhelming probability, $([\![\boldsymbol{x}]\!]_{k+s}, [\![\boldsymbol{y}]\!]_{k+s}, [\![z]\!]_{k+s})$ is an incorrect inner-product triple over $\mathbb{Z}_{2^{k+s}}$. In fact, we can show an even stronger statement: With probability $1 - 2^{-\lambda}$, the additive error $\epsilon = z - \boldsymbol{x} \cdot \boldsymbol{y}$ is not a multiple of $2^{k+\lambda}$. In other words, with probability $1 - 2^{-\lambda}$, $\mathsf{Po2}(\epsilon) < k + \lambda$.

4.1.3 Step 3: Reducing the Dimension of the Inner-Product Triple. Besides using the larger ring $\mathbb{Z}_{2^{k+s}}$, the techniques described so far carry a close resemblance with previous sublinear distributed zero-knowledge proofs [12]. However, as we will see, the third step is where we fundamentally deviate from the previous template. In this step, we try to reduce the dimension of the inner-product triple $([\![\boldsymbol{x}]\!]_{k+s}, [\![\boldsymbol{y}]\!]_{k+s}, [\![z]\!]_{k+s})$ obtained in Step 2 by a factor of q, where q is a chosen parameter. Recall that d is the dimension of each input inner-product triple $([\![\boldsymbol{\mu}_i]\!]_k, [\![\boldsymbol{\nu}_i]\!]_k, [\![w_i]\!]_k)$. Then the dimension of $([\![\boldsymbol{x}]\!]_{k+s}, [\![\boldsymbol{y}]\!]_{k+s}, [\![z]\!]_{k+s})$ is $d' = pd$. We first review how the line of works [10–12] achieves the dimension reduction.

Review of techniques in [10–12]. At a high level, the vectors $\boldsymbol{x}, \boldsymbol{y}$ are divided into q sub-vectors of the same dimension d'/q: $\boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_q)$ and $\boldsymbol{y} = (\boldsymbol{y}_1, \ldots, \boldsymbol{y}_q)$. Then, we define two vectors of degree-$(q-1)$ polynomials $\boldsymbol{f}(\cdot), \boldsymbol{h}(\cdot)$ such that $\boldsymbol{f}(i) = \boldsymbol{x}_i, \boldsymbol{g}(i) = \boldsymbol{y}_i$ for all $i \in [q]$, and we define a degree-$(2q-2)$ polynomial $h := \boldsymbol{f} \cdot \boldsymbol{g}$ (i.e., the inner-product between $\boldsymbol{f}$ and $\boldsymbol{g}$).

Note that the prover P_j can compute $\boldsymbol{f}(\cdot), \boldsymbol{g}(\cdot), h(\cdot)$ in clear. Now we ask P_j to share $h(i)$ for $i \in \{2, \ldots, 2q-1\}$. Since we should have $\sum_{i=1}^{q} h(i) = z$, all parties compute $[\![h(1)]\!] = [\![z]\!] - \sum_{i=2}^{q} [\![h(i)]\!]$.

At this stage, all parties can use $\{[\![\boldsymbol{f}(i)]\!]\}_{i=1}^{q}$ to compute replicated secret sharings of the coefficients of $\boldsymbol{f}(\cdot)$ via interpolation. Note that interpolation only involves linear operations. Similarly, all parties can use $\{[\![\boldsymbol{g}(i)]\!]\}_{i=1}^{q}$ to compute replicated secret sharings of the coefficients of $\boldsymbol{g}(\cdot)$ via interpolation. And all parties can use $\{[\![h(i)]\!]\}_{i=1}^{2q-1}$ to compute replicated secret sharings of the coefficients of $h(\cdot)$ via interpolation.

Note that, if $\boldsymbol{x} \cdot \boldsymbol{y} \neq z$, since $\boldsymbol{x} \cdot \boldsymbol{y} = \sum_{i=1}^{q} \boldsymbol{f}(i) \cdot \boldsymbol{g}(i)$ and $z = \sum_{i=1}^{q} h(i)$, there exists $i \in [q]$ such that $\boldsymbol{f}(i) \cdot \boldsymbol{g}(i) \neq h(i)$. Thus, it is sufficient to test $h = \boldsymbol{f} \cdot \boldsymbol{g}$.

When we work over a finite field or an extension ring, by the Schwartz–Zippel Lemma, it is sufficient to test a random evaluation point r. Thus, all parties compute and verify $([\![\boldsymbol{f}(r)]\!], [\![\boldsymbol{g}(r)]\!], [\![h(r)]\!])$, which is an inner-product triple of dimension d'/q.

Adapting the above approach over $\mathbb{Z}_{2^{k+s}}$. When we try to adapt the above approach over $\mathbb{Z}_{2^{k+s}}$, we identify the following problems.

- First, over $\mathbb{Z}_{2^{k+s}}$, we cannot do interpolation anymore. This makes the above reduction trick unavailable over $\mathbb{Z}_{2^{k+s}}$.

- Second, even if we can do interpolation, when checking the polynomial relation $h = \boldsymbol{f} \cdot \boldsymbol{g}$, just checking a random evaluation point is not sufficient anymore since the Schwartz–Zippel Lemma no longer holds.

To address these two issues, we first ask the prover party P_j to compute $z_{i,i'} = \boldsymbol{x}_i \cdot \boldsymbol{y}_{i'}$ for all $i, i' \in [q]$ and share those values to all parties using $\mathcal{F}_{\text{input}}$. Then our idea is to check the following inner-product relation of dimension d'/q: $(\sum_{i=1}^{q} \alpha_i \cdot \boldsymbol{x}_i) \cdot (\sum_{i=1}^{q} \beta_i \cdot \boldsymbol{y}_i) = z'$, where α_i, β_i are random coefficients over $\mathbb{Z}_{2^{k+s}}$ and $z' = \sum_{i=1}^{q} \sum_{i'=1}^{q} \alpha_i \beta_{i'} \cdot z_{i,i'}$. Notice that, here, the prover is distributing q^2 sharings instead of q as in the standard approach using interpolation. In fact, we can view the approach in [10–12] as a special case where α_i, β_i are set to be proper Lagrange coefficients, which in turn can be seen as encoding the vectors using a Reed-Solomon (RS) code, whose product is again a RS code of twice the dimension. Later on, a symbol at a random index in these codewords will be sampled, and the low dimension of the square ensures that only a linear (in q) number of extra elements are needed to reconstruct such symbol for the product. In contrast, we can interpret our approach as using a "random code", whose square code in general has *squared* dimension, which is the source of the extra q^2 inputs required by the prover.

Let $\boldsymbol{x}' = \sum_{i=1}^{q} \alpha_i \cdot \boldsymbol{x}_i$ and $\boldsymbol{y}' = \sum_{i=1}^{q} \beta_i \cdot \boldsymbol{y}_i$. We want to argue that, if $(\llbracket \boldsymbol{x} \rrbracket_{k+s}, \llbracket \boldsymbol{y} \rrbracket_{k+s}, \llbracket z \rrbracket_{k+s})$ is an incorrect inner-product triple, then after the dimension reduction, $(\llbracket \boldsymbol{x}' \rrbracket_{k+s}, \llbracket \boldsymbol{y}' \rrbracket_{k+s}, \llbracket z' \rrbracket_{k+s})$ is still an incorrect inner-product triple.

Let $\epsilon := z - \boldsymbol{x} \cdot \boldsymbol{y}$, $\epsilon_{i,i'} = z_{i,i'} - \boldsymbol{x}_i \cdot \boldsymbol{y}_{i'}$ for all $i, i' \in [q]$, and $\epsilon' = z' - \boldsymbol{x}' \cdot \boldsymbol{y}'$. Then we should have (1) $\epsilon = \epsilon_{1,1} + \cdots + \epsilon_{q,q}$, and (2) $\epsilon' = \sum_{i=1}^{q} \sum_{i'=1}^{q} \alpha_i \beta_{i'} \cdot \epsilon_{i,i'} = \sum_{i'=1}^{q} \beta_{i'} \cdot (\sum_{i=1}^{q} \alpha_i \cdot \epsilon_{i,i'})$. Now assume that $\mathsf{Po2}(\epsilon) < k + \lambda$ (Recall that this happens with probability $1 - 2^{-\lambda}$ from the argument in Step 2). From the first condition, there exists $i^\star$ such that $\mathsf{Po2}(\epsilon_{i^\star,i^\star}) < k + \lambda$ as well. From the second condition, we may view that ϵ' is computed by taking two random linear combinations:

- The first combination is to compute $\epsilon'_{i^\star} := \sum_{i=1}^{q} \alpha_i \epsilon_{i,i^\star}$.
- Let $\epsilon'_{i'} := \sum_{i=1}^{q} \alpha_i \epsilon'_{i,i'}$ for all $i' \neq i^\star$. Then the second combination is to compute $\epsilon' = \sum_{i'=1}^{q} \beta_{i'} \cdot \epsilon'_{i'}$.

Following the same argument as that in Step 2, each random linear combination may increase the number of 2-factors in the additive error by less than λ with probability $1 - 2^{-\lambda}$. Thus, with overwhelming probability, $\mathsf{Po2}(\epsilon'_{i^\star}) < \mathsf{Po2}(\epsilon_{i^\star,i^\star}) + \lambda < k + 2\lambda$ and $\mathsf{Po2}(\epsilon') < \mathsf{Po2}(\epsilon'_{i^\star}) + \lambda < k + 3\lambda$.

In summary, our approach of dimension reduction is as follows:

(1) For all $i, i' \in [q]$ and $(i, i') \neq (1, 1)$, the prover P_j computes $z_{i,i'} = \boldsymbol{x}_i \cdot \boldsymbol{y}_{i'}$ and shares $z_{i,i'}$ to all parties over $\mathbb{Z}_{2^{k+s}}$. Then all parties compute $\llbracket z_{1,1} \rrbracket_{k+s} := \llbracket z \rrbracket_{k+s} - \sum_{i=2}^{q} z_{i,i}$. This step is to ensure the first condition holds, "by definition".
(2) All parties invoke $\mathcal{F}_{\text{coin}}$ to randomly sample $\{\alpha_i\}_{i=1}^{q}, \{\beta_i\}_{i=1}^{q}$ in $\mathbb{Z}_{2^{k+s}}$.
(3) All parties locally compute $\llbracket \boldsymbol{x}' \rrbracket_{k+s} = \sum_{i=1}^{q} \alpha_i \cdot \llbracket \boldsymbol{x}_i \rrbracket_{k+s}$, $\llbracket \boldsymbol{y}' \rrbracket_{k+s} = \sum_{i=1}^{q} \beta_i \cdot \llbracket \boldsymbol{y}_i \rrbracket_{k+s}$ and $\llbracket z' \rrbracket_{k+s} = \sum_{i=1}^{q} \sum_{i'=1}^{q} \alpha_i \beta_{i'} \cdot \llbracket z_{i,i'} \rrbracket_{k+s}$.

As we argued above, if the additive error ϵ of $(\llbracket \boldsymbol{x} \rrbracket_{k+s}, \llbracket \boldsymbol{y} \rrbracket_{k+s}, \llbracket z \rrbracket_{k+s})$ satisfies that $\mathsf{Po2}(\epsilon) < k + \lambda$, then with overwhelming probability, the additive error ϵ' of $(\llbracket \boldsymbol{x}' \rrbracket_{k+s}, \llbracket \boldsymbol{y}' \rrbracket_{k+s}, \llbracket z \rrbracket'_{k+s})$ satisfies that $\mathsf{Po2}(\epsilon') < k + 3\lambda$. As we can see the number of 2-factors of the additive error in each iteration grows by 2λ.

4.1.4 Step 4: Checking the Final Multiplication Triple. By repeating Step 3 enough times, we finally end up checking a single multiplication triple. To simplify the verification of the final multiplication triple, we borrow the idea from [12] by inserting a random multiplication triple in the last iteration of the dimension reduction.

Concretely, in the last iteration, all parties hold an inner-product triple $(\llbracket \boldsymbol{x} \rrbracket_{k+s}, \llbracket \boldsymbol{y} \rrbracket_{k+s}, \llbracket z \rrbracket_{k+s})$ of dimension q. We first ask the prover P_j to share a random multiplication triple $(\llbracket x_0 \rrbracket_{k+s}, \llbracket y_0 \rrbracket_{k+s}, \llbracket z_0 \rrbracket_{k+s})$ to all parties. This random triple will be used as a random mask so that the final multiplication triple can be checked by directly reconstructing all three replicated secret sharings. To this end, we modify the dimension reduction procedure as follows:

(1) The prover party P_j randomly samples $x_0, y_0 \in \mathbb{Z}_{2^{k+s}}$. Then P_j shares x_0, y_0 to all parties. These are used to protect the privacy of P_j's secrets.
(2) For all $i, i' \in \{0, 1, \ldots, q\}$ and $(i, i') \neq (1, 1)$, P_j computes $z_{i,i'} = x_i \cdot y_{i'}$ and shares $z_{i,i'}$ to all parties. All parties compute $\llbracket z_{1,1} \rrbracket_{k+s}$ by $\llbracket z_{1,1} \rrbracket_{k+s} := \llbracket z \rrbracket_{k+s} - \sum_{i=2}^{q} z_{i,i}$. Note that P_j shares $z_{0,0} = x_0 \cdot y_0$ to all parties in this step.
(3) All parties invoke $\mathcal{F}_{\text{coin}}$ to randomly sample $\{\alpha_i\}_{i=1}^{q}, \{\beta_i\}_{i=1}^{q}$ in $\mathbb{Z}_{2^{k+s}}$. All parties set $\alpha_0 = \beta_0 = 1$.
(4) All parties compute $\llbracket x' \rrbracket_{k+s} = \sum_{i=0}^{q} \alpha_i \cdot \llbracket x_i \rrbracket_{k+s}$, $\llbracket y' \rrbracket_{k+s} = \sum_{i=0}^{q} \beta_i \cdot \llbracket y_i \rrbracket_{k+s}$ and $\llbracket z' \rrbracket_{k+s} = \sum_{i=0}^{q} \sum_{i'=0}^{q} \alpha_i \beta_{i'} \cdot \llbracket z_{i,i'} \rrbracket_{k+s}$.
(5) All parties reconstruct $\llbracket x' \rrbracket_{k+s}, \llbracket y' \rrbracket_{k+s}, \llbracket z' \rrbracket_{k+s}$ and check whether $z' \equiv_{k+s} x' \cdot y'$. If not, all parties abort.

Observe that when P_j is an honest party, x', y' are masked by x_0, y_0, and $z' = x' \cdot y'$. Thus $(\llbracket x' \rrbracket_{k+s}, \llbracket y' \rrbracket_{k+s}, \llbracket z' \rrbracket_{k+s})$ is a random multiplication triple, which is safe to reconstruct. When P_j is a corrupted party, following the same argument as that in Step 3, with overwhelming probability, the number of 2-factors in the additive error of $(\llbracket x' \rrbracket_{k+s}, \llbracket y' \rrbracket_{k+s}, \llbracket z' \rrbracket_{k+s})$ grows by at most 2λ.

The full description of this protocol Π_{VrfyIP} is in Protocol 4.2.1.

4.2 Communication Cost

We analyze in detail the communication complexity of our protocol in Appendix E. This communication depends quadratically in q and logarithmically in the dimension of the inner products (but it is independent of the number of inner products), which results in sublinear communication in $|C|$ in the context of verifying MPC computations. Interestingly, the check in [10, 11] depends linearly in q, but it turns out we obtain better communication since we avoid large-degree Galois rings. For $\lambda = 40$ and three parties, and taking $q = 4$, verifying $\delta = 2^{20} \approx 1$ million secret-shared products with our protocol requires 142.7 kB, while using ring extensions this requires 636.1 kB, about 5× more communication. For other parameter regimes of interest this factor tends to range between 3 and 5. We remark that the main point of avoiding Galois rings is not saving in communication — which is mostly a good side effect of our protocol — but reducing computation costs. As we will see in Section 6, our improvement in communication fall short when compared to our advantages in terms of computation, which

PROTOCOL 4.2.1. *(Π_{VrfyIP} -Verifying Inner-Product Triples with a Single Prover).*

All parties hold $\{(\llbracket \boldsymbol{\mu}_i \rrbracket_k, \llbracket \boldsymbol{\nu}_i \rrbracket_k, \llbracket w_i \rrbracket_k)\}_{i=1}^{p}$ as input. The prover party P_j in addition learns the whole sharings of $\{(\llbracket \boldsymbol{\mu}_i \rrbracket_k, \llbracket \boldsymbol{\nu}_i \rrbracket_k, \llbracket w_i \rrbracket_k)\}_{i=1}^{p}$. All parties agree on the parameters s and q.

- **Step 1: Lifting Inner-Product Triples to $\mathbb{Z}_{2^{k+s}}$**. For all $i \in [p]$,
  (1) All parties view $(\llbracket \boldsymbol{\mu}_i \rrbracket_k, \llbracket \boldsymbol{\nu}_i \rrbracket_k, \llbracket w_i \rrbracket_k)$ as replicated secret sharings over $\mathbb{Z}_{2^{k+s}}$: $(\llbracket \boldsymbol{\mu}'_i \rrbracket_{k+s}, \llbracket \boldsymbol{\nu}'_i \rrbracket_{k+s}, \llbracket w'_i \rrbracket_{k+s})$.
  (2) P_j computes $h_i \equiv_{k+s} \boldsymbol{\mu}'_i \cdot \boldsymbol{\nu}'_i - w'_i$.
  (3) P_j calls $\mathcal{F}_{\text{input}}$ to distribute shares $\llbracket h_i/2^k \rrbracket_s$. Then all parties locally multiply their shares of $\llbracket h_i/2^k \rrbracket_s$ by 2^k modulo 2^{k+s} and obtain $\llbracket h_i \rrbracket_{k+s}$.
  (4) All parties set $\llbracket \tilde{w}'_i \rrbracket_{k+s} := \llbracket w'_i \rrbracket_{k+s} + \llbracket h_i \rrbracket_{k+s}$.
- **Step 2: Merging into One Inner-Product Triple**.
  (1) All parties invoke $\mathcal{F}_{\text{coin}}$ to sample random coefficients $\theta_1, \ldots, \theta_p \in \mathbb{Z}_{2^{k+s}}$.
  (2) All parties set $\llbracket \boldsymbol{x} \rrbracket_{k+s} = (\theta_1 \cdot \llbracket \boldsymbol{\mu}'_1 \rrbracket_{k+s}, \ldots, \theta_p \cdot \llbracket \boldsymbol{\mu}'_p \rrbracket_{k+s})$, $\llbracket \boldsymbol{y} \rrbracket_{k+s} = (\llbracket \boldsymbol{\nu}'_1 \rrbracket_{k+s}, \ldots, \llbracket \boldsymbol{\nu}'_p \rrbracket_{k+s})$, $\llbracket z \rrbracket_{k+s} = \sum_{i=1}^{p} \theta_i \cdot \llbracket \tilde{w}'_i \rrbracket_{k+s}$.
- **Step 3: Reducing the Dimension of the Inner-Product Triple**. All parties repeat the following steps until the dimension of $(\llbracket \boldsymbol{x} \rrbracket_{k+s}, \llbracket \boldsymbol{y} \rrbracket_{k+s}, \llbracket z \rrbracket_{k+s})$ is at most q.
  (1) For all $i, i' \in [q]$ and $(i, i') \neq (1, 1)$, the prover party P_j computes $z_{i,i'} = \boldsymbol{x}_i \cdot \boldsymbol{y}_{i'}$ and shares $z_{i,i'}$ to all parties using $\mathcal{F}_{\text{input}}$. Then all parties compute $\llbracket z_{1,1} \rrbracket_{k+s}$ by $\llbracket z_{1,1} \rrbracket_{k+s} := \llbracket z \rrbracket_{k+s} - \sum_{i=2}^{q} z_{i,i}$.
  (2) All parties invoke $\mathcal{F}_{\text{coin}}$ to randomly sample $\{\alpha_i\}_{i=1}^{q}, \{\beta_i\}_{i=1}^{q}$ in $\mathbb{Z}_{2^{k+s}}$.
  (3) All parties locally set $\llbracket \boldsymbol{x}' \rrbracket_{k+s} = \sum_{i=1}^{q} \alpha_i \cdot \llbracket \boldsymbol{x}_i \rrbracket_{k+s}$, $\llbracket \boldsymbol{y}' \rrbracket_{k+s} = \sum_{i=1}^{q} \beta_i \cdot \llbracket \boldsymbol{y}_i \rrbracket_{k+s}$ and $\llbracket z' \rrbracket_{k+s} = \sum_{i=1}^{q} \sum_{i'=1}^{q} \alpha_i \beta_{i'} \cdot \llbracket z_{i,i'} \rrbracket_{k+s}$.
  (4) All parties redefine $(\llbracket \boldsymbol{x} \rrbracket_{k+s}, \llbracket \boldsymbol{y} \rrbracket_{k+s}, \llbracket z \rrbracket_{k+s}) := (\llbracket \boldsymbol{x}' \rrbracket_{k+s}, \llbracket \boldsymbol{y}' \rrbracket_{k+s}, \llbracket z' \rrbracket_{k+s})$.
- **Step 4: Checking the Final Multiplication Triple.** All parties hold $(\llbracket \boldsymbol{x} \rrbracket_{k+s}, \llbracket \boldsymbol{y} \rrbracket_{k+s}, \llbracket z \rrbracket_{k+s})$ of dimension at most q.
  (1) The prover P_j randomly samples $x_0, y_0 \in \mathbb{Z}_{2^{k+s}}$. Then P_j calls $\mathcal{F}_{\text{input}}$ to distribute shares $\llbracket x_0 \rrbracket_{k+s}, \llbracket y_0 \rrbracket_{k+s}$ to all parties.
  (2) For all $i, i' \in \{0, 1, \ldots, q\}$ and $(i, i') \neq (1, 1)$, P_j computes $z_{i,i'} = x_i \cdot y_{i'}$ and calls $\mathcal{F}_{\text{input}}$ to distribute shares $\llbracket z_{i,i'} \rrbracket_{k+s}$. Then all parties compute $\llbracket z_{1,1} \rrbracket_{k+s} := \llbracket z \rrbracket_{k+s} - \sum_{i=2}^{q} z_{i,i}$. Note that P_j shares $z_{0,0} = x_0 \cdot y_0$ to all parties in this step.
  (3) All parties invoke $\mathcal{F}_{\text{coin}}$ to randomly sample $\{\alpha_i\}_{i=1}^{q}, \{\beta_i\}_{i=1}^{q}$ in $\mathbb{Z}_{2^{k+s}}$. All parties set $\alpha_0 = \beta_0 = 1$.
  (4) All parties locally set $\llbracket x' \rrbracket_{k+s} = \sum_{i=0}^{q} \alpha_i \cdot \llbracket x_i \rrbracket_{k+s}$, $\llbracket y' \rrbracket_{k+s} = \sum_{i=0}^{q} \beta_i \cdot \llbracket y_i \rrbracket_{k+s}$ and $\llbracket z' \rrbracket_{k+s} = \sum_{i=0}^{q} \sum_{i'=0}^{q} \alpha_i \beta_{i'} \cdot \llbracket z_{i,i'} \rrbracket_{k+s}$.
  (5) All parties recover $\llbracket x' \rrbracket_{k+s}, \llbracket y' \rrbracket_{k+s}, \llbracket z' \rrbracket_{k+s}$ and check if $z' \equiv_{k+s} x' \cdot y'$. If not, all parties abort, otherwise output accept.

are much more massive. Finally, we remark that we also measure communication experimentally in Section 6.

5 Soundness Analysis of Π_{VrfyIP} and Optimizations

Now we give a tight soundness analysis of Π_{VrfyIP} and optimizations for general scenarios. We give concrete optimizations that only work for 3-party computation in Appendix D.1 and discuss other optimizations of postponing pair-wise consistency check, using PRG, and applying Fiat-Shamir transformation in Appendix D.2.

5.1 Soundness Analysis of Π_{VrfyIP}

Recall that in Π_{VrfyIP}, when the prover party P_j is corrupted and at least one of the input inner-product triples is incorrect, with overwhelming probability, the number of 2-factors in the additive error of the inner-product triple obtained in Step 2 is bounded by $k + \lambda$. And in each iteration of Step 3 or Step 4, the number of 2-factors in the additive error of the triple to be checked grows by 2λ. Then to ensure that the final additive error ϵ is not 0 when modulo 2^{k+s}, we have to set $k + s \geq k + (2\log_q(p \cdot d) + 1)\lambda$, indicating that $s \geq (2\log_q(p \cdot d) + 1)\lambda$. This is too large to be practical.

We note that our above analysis is very pessimistic: Each time we take a random linear combination, we assume that the number of 2-factors of the additive error always grows by λ (Note that each dimension reduction step consists of doing two times of random linear combinations). To get a better bound on s, we establish a connection between our protocol and the following game.

$\mathcal{G}ame(k, s, T)$. Consider an interactive game between an adversary $\mathcal{A}_g$ and a challenger $\mathcal{C}_g$. Recall that for all $x \in \mathbb{Z}_{2^{k+s}}$ and $x \neq 0$, we define $\mathsf{Po2}(x)$ to be the number of 2-factors in x, *i.e.*, the largest integer u s.t. 2^u divides x, and $\mathsf{Po2}(x) := k + s$ if $x = 0$. Given the number of interactive rounds T, the game works as follows.

(1) $\mathcal{A}_g, \mathcal{C}_g$ initially have $E_0 = k - 1$.
(2) In each round i $(1 \leq i \leq T)$, $\mathcal{A}_g$ and $\mathcal{C}_g$ repeat the following:
  (a) $\mathcal{A}_g$ chooses arbitrary $e_i, c_i \in \mathbb{Z}_{2^{k+s}}$ under the requirement that $\mathsf{Po2}(e_i) \leq E_{i-1}$, and sends the two values to $\mathcal{C}_g$.
  (b) $\mathcal{C}_g$ picks a uniformly random value $r_i \in \mathbb{Z}_{2^{k+s}}$ and responds r_i to $\mathcal{A}_g$.
  (c) $\mathcal{A}_g$ and $\mathcal{C}_g$ compute $E_i = \mathsf{Po2}(r_i \cdot e_i + c_i)$.
(3) $\mathcal{A}_g$ wins if and only if in the last round T, $E_T = k + s$.

We show that an adversary $\mathcal{A}$ of our protocol who has advantage p (of forcing honest parties accepting incorrect inner-product triples) can be translated to some adversary $\mathcal{A}_g$ with the same advantage p (of winning the above game) with $T = 2\log_q(p \cdot d) + 1$. We give the explanation of the connection between our protocol and the above game in Appendix G.1.

Main Lemma of $\mathcal{G}ame(k, s, T)$. The value of considering the game above is that, as it turns out, we are able to bound its probability of success much more tightly. Towards this end, we obtain the following lemma, which is proven in Appendix G.2. We note that Lemma 5.1 gives the tight upper bound on the winning probability of $\mathcal{A}_g$. See Appendix G.2 for $\mathcal{A}_g$ that matches this bound.

Lemma 5.1. *Let k, s, T be positive integers. For any adversary $\mathcal{A}_g$, the probability that $\mathcal{A}_g$ wins $\mathcal{G}ame(k, s, T)$ is at most $\sum_{j=0}^{T-1} \binom{s+j}{s} \cdot \frac{1}{2^{s+1+j}}$.*

In Appendix G.6.1, we show that when $s = \lambda + T(1/2 + \log(5/2 + 3\lambda/T))$ (assuming that $T \le \lambda$ and $3T \le s$), the winning probability of $\mathcal{A}_g$ is at most $2^{-\lambda}$. Thus, we have the following lemma. The proof can be found in Appendix G.3.

Lemma 5.2. *Let p, d, q be positive integers and $T = 2\lceil \log_q(p \cdot d) \rceil + 1$. Assume that λ is the security parameter and $T \le \lambda$. When $s = \max(3T, \lambda + T(1/2 + \log(5/2 + 3\lambda/T)))$, the protocol Π_{VrfyIP} securely computes $\mathcal{F}_{VrfyIP}$ with abort in the $\{\mathcal{F}_{input}, \mathcal{F}_{coin}\}$-hybrid model against a malicious adversary controlling $t = \frac{n-1}{2}$ corrupted parties and achieves soundness error $2^{-\lambda}$.*

5.2 General Optimizations

Removing the Multiplicative Overhead of p in Π_{VrfyIP}. Recall that the input of Π_{VrfyIP} consists of p inner-product triples $\{(\llbracket \boldsymbol{\mu}_i \rrbracket_k, \llbracket \boldsymbol{\nu}_i \rrbracket_k, \llbracket w_i \rrbracket_k)\}_{i=1}^{p}$ and each inner-product triple has dimension d. Then in Step 2 of Π_{VrfyIP}, the merged inner-product triple $(\llbracket \boldsymbol{x} \rrbracket_{k+s}, \llbracket \boldsymbol{y} \rrbracket_{k+s}, \llbracket z \rrbracket_{k+s})$ has dimension $p \cdot d$. We show that the multiplicative overhead of p can be removed when considering how $\{(\llbracket \boldsymbol{\mu}_i \rrbracket_k, \llbracket \boldsymbol{\nu}_i \rrbracket_k, \llbracket w_i \rrbracket_k)\}_{i=1}^{p}$ are obtained in Π_{VrfySSIP}.

Recall that Π_{VrfyIP} is invoked in Π_{VrfySSIP} to verify the correctness of each prover party P_j. In particular,

- In Π_{VrfySSIP}, all parties transform $\{(\llbracket \boldsymbol{a}_i \rrbracket_k, \llbracket \boldsymbol{b}_i \rrbracket_k, \llbracket c_i \rrbracket_k)\}_{i=1}^{m}$ into λ inner-product triples $\{(\llbracket \boldsymbol{a}'_i \rrbracket_k, \llbracket \boldsymbol{b}'_i \rrbracket_k, \llbracket c'_i \rrbracket_k)\}_{i=1}^{\lambda}$ by random subset sum.
- For each $(\llbracket \boldsymbol{a}'_i \rrbracket_k, \llbracket \boldsymbol{b}'_i \rrbracket_k, \llbracket c'_i \rrbracket_k)$, all parties locally compute an additive sharing of $\langle c'_i \rangle_k$ and each party P_j shares his additive share ${c'}_i^{(j)}$ using the replicated secret sharing scheme.
- Each prover P_j needs to prove he correctly computes ${c'}_i^{(j)}$. As shown in Appendix A, this is transformed to verifying λ inner-product triples where all the replicated secret sharings are known to P_j. The verification task is handled by Π_{VrfyIP}.

Thus, we have $p = \lambda$ and $(\llbracket \boldsymbol{\mu}_i \rrbracket_k, \llbracket \boldsymbol{\nu}_i \rrbracket_k, \llbracket w_i \rrbracket_k)$ is obtained from $(\llbracket \boldsymbol{a}'_i \rrbracket_k, \llbracket \boldsymbol{b}'_i \rrbracket_k, \llbracket c'_i \rrbracket_k)$.

Note that in Step 2 of Π_{VrfyIP}, we compute a random linear combination of $\{(\llbracket \boldsymbol{\mu}_i \rrbracket_k, \llbracket \boldsymbol{\nu}_i \rrbracket_k, \llbracket w_i \rrbracket_k)\}_{i=1}^{p}$ after lifting to $\mathbb{Z}_{2^{k+s}}$. Since

- $\boldsymbol{\mu}_i \cdot \boldsymbol{\nu}_i$ models the procedure of P_j computing his additive share of ${c'}_i^{(j)}$ for triple $(\llbracket \boldsymbol{a}'_i \rrbracket_k, \llbracket \boldsymbol{b}'_i \rrbracket_k, \llbracket c'_i \rrbracket_k)$,
- and $(\llbracket \boldsymbol{a}'_i \rrbracket_k, \llbracket \boldsymbol{b}'_i \rrbracket_k, \llbracket c'_i \rrbracket_k)$ is a subset sum of $\{(\llbracket \boldsymbol{a}_i \rrbracket_k, \llbracket \boldsymbol{b}_i \rrbracket_k, \llbracket c_i \rrbracket_k)\}_{i=1}^{m}$,

$\boldsymbol{\mu}_i \cdot \boldsymbol{\nu}_i$ corresponds to computing P_j's additive share of some subset sum of $\{\boldsymbol{a}_i \cdot \boldsymbol{b}_i\}_{i=1}^{m}$. Thus, the random linear combination of $\{\boldsymbol{\mu}_i \cdot \boldsymbol{\nu}_i\}_{i=1}^{\lambda}$ also corresponds to computing P_j's additive share of some random linear combination of $\{\boldsymbol{a}_i \cdot \boldsymbol{b}_i\}_{i=1}^{m}$. Thus, we may combine the like terms in Step 2 of Π_{VrfyIP}. As a result, the dimension of the inner-product triple $(\llbracket \boldsymbol{x} \rrbracket_{k+s}, \llbracket \boldsymbol{y} \rrbracket_{k+s}, \llbracket z \rrbracket_{k+s})$ is reduced from $\lambda \cdot d$ to d, where $d = \delta \cdot \binom{n-1}{t}^2$ and δ is the number of products (sum of the dimensions) of $\{(\llbracket \boldsymbol{a}_i \rrbracket_k, \llbracket \boldsymbol{b}_i \rrbracket_k, \llbracket c_i \rrbracket_k)\}_{i=1}^{m}$.

Further Boosting the Soundness of Π_{VrfyIP}. Although in Section 5.1, to achieve λ-bit security, we have reduced the requirement on s from $s = T \cdot \lambda$ to $s = \lambda + T(1/2 + \log(5/2 + 3\lambda/T))$, where $T = 2\log_q d + 1$ (considering the first optimization), this may still be too large to use in practice.

A natural idea is to repeat Π_{VrfyIP} by ℓ times so that we may choose to use a smaller $s = \lambda/\ell + T(1/2 + \log(5/2 + 3\lambda/(\ell \cdot T)))$. However, this also means that the computation complexity grows by a factor of ℓ due to the repetition.

We note that the most computationally expensive steps in Π_{VrfyIP} are Step 1, Step 2, and Step 3.1 in the first iteration:

- In Step 1, the prover P_j computes h_i for each triple. This step has computation complexity $O(p \cdot d)$.
- In Step 2, all parties compute $(\llbracket \boldsymbol{x} \rrbracket_{k+s}, \llbracket \boldsymbol{y} \rrbracket_{k+s}, \llbracket z \rrbracket_{k+s})$ from p inner-product triple of dimension d. This step has computation complexity $O(p \cdot d)$.
- In Step 3.1, the prover computes $z_{i,i'} = \boldsymbol{x}_i \cdot \boldsymbol{y}_{i'}$ for all $i, i' \in [q]$. Since $\boldsymbol{x}_i, \boldsymbol{y}_{i'}$ have dimension d/q. This step has computation complexity $O(q^2 \cdot d/q) = O(q \cdot d)$.
- The rest of steps have computation complexity $O(d)$.

Thus, our idea is to keep the computationally expensive steps running once while repeating the rest of the steps to boost the soundness. Concretely, we will repeat Step 3 and Step 4 in Π_{VrfyIP} by two times. (Note that in the first iteration of Step 3, we only repeat the process of generating random challenges. So Step 3.1 in the first iteration only runs once).

We denote the above optimized protocol by $\Pi_{\mathrm{VrfyIP}}^{\mathrm{Opt}}$. In Appendix G.6.2, we give the soundness analysis of $\Pi_{\mathrm{VrfyIP}}^{\mathrm{Opt}}$, and also an estimation of s to achieve λ-bit security. When focusing on the concrete case where $\lambda = 40$, it suffices to use $s = 64$ when $T \le 21$ (obtained by directly computing the probability in Lemma G.3 rather than using the estimation in G.6.2). Note that $T = 21$ means that $\log_q d$ can be as large as 10. When $q = 8$, this allows us to check inner-product triples of dimension $d \le 8^{10} = 2^{30}$ in Π_{VrfyIP}.

Reusing Randomness in Different Batches. When the m inner-product triples we need to verify in Π_{VrfySSIP} all have the same dimension, we may divide these m inner-product triples into batches of size B. Then we check each batch of inner-product triples by using Π_{VrfySSIP} and run all invocations of Π_{VrfySSIP} in parallel. In particular, we can use the same set of random values generated from $\mathcal{F}_{\mathrm{coin}}$ in all batches. Note that this will not make the soundness error worse because we may only focus on the batch which is incorrect. So the soundness error remains the same as that when we check all inner-product triples using one call of Π_{VrfySSIP}.

This optimization brings us two benefits. First, we can save the number of calls to $\mathcal{F}_{\mathrm{coin}}$. Second, in the second step of Π_{VrfyIP} where all parties compute $(\llbracket \boldsymbol{x} \rrbracket_{k+s}, \llbracket \boldsymbol{y} \rrbracket_{k+s}, \llbracket z \rrbracket_{k+s})$, we note that the computation essentially computes coefficients that only depend on the random values generated in $\mathcal{F}_{\mathrm{coin}}$. By using the same set of random values, we only need to compute these coefficients once.

On the other hand, this optimization increases the communication complexity of the verification stage by a factor of m/B. However, the overall communication complexity remains to be sublinear in the input size when B is large enough.

6 Experimental Results

We fully implement our verification protocol for the particular case of three-party computation, which is the setting considered in [11], and in the protocols from [14, 38, 42] that use [11] as a building block. We write our code as part of the MP-SPDZ framework [36], adding our verification protocol to the passive protocol from [6], which is available in MP-SPDZ. For the purpose of comparison we also implement the verification protocol from [11], and we use the NTL numerical library for the implementation of the Galois ring extensions needed in this protocol. Although the verification protocol from [11] with Galois ring extensions has been used in many previous works [14, 33, 37, 38, 42], the only open implementation we are aware is the one from [33], which is for the four-party setting. For completeness, in Appendix H, we test their implementation and verify experimentally that the running time of our Galois ring implementation of [11] is consistent with theirs.

For our comparisons we do not simply run our verification protocol against the one from [11], but we do this in the context of compiling the passively secure three-party protocol from [6] to active security using these verification checks. The motivation for this is two-fold. First, these checks are typically not used in a standalone manner, but instead they are used at the end of an MPC protocol to check its correctness; only comparing our verification against the one from [11] says little about how much such improvements translate to the actual MPC context, since in that setting the checks are only a component of a wider protocol. Second, the verification protocol in [11] is tied to the specific 3PC protocol in [6], unlike ours which is protocol-independent and only requires secret-shared inner products. Note that this plays against us: their verification exploits properties of the 3PC protocol that is being verified, while ours is entirely black-box.

In what follows we experimentally compare three different three-party protocols: (1) the passive protocol from [6], compiled to active security with our verification protocol, (2) the same passive protocol but compiled with [11], which uses ring extensions, and (3) the plain passive protocol from [6] without any verification steps, which is done in order to better understand the overhead of obtaining active security with the different protocols above.

Experimental Setup. We deploy our three-party protocol on three Alibaba Cloud g7.8xlarge instances running Ubuntu 20.04, each equipped with 32-core Intel(R) Xeon(R) Platinum 8369B CPU @2.70 GHz and 128GB of RAM. [11] The machines are in a LAN with about 23Gbps bandwidth and $30\mu s$ (one-way) latency. As for WAN setting, we use the Linux tc command to set the bandwidth at 80Mbps and latency at 40ms, which simulates actual network conditions between two distant machines.

In the experiments we study the performance of the protocols above for two classes of circuits:

- We fix the depth to be 10 and vary the total number of multiplications, namely 10K, 100K, 1M and 10M multiplications. In these experiments, since the circuits have low depth, the final distributed product check plays an important role in the resulting end-to-end runtimes, so these results allow us to understand how well our verification scales with respect to the other protocols as the number of multiplications increases. Results are reported in Tables 1 for LAN and WAN.
- We fix the total number of multiplications to 1M, and vary the depth as 1, 10, 100 and 1,000. As the depth increases the effect of the final distributed product check on the end-to-end runtimes is less and less noticeable, so these results reflect up to what extent our protocol — which only improves this step — impacts total performance. Results are reported in Tables 2 for LAN and WAN.

For the ring we use $k = 64$, that is, we check triples modulo 2^{64}, and we use $\lambda = 40$ bits of statistical security. We set $s = 64$ (so our proof operates over $\mathbb{Z}_{2^{k+s}} = \mathbb{Z}_{2^{128}}$), and the extension degree to be 47 for the extension ring-based approach [11] to achieve 40-bit security. Besides, both our protocol and the extension ring-based protocol [11] set the compression parameter $q = 8$, and the batch size parameter (see Section 5.2) $B = 10,000$ for circuit size $\leq 1M$, and $B = 100,000$ for circuit size $10M$, since they are experimentally shown to be the optimal choices for the two protocols.

We run each program with *a single thread* on a *single CPU core*. The results we report are the average of ten runs. Additionally, to see the performance of the programs running with multiple threads, we test them using 10 threads with 10 CPU cores in the verification phase, and present the results in Appendix I.

Experiments in the LAN setting. We first run the protocols in the LAN setting, where computation impacts more end-to-end runtimes than communication. In Table 1 we see the results of the different protocols for depth-10 circuits of varying sizes 10K, 100K, 1M, 10M, and in Table 2 we fix the number of multiplications to 1M, and vary depth 1, 10, 100 and 1,000. We make several interesting observations about these results. First, we see that, when compared to [11], even though communication in [11] is generally only slightly larger than ours, our protocol can be up to one order of magnitude better in terms of runtimes. For example, for one million multiplication gates and depth 10 and 100, our protocol is nearly 43.4× and 44.2× better than the one by [11]. This is no surprise: when the depth is low, a big portion of the end-to-end runtimes is actually dictated by the verification step, which uses large-degree ring extensions in [11], while we avoid them entirely in our work. As the depth increases to 1,000, the impact of the distributed product check in the end-to-end runtime is less noticeable but even there using our verification protocol leads to 36.1× improvements with respect to using the check from [11]. In particular, ours is the first concretely practical work that enables active security at essentially the same communication costs as semihonest, over $\mathbb{Z}_{2^k}$, while achieving concrete practical efficiency. Indeed, we see that when we compare with the plain *passive* protocol from [6], our protocol is not considerably more expensive in terms of runtimes: for depth ten thousand gates we are only 5.6 times more expensive, and for ten million gates this factor is only 7.7; for larger

[11] All these Alibaba servers, like Amazon servers, can be rented by anyone: https://us.alibabacloud.com/en, making our experiments easily reproducible for future research.

Table 1: Comparison between the three-party passive protocol from [6] when compiled with our approach, with BGIN19 [11], or without any compilation. We consider circuits of depth 10 with varying sizes.

Protocol	# Mults	1 Thread		10 Threads		Comm. (MB)
		LAN Time (s)	WAN Time (s)	LAN Time (s)	WAN Time (s)	
Ours	10K	0.0062 (×**1**)	1.13 (×**1**)	0.0064 (×**1**)	1.06 (×**1**)	0.28 (×**1**)
	100K	0.029 (×**1**)	1.39 (×**1**)	0.014 (×**1**)	1.33 (×**1**)	2.75 (×**1**)
	1M	0.26 (×**1**)	3.17 (×**1**)	0.093 (×**1**)	2.89 (×**1**)	27.44 (×**1**)
	10M	2.82 (×**1**)	13.98 (×**1**)	0.95 (×**1**)	12.12 (×**1**)	244.05 (×**1**)
BGIN19	10K	0.11 (×**17.7**)	0.88 (×**0.8**)	0.11 (×**17.2**)	0.79 (×**0.7**)	0.33 (×**1.18**)
	100K	1.09 (×**37.6**)	2.14 (×**1.5**)	0.45 (×**32.1**)	1.4 (×**1.1**)	3.34 (×**1.21**)
	1M	11.29 (×**43.4**)	14.11 (×**4.5**)	4.35 (×**46.8**)	6.98 (×**2.4**)	33.42 (×**1.22**)
	10M	121.16 (×**43.0**)	135.67 (×**9.7**)	42.26 (×**44.5**)	54.03 (×**4.5**)	251.43 (×**1.03**)
Passive	10K	0.0011 (×**0.18**)	0.45 (×**0.40**)	-	-	0.24 (×**0.85**)
	100K	0.0033 (×**0.11**)	0.67 (×**0.48**)			2.4 (×**0.87**)
	1M	0.031 (×**0.12**)	2.16 (×**0.68**)			24 (×**0.87**)
	10M	0.36 (×**0.13**)	11.87 (×**0.85**)			240 (×**0.98**)

Table 2: Comparison between the three-party passive protocol from [6] when compiled with our approach, with BGIN19 [11], or without any compilation. We consider circuits of size 1M with varying depths.

Protocol	Depth	1 Thread		10 Threads		Comm. (MB)
		LAN Time (s)	WAN Time (s)	LAN Time (s)	WAN Time (s)	
Ours	1	0.39 (×**1**)	2.82 (×**1**)	0.22 (×**1**)	2.59 (×**1**)	27.44 (×**1**)
	10	0.26 (×**1**)	3.17 (×**1**)	0.093 (×**1**)	2.89 (×**1**)	
	100	0.26 (×**1**)	6.08 (×**1**)	0.095 (×**1**)	5.87 (×**1**)	
	1,000	0.32 (×**1**)	42.13 (×**1**)	0.15 (×**1**)	41.91 (×**1**)	
BGIN19	1	9.49 (×**24.3**)	12.85 (×**4.6**)	4.14 (×**18.8**)	6.19 (×**2.4**)	33.42 (×**1.22**)
	10	11.29 (×**43.4**)	14.11 (×**4.5**)	4.35 (×**46.8**)	6.98 (×**2.4**)	
	100	11.49 (×**44.2**)	17.08 (×**2.8**)	4.40 (×**46.3**)	10.00 (×**1.7**)	
	1,000	11.54 (×**36.1**)	53.41 (×**1.3**)	4.39 (×**29.3**)	46.08 (×**1.1**)	
Passive	1	0.16 (×**0.41**)	1.81 (×**0.64**)	-	-	24.00 (×**0.87**)
	10	0.031 (×**0.12**)	2.16 (×**0.68**)			
	100	0.03 (×**0.12**)	5.05 (×**0.83**)			
	1,000	0.08 (×**0.25**)	40.95 (×**0.97**)			

depth this gap shrinks even more. Furthermore, in terms of communication, we are much closer, and for ten million gates our actively secure protocol only incurs an extra overhead of $2\% = (\frac{1}{0.98} - 1)\%$ with respect to the passive protocol.

Experiments in the WAN setting. In the WAN case, there is more time available to perform expensive computations, and hence the overheads of using Galois rings may be less harmful. As shown in Table 1 and Table 2, indeed, our improvement factor over [11] is not as large as that in the LAN case, but it is still considerable: for ten million gates and depth 10 we can get around 9.7× improvement in runtimes. As the depth increases, the improvement factor on the end-to-end runtime goes down, which is due to the fact that for large depths the effect of the final verification step on the total runtime is less noticeable. As the network becomes slower, so does the passively secure protocol of [6], which means that the overhead of compiling to active security using our sublinear distributed product checks is less appreciable, which is particularly true as the depth grows since our verification check is constant-round. We see this reflected in our experimental results: for circuits with 10M gates and depth 10 our protocol only adds $18\% = (\frac{1}{0.85} - 1)\%$ overhead to the passive runtimes, and for 1M gates and depth 1,000 this is only an extra $3\% = (\frac{1}{0.97} - 1)\%$. Thus, thanks to our work, we can truly claim that, in several practical settings, active security comes at the *same concrete cost* as semi-honest.

Acknowledgments

Y. Song was supported in part by the National Basic Research Program of China Grant 2011CBA00300, 2011CBA00301, the National Natural Science Foundation of China Grant 61033001, 61361136003.

This paper was prepared in part for information purposes by the Artificial Intelligence Research group of JPMorgan Chase & Co and its affiliates ("JP Morgan"), and is not a product of the Research Department of JP Morgan. JP Morgan makes no representation and warranty whatsoever and disclaims all liability, for the completeness, accuracy or reliability of the information contained herein. This document is not intended as investment research or investment advice, or a recommendation, offer or solicitation for the purchase or sale of any security, financial instrument, financial product or service, or to be used in any way for evaluating the

merits of participating in any transaction, and shall not constitute a solicitation under any jurisdiction or to any person, if such solicitation under such jurisdiction or to such person would be unlawful. 2024 JP Morgan Chase & Co. All rights reserved.

References

[1] Ittai Abraham, Gilad Asharov, Shravani Patil, and Arpita Patra. 2023. Detect, Pack and Batch: Perfectly-Secure MPC with Linear Communication and Constant Expected Time. In *Advances in Cryptology – EUROCRYPT 2023*, Carmit Hazay and Martijn Stam (Eds.). Springer Nature Switzerland, Cham, 251–281.

[2] Ittai Abraham, Gilad Asharov, and Avishay Yanai. 2022. Efficient Perfectly Secure Computation with Optimal Resilience. *Journal of Cryptology* 35, 4 (2022), 27. https://doi.org/10.1007/s00145-022-09434-2

[3] Mark Abspoel, Ronald Cramer, Ivan Damgård, Daniel Escudero, and Chen Yuan. 2019. Efficient information-theoretic secure multiparty computation over Z2k via Galois rings. In *Theory of Cryptography Conference*. Springer, 471–501.

[4] Mark Abspoel, Anders Dalskov, Daniel Escudero, and Ariel Nof. 2021. An efficient passive-to-active compiler for honest-majority MPC over rings. In *International Conference on Applied Cryptography and Network Security*. Springer, 122–152.

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

[6] 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 *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM, 805–817. https://doi.org/10.1145/2976749.2978331

[7] Gilad Asharov, Yehuda Lindell, and Tal Rabin. 2011. Perfectly-Secure Multiplication for Any t < n/3. In *Advances in Cryptology – CRYPTO 2011*, Phillip Rogaway (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 240–258.

[8] Alessandro N. Baccarini, Marina Blanton, and Chen Yuan. 2020. Multi-Party Replicated Secret Sharing over a Ring with Applications to Privacy-Preserving Machine Learning. *IACR Cryptol. ePrint Arch.* (2020), 1577. https://eprint.iacr.org/2020/1577

[9] Zuzana Beerliová-Trubíniová and Martin Hirt. 2008. Perfectly-Secure MPC with Linear Communication Complexity. In *Theory of Cryptography*, Ran Canetti (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 213–230.

[10] 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 *Annual International Cryptology Conference*. Springer, 67–97.

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

[12] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. 2020. Efficient fully secure computation via distributed zero-knowledge proofs. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 244–276.

[13] Dario Catalano, Mario Di Raimondo, Dario Fiore, and Irene Giacomelli. 2020. MonZ2ka: Fast Maliciously Secure Two Party Computation on Z2k. In *Public-Key Cryptography - PKC 2020 - 23rd IACR International Conference on Practice and Theory of Public-Key Cryptography, Edinburgh, UK, May 4-7, 2020, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12111)*, Aggelos Kiayias, Markulf Kohlweiss, Petros Wallden, and Vassilis Zikas (Eds.). Springer, 357–386. https://doi.org/10.1007/978-3-030-45388-6_13

[14] Harsh Chaudhari, Ashish Choudhury, Arpita Patra, and Ajith Suresh. 2019. ASTRA: high throughput 3pc over rings with application to secure prediction. In *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*. 81–92.

[15] Harsh Chaudhari, Rahul Rachuri, and Ajith Suresh. 2020. Trident: Efficient 4PC Framework for Privacy Preserving Machine Learning. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/trident-efficient-4pc-framework-for-privacy-preserving-machine-learning/

[16] Jung Hee Cheon, Dongwoo Kim, and Keewoo Lee. 2021. MHz2k: MPC from HE over Z/2kZ with New Packing, Simpler Reshare, and Better ZKP. In *Advances in Cryptology - CRYPTO 2021 - 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16-20, 2021, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12826)*, Tal Malkin and Chris Peikert (Eds.). Springer, 426–456. https://doi.org/10.1007/978-3-030-84245-1_15

[17] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. 2018. Fast large-scale honest-majority MPC for malicious adversaries. In *Advances in Cryptology–CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part III 38*. Springer, 34–64.

[18] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. 2018. Fast Large-Scale Honest-Majority MPC for Malicious Adversaries. In *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part III (Lecture Notes in Computer Science, Vol. 10993)*, Hovav Shacham and Alexandra Boldyreva (Eds.). Springer, 34–64. https://doi.org/10.1007/978-3-319-96878-0_2

[19] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. 2018. SPD$\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for Dishonest Majority. In *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10992)*, Hovav Shacham and Alexandra Boldyreva (Eds.). Springer, 769–798. https://doi.org/10.1007/978-3-319-96881-0_26

[20] Anders Dalskov, Daniel Escudero, and Ariel Nof. 2022. Fast Fully Secure Multi-Party Computation over Any Ring with Two-Thirds Honest Majority. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 653–666.

[21] Anders P. K. Dalskov and Daniel Escudero. 2021. Honest Majority MPC with Abort with Minimal Online Communication. In *Progress in Cryptology - LATINCRYPT 2021 - 7th International Conference on Cryptology and Information Security in Latin America, Bogotá, Colombia, October 6-8, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12912)*, Patrick Longa and Carla Ràfols (Eds.). Springer, 453–472. https://doi.org/10.1007/978-3-030-88238-9_22

[22] Anders P. K. Dalskov, Daniel Escudero, and Marcel Keller. 2021. Fantastic Four: Honest-Majority Four-Party Secure Computation With Malicious Security. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 2183–2200. https://www.usenix.org/conference/usenixsecurity21/presentation/dalskov

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

[24] Hendrik Eerikson, Marcel Keller, Claudio Orlandi, Pille Pullonen, Joonas Puura, and Mark Simkin. 2020. Use Your Brain! Arithmetic 3PC for Any Modulus with Active Security. In *1st Conference on Information-Theoretic Cryptography (ITC 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 163)*, Yael Tauman Kalai, Adam D. Smith, and Daniel Wichs (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 5:1–5:24. https://doi.org/10.4230/LIPIcs.ITC.2020.5

[25] Daniel Escudero and Eduardo Soria-Vazquez. 2021. Efficient information-theoretic multi-party computation over non-commutative rings. In *Advances in Cryptology–CRYPTO 2021: 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16–20, 2021, Proceedings, Part II 41*. Springer, 335–364.

[26] Daniel Escudero, Chaoping Xing, and Chen Yuan. 2022. More Efficient Dishonest Majority Secure Computation over Z2k via Galois Rings. In *Annual International Cryptology Conference*. Springer, 383–412.

[27] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. 2017. High-throughput secure three-party computation for malicious adversaries and an honest majority. In *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 225–255.

[28] Daniel Genkin, Yuval Ishai, Manoj Prabhakaran, Amit Sahai, and Eran Tromer. 2014. Circuits resilient to additive attacks with applications to secure computation. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, David B. Shmoys (Ed.). ACM, 495–504. https://doi.org/10.1145/2591796.2591861

[29] Shafi Goldwasser and Yehuda Lindell. 2005. Secure Multi-Party Computation without Agreement. *Journal of Cryptology* 18, 3 (2005).

[30] Vipul Goyal, Hanjun Li, Rafail Ostrovsky, Antigoni Polychroniadou, and Yifan Song. 2021. ATLAS: Efficient and Scalable MPC in the Honest Majority Setting. In *Advances in Cryptology – CRYPTO 2021*. Springer International Publishing, Cham, 244–274.

[31] Vipul Goyal, Yanyi Liu, and Yifan Song. 2019. Communication-Efficient Unconditional MPC with Guaranteed Output Delivery. In *Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 11693)*, Alexandra Boldyreva and Daniele Micciancio (Eds.). Springer, 85–114. https://doi.org/10.1007/978-3-030-26951-7_4

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

[33] Aditya Hegde, Nishat Koti, Varsha Bhat Kukkala, Shravani Patil, Arpita Patra, and Protik Paul. 2023. Attaining GOD Beyond Honest Majority with Friends and Foes. In *Advances in Cryptology–ASIACRYPT 2022: 28th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, December 5–9, 2022, Proceedings, Part I*. Springer, 556–587.

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

[35] Daniel Kales and Greg Zaverucha. 2020. An attack on some signature schemes constructed from five-pass identification schemes. In *Cryptology and Network Security: 19th International Conference, CANS 2020, Vienna, Austria, December 14–16, 2020, Proceedings*. Springer, 3–22.

[36] Marcel Keller. 2020. MP-SPDZ: A Versatile Framework for Multi-Party Computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 1575–1590. https://doi.org/10.1145/3372297.3417872

[37] Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, and Bhavish Raj Gopal. 2022. PentaGOD: Stepping beyond Traditional GOD with Five Parties. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 1843–1856.

[38] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. 2021. {SWIFT}: Super-fast and Robust {Privacy-Preserving} Machine Learning. In *30th USENIX Security Symposium (USENIX Security 21)*. 2651–2668.

[39] Nishat Koti, Arpita Patra, Rahul Rachuri, and Ajith Suresh. 2022. Tetrad: Actively Secure 4PC for Secure Training and Inference. In *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/auto-draft-202/

[40] Payman Mohassel and Peter Rindal. 2018. ABY^3: A Mixed Protocol Framework for Machine Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 35–52. https://doi.org/10.1145/3243734.3243760

[41] Emmanuela Orsini, Nigel P. Smart, and Frederik Vercauteren. 2020. Overdrive2k: Efficient Secure MPC over Z2k from Somewhat Homomorphic Encryption. In *Topics in Cryptology - CT-RSA 2020 - The Cryptographers' Track at the RSA Conference 2020, San Francisco, CA, USA, February 24-28, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12006)*, Stanislaw Jarecki (Ed.). Springer, 254–283. https://doi.org/10.1007/978-3-030-40186-3_12

[42] Arpita Patra and Ajith Suresh. 2020. BLAZE: Blazing Fast Privacy-Preserving Machine Learning. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/blaze-blazing-fast-privacy-preserving-machine-learning/

[43] Sameer Wagh. 2022. Pika: Secure computation using function secret sharing over rings. *Proceedings on Privacy Enhancing Technologies* (2022).

[44] Sameer Wagh, Divya Gupta, and Nishanth Chandran. 2019. SecureNN: 3-Party Secure Computation for Neural Network Training. *Proceedings on Privacy Enhancing Technologies* 3 (2019), 26–49.

[45] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. 2021. Falcon: Honest-Majority Maliciously Secure Framework for Private Deep Learning. *Proceedings on Privacy Enhancing Technologies* 1 (2021), 188–208.

[46] Andrew C Yao. 1982. Protocols for secure computations. In *23rd annual symposium on foundations of computer science (sfcs 1982)*. IEEE, 160–164.

A Properties of RSS

Pairwise Consistency. In the setting of $n = 2t + 1$, each additive share is held by a subset of $t + 1$ parties. For a replicated secret sharing $[\![x]\!]$, we say $[\![x]\!]$ is pairwise consistent if for every set T of size $t+1$, all honest parties in T hold the same value x_T. Note that a malicious dealer may distribute an inconsistent $[\![x]\!]$ to all parties. As noted in [12], to check whether some sharings $[\![x]\!]$ are pairwise consistent, every party P_i sends to every other party P_j with $j > i$ a hash of the concatenated shares $(x_T)_{P_i,P_j \in T}$, which P_j uses to compare against his local shares. This involves a communication of $n(n-1)/2$ digests in total. Furthermore, multiple secrets can be checked with the same communication by concatenating the respective shares in the hashes.

Linear Operations. The RSS scheme is linearly homomorphic, which means that, from sharings $[\![x]\!]$ and $[\![y]\!]$, parties can *locally* compute sharings $[\![x+y]\!]$, and also $[\![x \pm c]\!]$ and $[\![c \cdot x]\!]$ for any publicly known value $c \in \mathbb{Z}_{2^k}$. Finally, shares $[\![c]\!]$ can be locally generated as long as the value c is known by one set T_0 of size $t+1$ by setting $c_T := c$ for $T = T_0$, and $c_T = 0$ for the other sets of size $t+1$. In particular, addition by a constant only requires this constant to be known by $t+1$ parties.

Local Multiplication. As observed in [8], given a pair of replicated secret sharings $[\![x]\!]$, $[\![y]\!]$, all parties can locally compute an additive sharing of the multiplication result $\langle z \rangle = \langle x \cdot y \rangle$. To be more concrete, recall that a replicated secret sharing $[\![x]\!]$ is defined by $x = \sum_{T \subset \mathcal{P}, |T|=t+1} x_T$ where each share x_T is held by parties in T. Then we have $x \cdot y = \left(\sum_{T \subset \mathcal{P}, |T|=t+1} x_T\right) \cdot \left(\sum_{T \subset \mathcal{P}, |T|=t+1} y_T\right) = \sum_{T_1,T_2 \subset \mathcal{P}, |T_1|=|T_2|=t+1} x_{T_1} \cdot y_{T_2}$. Note that for all $T_1, T_2 \subset \mathcal{P}$ and $|T_1| = |T_2| = t+1$, $T_1 \cap T_2 \neq \emptyset$, implying that at least one party holds both the shares x_{T_1} and y_{T_2}. Thus, for all $T_1, T_2 \subset \mathcal{P}$ and $|T_1| = |T_2| = t+1$, we assign the party $P_i \in T_1 \cap T_2$ with the smallest index to locally compute $x_{T_1} \cdot y_{T_2}$. Then each party P_i locally add up all the terms $x_{T_1} \cdot y_{T_2}$ he have computed and view the summation as his additive share $\langle z \rangle$. In this way, all parties together hold an additive sharing $\langle z \rangle$. We denote the above process by $\langle z \rangle = [\![x]\!] \cdot [\![y]\!]$. Note that each party computes an inner-product over their original shares of $[\![x]\!]$, $[\![y]\!]$. In particular, the dimension of the inner-product is bounded by $\binom{n-1}{t}^2$.

Local Conversion. Given a sharing $[\![x]\!]$, the parties can *locally* obtain sharings $[\![x_S]\!]$ for every $S \subseteq \mathcal{P}$ with $|S| = t+1$. To do this, for every set $T \subseteq \mathcal{P}$ with $|T| = t+1$, we define the corresponding additive share of x_S as $x_{S,T} := x_S$ if $T = S$, and $x_{S,T} := 0$ otherwise. The parties in the set $T = S$ can define their share $x_{S,T} = x_S$ since, by definition of x_S, they know this value.

Modulo Reduction. We describe a modulo reduction operation on RSS. For a secret sharing $[\![x]\!]_{k+s}$ where $x \in \mathbb{Z}_{2^{k+s}}$ where s is a positive integer, the parties can reduce the sharing $[\![x]\!]_{k+s}$ from modulo 2^{k+s} to 2^k simply by taking $[\![x \bmod 2^k]\!]_k = [\![x]\!]_{k+s} \bmod 2^k$. That is, each party reduces their additive share locally modulo 2^k.

B Client-Server Model

In the client-server model, clients provide inputs to the functionality and receive outputs, and servers can participate in the computation but do not have inputs or outputs. Each party may have different roles in the computation. And if each party plays a single client and a single server, this corresponds to a protocol in the standard MPC model.

In our construction, the clients only participate in the input phase and the output phase. The main computation is conducted by the servers. We can use the set $\mathcal{P} = \{P_1, \ldots, P_n\}$ to denote the n servers, and refer to the servers as parties. One benefit of the client-server model is the following theorem from [28].

Theorem B.1 (Lemma 5.2 [28]). *Let* Π *be a protocol computing a* c*-client circuit* C *using* $n = 2t + 1$ *parties. Then, if* Π *is secure against any adversary controlling exactly* t *parties, then* Π *is secure against any adversary controlling at most* t *parties.*

This theorem allows us to only consider the case where the adversary controls exactly t parties.

C Complete MPC Protocol

In this section we describe in detail a complete MPC protocol that makes use of our techniques for efficient verification. Recall that we take $n = 2t+1$, and we use replicated secret sharing with threshold t as defined in Section 2.3, which is denoted by $[\![x]\!]_k$ for secrets $x \in \mathbb{Z}_{2^k}$. Let C be an arithmetic circuit over $\mathbb{Z}_{2^k}$, given by input, addition, multiplication and output gates. The functionality we instantiate is denoted by $\mathcal{F}_{\mathrm{MPC}}$, which models secure computation of C, and it works as follows: for each input gate owned by party P_i the functionality receives x from P_i; then it computes the circuit C on these inputs, and sends the output to all the parties. All of our functionalities allow for *abort*, which means that at any time of the interaction the adversary can instruct the functionality to abort, in which the functionality sends a special signal to the honest parties, which causes them to halt and abort. The adversary could instruct a specific set of honest parties to abort only, in which case we would be talking about *selective abort*, or the adversary may be restricted to either cause all or none of the honest parties to abort, which refers to *unanimous abort*. Selective abort can be compiled to unanimous abort by using a broadcast channel [29].

The MPC protocol Π_{MPC} is described as Protocol C.0.1, and in Theorem C.1. In essence, the parties proceed by letting the clients provide input using the $\mathcal{F}_{\mathrm{input}}$ functionality, addition gates are handled locally using the linearity of the underlying secret sharing scheme, and multiplications make use $\mathcal{F}_{\mathrm{mult}}$, which is a multiplication protocol that is secure up to additive attacks. Then, output gates involve reconstruction of the underlying secret towards the corresponding client, but prior to this a verification step using $\mathcal{F}_{\mathrm{VrfySSIP}}$ is carried out in order to check the correctness of the multiplication gates. Below, we discuss some details regarding the implementation of some of the functionalities used in the protocol.

Theorem C.1. *Protocol Π_{MPC} securely instantiates Functionality $\mathcal{F}_{MPC}$ with abort in the $(\mathcal{F}_{input}, \mathcal{F}_{mult}, \mathcal{F}_{VrfySSIP})$-hybrid model, with perfect security.*[12]

Proof. We define a simulator $\mathcal{S}$ that interacts with the adversary in the ideal world. For the input gates, $\mathcal{S}$ emulates $\mathcal{F}_{\mathrm{input}}$ by receiving input from each corrupt party, and distributing consistent sharings of this input. It also emulates honest parties' inputs by simply sending consistent sharings of some dummy value. Since the adversary corrupts at most t parties, this is indistinguishable from the real world where the sharings actually correspond to the real honest parties' inputs. Notice that $\mathcal{S}$ knows the input shares held by the corrupt parties. This invariant will be preserved through the computation of the circuit.

Addition gates are handled locally, and for these $\mathcal{S}$ internally adds the shares of the corrupt parties to preserve the invariant. Now, for each multiplication gate, $\mathcal{S}$ emulates $\mathcal{F}_{\mathrm{mult}}$ by receiving the additive error ϵ by the adversary, and sending back shares of a dummy value as the shares of the product, which again, is indistinguishable from the real world since the adversary only gets t shares. For the verification step, $\mathcal{S}$ emulates $\mathcal{F}_{\mathrm{VrfySSIP}}$, which is called on input all the secret-shared inputs and outputs of all computed multiplication gates. Recall that $\mathcal{S}$ holds the corrupt parties' shares of these wires, together with the additive error ϵ_i that the adversary introduced in the i-th gate, for $i \in \{1, \dots, |C|\}$. The emulation of $\mathcal{F}_{\mathrm{VrfySSIP}}$ is done as follows: $\mathcal{S}$ sends ϵ_i to the adversary, and if there is one ϵ_i that is not zero modulo 2^k, then $\mathcal{S}$ sends an abort signal to $\mathcal{F}_{\mathrm{MPC}}$.

PROTOCOL C.0.1. *(Π_{MPC} – Protocol for Securely Computing an Arithmetic Circuit over $\mathbb{Z}_{2^k}$.).*

Consider an arithmetic circuit over $\mathbb{Z}_{2^k}$, given by input, addition, multiplication and output gates.

- **Input gates.** For every input gate corresponding to a given client, this client calls the $\mathcal{F}_{\mathrm{input}}$ functionality on his input $x \in \mathbb{Z}_{2^k}$, so that the parties obtain consistent sharings $[\![x]\!]_k$.
- **Addition gates.** Given an addition gate with secret-shared inputs $[\![x]\!]_k$, $[\![y]\!]_k$, the parties locally compute $[\![x+y]\!]_k = [\![x]\!]_k + [\![y]\!]_k$.
- **Multiplication gates.** Given a multiplication gate with secret-shared inputs $[\![x]\!]_k$, $[\![y]\!]_k$, the parties call $\mathcal{F}_{\mathrm{mult}}$ to obtain $[\![z]\!]_k$, where $z = x \cdot y + \epsilon$ for some additive error $\epsilon \in \mathbb{Z}_{2^k}$ chosen by the adversary.
- **Verification phase.** After all multiplication gates have been processed, obtaining m secret-shared triples $\{([\![a_i]\!]_k, [\![b_i]\!]_k, [\![c_i]\!]_k)\}_{i=1}^m$, the parties call $\mathcal{F}_{\mathrm{VrfySSIP}}$ on these secret-shared values.
- **Output gates.** For every output gate corresponding to some client, and if the verification phase did not result in abort, the parties call $\mathsf{reconstruct}([\![x]\!]_k, \mathsf{client})$—where the underlying shared value in the gate is $[\![x]\!]_k$—in order to reconstruct x towards client.

Otherwise, $\mathcal{S}$ receives the output values of the circuit from $\mathcal{F}_{\mathrm{MPC}}$. Recall that for each such output wire $\mathcal{S}$ has the corrupt parties' shares. $\mathcal{S}$ then samples honest parties' sharings that are consistent with the provided outputs, and then $\mathcal{S}$ emulates the honest parties' behavior in the calls to reconstruct by using these shares. This is indistinguishable from the real world since, due to the definition of $\mathcal{F}_{\mathrm{VrfySSIP}}$, there is no abort in the real world if and only if all of the multiplication gates, and in particular the whole circuit, has been computed correctly. As a result, the final shares the honest parties send in the real world correspond to the correct output, as generated by $\mathcal{S}$ in the ideal world. □

C.1 Details on Some Functionalities

C.1.1 On the Key Setup. For replicated-secret-sharing-based protocols, it is common to assume a one-time setup where the parties have some shared random keys, which are then used to boost the efficiency of several parts of the protocol. We assume the following forms of setup:

[12] The instantiation of the functionality $\mathcal{F}_{\mathrm{VrfySSIP}}$ is the one that is computationally secure (due to the use of PRG), but all the other components are perfectly secure.

- For each $T \subseteq \mathcal{P}$ with $|T| = t + 1$, parties in T all have a common uniformly random key $k_T \in \{0, 1\}^\sigma$.
- For each $j \in [n]$, and for each $T \subseteq \mathcal{P}$ with $|T| = t + 1$, parties in T all have a common uniformly random key $k_{j,T} \in \{0, 1\}^\sigma$, and party P_j has all the keys $\{k_{j,T}\}_{T \subseteq \mathcal{P}, |T|=t+1}$.

C.1.2 Instantiating $\mathcal{F}_{rand}$. Recall that $\mathcal{F}_{\text{rand}}$ is a functionality that samples consistent shares $[\![r]\!]_k$, where $r \in \mathbb{Z}_{2^k}$ is uniformly random. A common instantiation of $\mathcal{F}_{\text{rand}}$ is the following:

- For each $T \subseteq \mathcal{P}$ with $|T| = t + 1$, and for each $P_i \in T$, P_i sets $r_T = \mathsf{PRG}_{k_T}(\texttt{next}) \in \mathbb{Z}_{2^k}$.
- Output the sharings $[\![r]\!]_k = \{r_T\}_{T \subseteq \mathcal{P}, |T|=t+1}$.

C.1.3 Instantiating $\mathcal{F}_{coin}$. Recall that $\mathcal{F}_{\text{coin}}$ samples a *public* random bitstring $r \in \{0, 1\}^\sigma$. To achieve this, the parties can call $\mathcal{F}_{\text{rand}}$ to obtain $[\![r]\!]_\sigma$, followed by multiple calls to $\mathsf{reconstruct}$ where each party learns the value of r (or abort). The cost of this approach is that of reconstructing one σ-bit secret, which is $\binom{n-1}{t+1} \cdot n \cdot \sigma$ bits.

C.1.4 Instantiating $\mathcal{F}_{input}$. Recall that $\mathcal{F}_{\text{input}}$ takes input $x \in \mathbb{Z}_{2^k}$ from a party or client, and distributes consistent sharings $[\![x]\!]_k$ to the parties. To instantiate this primitive, the parties execute the following two steps:

- *Generate random mask.*
  - If input provider is a client, the parties call $\mathcal{F}_{\text{rand}}$ to generate $[\![r]\!]_k$, and they reconstruct r to the client.
  - If input provider is a party P_j, define $r_T = \mathsf{PRG}_{k_{j,T}}(\texttt{next})$ and let $[\![r]\!]_k = \{r_T\}_{T \subseteq \mathcal{P}, |T|=t+1}$. Since P_j knows the keys $\{k_{j,T}\}_{T \subseteq \mathcal{P}, |T|=t+1}$, P_j can compute $r = \sum_{T \subseteq \mathcal{P}, |T|=t+1} r_T$.[13]
- *Send masked input.* Let T_0 be a fixed subset of parties of size $t + 1$.

(1) The input provider, having input $x \in \mathbb{Z}_{2^k}$, and knowing $r \in \mathbb{Z}_{2^k}$, sends $x - r$ to the parties in T_0.

(2) The parties define locally $[\![x - r]\!]_k$ (remember it suffices that this value is known by $t + 1$ parties), and run a pairwise consistency check. Then the parties define $[\![x]\!]_k = [\![r]\!]_k + [\![x - r]\!]_k$.

The cost of this is t messages over $\mathbb{Z}_{2^k}$ from the input provider to the parties in T_0 (we can take T_0 such that the input provider belongs to this set), and the cost of the consistency check, which is $n(n - 1)/2$ digests, and is independent of the number of inputs being shared (across all input providers).

We point out that functionality $\mathcal{F}_{\text{input}}$, in the way we have defined it here, outputs pairwise consistent sharings, which is guaranteed in its implementation via a pairwise consistency check. However, in some cases, such as in our verification protocol Π_{VrfyIP}, such check, whose complexity is independent of the amount of sharings, can be postponed to a later stage. This can be easily formalized by modifying $\mathcal{F}_{\text{input}}$ so that it provides possibly inconsistent sharings, allowing the parties to query if this is the case at a later point.

C.2 Passive Multiplication

First, we define $\mathcal{F}_{\text{mult}}$ as Functionality C.2.1. Recall that this functionality takes as input a pair of sharings $[\![x]\!]_k$, $[\![y]\!]_k$, and returns shares $[\![z]\!]_k$, where $z = x \cdot y + \epsilon$ for an additive error $\epsilon \in \mathbb{Z}_{2^k}$ chosen by the adversary.

FUNCTIONALITY C.2.1. *($\mathcal{F}_{mult}$ – Passive multiplication with additive errors).*

Let $\mathcal{S}$ be the ideal world adversary.

(1) $\mathcal{F}_{\text{mult}}$ receives consistent shares of $[\![x]\!]_k$ and $[\![y]\!]_k$ from the honest parties. From this, $\mathcal{F}_{\text{mult}}$ computes the secrets x, y. $\mathcal{F}_{\text{mult}}$ also computes the shares of the corrupt parties and sends them to $\mathcal{S}$.

(2) $\mathcal{F}_{\text{mult}}$ waits for $\epsilon \in \mathbb{Z}_{2^k}$ from $\mathcal{S}$, and upon receiving this value $\mathcal{F}_{\text{mult}}$ computes $z = x \cdot y + \epsilon$. Then, $\mathcal{F}_{\text{mult}}$ distributes shares of $[\![z]\!]_k$.

There are multiple ways of instantiating $\mathcal{F}_{\text{mult}}$, and we consider two possible variants: BGW-like, and DN07-like. Below, we assume the parties have sharings $[\![x]\!]_k$ and $[\![y]\!]_k$, and the goal is for them to obtain $[\![x \cdot y]\!]_k$.

BGW-like.

(1) Parties compute locally $\langle x \cdot y \rangle_k = [\![x]\!]_k \cdot [\![y]\!]_k$, as described in Appendix A.

(2) For each $j \in [n]$, the parties call $\mathcal{F}_{\text{input}}$ with P_j as the input provider so that the parties obtain $[\![z^{(j)}]\!]_k$, where $z^{(j)}$ is P_j's additive share in $\langle x \cdot y \rangle_k$.

(3) The parties define locally $[\![x \cdot y]\!]_k = \sum_{j=1}^n [\![z^{(j)}]\!]_k$.

The cost of this approach corresponds to n calls to $\mathcal{F}_{\text{input}}$, which costs $ntk = nk(n - 1)/2$ bits per multiplication, plus $n(n - 1)/2$ digests/elements, independently of the number of multiplications.

DN07-like.

(1) Parties generate a pair $([\![r]\!]_k, \langle r \rangle_k)$ as follows:
   (a) Parties generate $[\![r_1]\!]_k, \dots, [\![r_n]\!]_k$, where each P_i knows r_i, in the same way as the masks are generated in the instantiation of $\mathcal{F}_{\text{input}}$ above.
   (b) Parties define $[\![r]\!]_k := \sum_{i=1}^n [\![r_i]\!]_k$, and $\langle r \rangle_k := (r_1, \dots, r_n)$.

(2) The parties compute locally $\langle d \rangle_k = [\![x]\!]_k \cdot [\![y]\!]_k - \langle r \rangle_k$, and send their additive shares to P_1 for reconstruction.

(3) P_1 reconstructs this value, and sends d to a fixed subset T_0 of size $t + 1$.

(4) The parties define locally $[\![d]\!]_k$, and run a pairwise consistency check.[14] Then the parties define $[\![x \cdot y]\!]_k = [\![r]\!]_k + [\![d]\!]_k$.

The cost of this approach corresponds to $(n - 1)k$ bits sent to P_1, plus $t \cdot k$ bits from P_1 to t parties, so a total of $k(n + t - 1) = 3k(n - 1)/2$. We must also add the cost of the pairwise consistency check, but this is independent of the number of multiplications.

[13] One can also use this approach for the case in which the input provider is a client, at the expense of requiring key setup also with this client. This may be reasonable if the client's input is very large.

[14] We note that these consistency checks do not need to be carried out immediately, and they can be aggregated together at the end of the protocol before the output is revealed. See for example [22].

Notice that this approach is linear in n, unlike the BGW-like alternative from above. However, it requires two rounds (all parties to P_1 and then P_1 to t parties), while BGW-like requires quadratic communication with only one round where each party sends one message to t other parties.

C.3 Three-Party Case

For $n = 3$ (and $t = 1$), both the BGW-like and the DN07-like protocols lead to a communication complexity of $3k$ bits per multiplication. However, BGW-like is preferable since it only involves one round, and in fact, in this case this protocol coincides with the one proposed in [6].

D Optimizations for 3PC and Discussions

D.1 Optimizations for 3PC

For 3-party computation, we have $\mathcal{P} = \{P_0, P_1, P_2\}$ and there is exactly one corrupted party. A replicated secret sharing $[\![x]\!]$ can be written as $[\![x]\!] = (x_0, x_1, x_2)$ where P_i holds (x_{i-1}, x_{i+1}) (with indices modulo 3). We propose the following optimizations for 3-party computation.

Avoiding Pairwise Consistency Check. In the setting of three-party computation, the instantiation of $\mathcal{F}_{\text{input}}$ in Appendix C.1 can achieve the pairwise consistency for free when the dealer is one of the party (i.e., not the client).

Recall that in the instantiation in Appendix C.1, all parties first locally prepare a random replicated secret sharing $[\![r]\!]$ such that r is known to the dealer D. In particular, $[\![r]\!]$ satisfies the pairwise consistency. Then the dealer D shares $[\![x - r]\!]$, where x is the value to be shared to all parties. In particular, the dealer D only sends $x - r$ to parties in a fixed set T_0 of size $t + 1$ and $D \in T_0$ if D is one of the three parties. When there are just 3 parties, it means that $t = 1$ and $|T_0| = 2$. Thus, D only sends $x - r$ to one of the other two parties. Now we show that $[\![x - r]\!]$ always satisfies the pairwise consistency. It is sufficient to focus on the share held by two honest parties. If the dealer is honest, then the pairwise consistency always holds. If the dealer is corrupted, then the share held by the two honest parties are 0 by default. Thus the pairwise consistency always holds as well. Therefore, the replicated secret sharing $[\![x]\!] := [\![x - r]\!] + [\![r]\!]$ always satisfies the pairwise consistency.

Balanced Local Multiplication Procedure. For two vectors of replicated secret sharings of dimension d, $[\![\boldsymbol{x}]\!] = (\boldsymbol{x}_0, \boldsymbol{x}_1, \boldsymbol{x}_2)$ and $[\![\boldsymbol{y}]\!] = (\boldsymbol{y}_0, \boldsymbol{y}_1, \boldsymbol{y}_2)$, each party P_j can compute $z_j = \boldsymbol{x}_{j-1} \cdot \boldsymbol{y}_{j+1} + \boldsymbol{x}_{j+1} \cdot \boldsymbol{y}_{j-1} + \boldsymbol{x}_{j+1} \cdot \boldsymbol{y}_{j+1}$. Then $\langle z \rangle = (z_0, z_1, z_2)$ is an additive sharing of $\boldsymbol{x} \cdot \boldsymbol{y}$. Furthermore, since both P_j and P_{j-1} holds $\boldsymbol{x}_{j+1} \cdot \boldsymbol{y}_{j+1}$, we may view that all parties hold a replicated secret sharing of $\boldsymbol{x}_{j+1} \cdot \boldsymbol{y}_{j+1}$. Thus, when verifying that P_j correctly computes z_j, it is sufficient to ask P_j to share z_j using the replicated secret sharing scheme and verify that $\boldsymbol{x}_{j-1} \cdot \boldsymbol{y}_{j+1} + \boldsymbol{x}_{j+1} \cdot \boldsymbol{y}_{j-1} = z_j - \boldsymbol{x}_{j+1} \cdot \boldsymbol{y}_{j+1}$, which corresponds to an inner-product triple of dimension $2d$. To be more concrete:

(1) P_j shares z_j.
(2) All parties locally compute $[\![z_j]\!] - [\![\boldsymbol{x}_{j+1} \cdot \boldsymbol{y}_{j+1}]\!]$.
(3) All parties check that P_j correctly computes z_j by checking the correctness of the inner-product triple $(([\![\boldsymbol{x}_{j-1}]\!], [\![\boldsymbol{x}_{j+1}]\!]), ([\![\boldsymbol{y}_{j+1}]\!], [\![\boldsymbol{y}_{j-1}]\!]), [\![z_j - \boldsymbol{x}_{j+1} \cdot \boldsymbol{y}_{j+1}]\!])$.

D.2 Discussions

Postponing the Pairwise Consistency Check in $\mathcal{F}_{input}$. Our protocol Π_{VrfyIP} calls $\mathcal{F}_{\text{input}}$ multiple times in each round, and using our implementation of this functionality from Appendix C.1, this would require the parties to run a pairwise consistency check in every round. However, it turns out that such check can be postponed until the fourth item of Step 4, at the very end of the protocol. One can model this into $\mathcal{F}_{\text{input}}$ and re-prove security in this case, but we avoid this for the sake of simplicity.

About Using a PRG for $\mathcal{F}_{coin}$. In Step 2.1, Step 3.2, and Step 4.3 of Π_{VrfyIP}, we use $\mathcal{F}_{\text{coin}}$ to generate many random values in $\mathbb{Z}_{2^{k+s}}$. A natural way of optimizing each of these steps is to use $\mathcal{F}_{\text{coin}}$ to generate a short PRG seed and then all parties locally expand the seed to obtain the random values they need.

However, we note that generating a short PRG seed and expanding the seed is *not* a secure instantiation of the functionality $\mathcal{F}_{\text{coin}}$ since in the real world, the adversary always learns the random seed and the output of the PRG corresponding to this seed while in the ideal world, the adversary only receives random values from $\mathcal{F}_{\text{coin}}$ and does not know the corresponding PRG seed (In fact, with overwhelming probability, such a seed does not exist). Our security proof of Lemma 5.2 relies on the reduction from an adversary $\mathcal{A}$ of Π_{VrfyIP} to an adversary $\mathcal{A}_g$ of $\mathcal{G}ame(k, s, T)$. In the reduction, we require the property that the random values are generated by $\mathcal{F}_{\text{coin}}$ rather than expanding from a random seed.

We note that this issue is because we try to model $\mathcal{G}ame(k, s, T)$ in a general form, which does not corresponds to the exact scenario of our protocol. In particular, the reduction works even if the adversary $\mathcal{A}$ of Π_{VrfyIP} can choose a part of the random values by himself! We may fix this issue by incorporating the analysis of the game together with the security proof of our protocol.

About Fiat-Shamir Transformation. We note that most of interactions in Π_{VrfyIP} involve generating random values in $\mathbb{Z}_{2^{k+s}}$ by using $\mathcal{F}_{\text{coin}}$. In the previous single-prover setting of [12], the authors suggest the use of the Fiat-Shamir heuristic to compress the round complexity, which is currently logarithmic on the length of the statement. However, as the authors of [11] admit (see footnote 3 in that work): "there are still gaps in our understanding of the soundness of this heuristic when analyzed in the random oracle model". Indeed, the security of this approach is not well understood, and must be analyzed heuristically in a targeted manner for each case. To highlight this difficulty, we point out for example to the work of [35], which shows that certain 5-round interactive proofs, when compiled with Fiat-Shamir, suffer from a massive soundness loss. This is done in the context of post-quantum signatures based on MPC-in-the-head-based interactive proofs, and the results in [35] turned out to be devastating for some of the proposals in the NIST PQ competition, leading them to modify their parameters in a way that ultimately led to larger signatures.

The authors in [12] do not dig deeper in the task of analyzing the security of the Fiat-Shamir transform when applied to their protocol, which is a necessary task in order to understand the concrete security of such approach. Here, we comment in a bit more depth about the security of this approach in our protocol. The Fiat-Shamir techniques works as follows: each time all parties need to prepare random values, they may compute a random seed by applying a random oracle on the common transcript and then expand the seed to obtain the desired random values. As we have mentioned before, this transformation is known to have some soundness loss in some cases like, for example, 5-round protocols [35]. In our case, we are able to exhibit an explicit attack when we compile our protocol using Fiat-Shamir, that has to be considered when analyzing the resulting soundness.

The attack consists of a corrupted prover sampling random values repeatedly in each iteration, and choosing the one that increases the advantage to the most extent. Translating to $\mathcal{G}ame(k, s, T)$, it means that $\mathcal{A}_g$ can ask $\mathcal{C}_g$ to repeatedly sample r_i in each Round i, until $\mathcal{A}_g$ finds a challenge r_i that is more advantageous. Consider the following strategy of an adversary in $\mathcal{G}ame(k, s, T)$:

- In each iteration, $\mathcal{A}_g$ chooses $e_i = 2^{E_{i-1}}$ and $c_i = 0$. Note that $\mathsf{Po2}(e_i) = E_{i-1}$.
- $\mathcal{A}_g$ asks $\mathcal{C}_g$ to repeatedly sample r_i until r_i is a multiple of $2^{c \log \lambda}$ for some constant c. Then $E_i = \mathsf{Po2}(e_i \cdot r_i + c_i) \geq E_{i-1} + c \log \lambda$. Notice that, crucially, this succeeds in a polynomial number of attempts.

It means that $\mathcal{A}_g$ can always cause $E_i \geq E_{i-1} + c \log \lambda$. In other words, $\mathcal{A}_g$ can cause E_i to increase by $c \log \lambda$ for free! This increases s by $cT \log \lambda$ to achieve the same level of security as the one without Fiat-Shamir transformation.

Notice that this attack is only possible because, in our $\mathbb{Z}_{2^{k+s}}$ setting, there is a fundamental difference between a challenge that is only divisible by a small power of two, and a challenge with a high power of two as a divisor: the latter turns out to be "easier" to reply to for a cheating prover. This does not occur in the finite field case, where any non-zero challenge is as good as any other.[15]

E Detailed Communication Costs

Communication Cost of Π_{VrfyIP}. To analyze the communication complexity of Π_{VrfyIP}, let d denote the dimension of each of the p inner-product triples to check. Then the inner-product triple obtained in Step 2 has dimension $d' = p \cdot d$. As discussed in Appendix D.2, we may take the costs of the calls to $\mathcal{F}_{\mathrm{coin}}$, per round, to be the cost of reconstructing one σ-bit value, where σ is the computational security parameter of some PRG. The cost of Π_{VrfyIP} is calculated below.

- Step 1 requires p calls to $\mathcal{F}_{\mathrm{input}}$ over $\mathbb{Z}_{2^s}$. Using the instantiation from Appendix C.1 (ignoring pairwise consistency checks), this costs $t \cdot s \cdot p$ bits.
- The dimension of the inner-products after Step 2 is $p \cdot d$. Step 3 is repeated $\log_q(p \cdot d) - 1$ times. Each of these consists of $q^2 - 1$ calls to $\mathcal{F}_{\mathrm{input}}$ over $\mathbb{Z}_{2^{k+s}}$. These are $(q^2 - 1) \cdot (\log_q(p \cdot d) - 1)$ calls to $\mathcal{F}_{\mathrm{input}}$, each of which costs $(k + s) \cdot t$ bits.
- The first two items in Step 4 require $(q + 1)^2 - 1$ calls to $\mathcal{F}_{\mathrm{input}}$ over $\mathbb{Z}_{2^{k+s}}$.
- The rest of step 4 requires reconstructing three elements over $\mathbb{Z}_{2^{k+s}}$, which costs $3n(k + s)\binom{n-1}{t+1}$ bits.
- Overall, $\mathcal{F}_{\mathrm{coin}}$ is called in $\log_q(p \cdot d) + 1$ rounds, which has a cost of this many σ-bit reconstructions, or $(\log_q(p \cdot d) + 1) \cdot n \cdot \binom{n-1}{t+1} \cdot \sigma$ bits.

Communication Cost of $\Pi_{VrfySSIP}$. When Π_{VrfyIP} is used in the context of instantiating the multi-prover functionality $\mathcal{F}_{\mathrm{VrfySSIP}}$ to check m inner-products which contain δ multiplications in total, this protocol is called n times (once for each prover), with $p = \lambda$ and $d = \delta\binom{n-1}{t}^2$. Using the analysis for Π_{VrfyIP} from above, and taking into account we must add an extra round of $\mathcal{F}_{\mathrm{coin}}$ from the reduction to Π_{VrfySSIP}, together with the optimization from Section 5.2 that allows us to shrink the dimension of the inner-product tuple from $p \cdot d$ to d, the cost in bits of Π_{VrfySSIP} is:

$$n \cdot \Big(\underbrace{ts\lambda}_{\substack{\text{Sharing} \\ h_i\text{'s}}} + \underbrace{(q^2 - 1)(\log_q(d) - 1)t(k + s)}_{\text{Sharing } z_{i,i'} \text{ in recursion}} + \underbrace{((q + 1)^2 - 1)t(k + s)}_{z_{i,i'} \text{ in final recursion}} \tag{2}$$

$$+ \Big(\underbrace{3n(k + s)}_{\substack{\text{Final} \\ \text{reconstructions}}} + \underbrace{(\log_q(d) + 2)n\sigma}_{\text{Calls to } \mathcal{F}_{\mathrm{coin}}} \Big) \binom{n-1}{t+1} \Big) \tag{3}$$

$$+ \Big(\underbrace{n\sigma}_{\mathcal{F}_{\mathrm{coin}}} + \underbrace{\lambda nk}_{\text{Zero check}} \Big) \cdot \binom{n-1}{t+1}. \tag{4}$$

Finally, our costs are given in terms of s, which determines the final soundness level of our construction. By Lemma 5.1, to achieve a security level of $2^{-\lambda}$, it suffices to take $s = \lambda + T(1/2 + \log(5/2 + 3\lambda/T)) = \lambda + O(T \cdot \log(\lambda/T))$, where $T = \lceil 2 \log_q(d) + 1 \rceil$ (with our optimization from Section 5.2).

E.0.1 Comparison with [10, 12]. We also compare our concrete communication with that of the distributed check of [10], which we sketched in Section 4.1.3. The protocol is structurally similar to ours, and hence easy to analyze. Let us denote by ℓ the degree of the Galois ring extension used in their protocol. When instantiating Π_{VrfyIP}, the main difference (besides the larger ring) is that, in every recursion step, the prover only need to provide $(2q-2)$ extra inputs, instead of $(q^2 - 1)$ as in our case.

- Recursion requires $\log_q(p \cdot d) \cdot (2q - 2)$ calls to $\mathcal{F}_{\mathrm{input}}$ over $\mathbb{Z}_{2^k}^{\ell}$,[16] each of which costs $k \cdot \ell \cdot t$ bits.
- The final checking step requires reconstructing three elements over $\mathbb{Z}_{2^k}^{\ell}$, which costs $3nk\ell\binom{n-1}{t+1}$ bits.
- $\mathcal{F}_{\mathrm{coin}}$ is called on $\log_q(p \cdot d)$ rounds, which costs $\log_q(p \cdot d) \cdot n \cdot \sigma\binom{n-1}{t+1}$ bits.

The relation between ℓ and the desired security parameter in this case is simple: ℓ can be taken to be $\lambda + 1 + \log_2(1 + 2\log_2(p \cdot d))$, and the resulting soundness will be $2^{-\lambda}$.

Cost of Π_{VrfyIP} *Using Ring Extensions.* When using the extension-based instantiation of $\mathcal{F}_{\mathrm{VrfyIP}}$ to implement $\mathcal{F}_{\mathrm{VrfySSIP}}$, there are only a few minor differences with respect to our approach: only one linear combination is needed, which leads to a single inner

[15] To dispel any doubts, we point out that our attack is not fixed by simply requiring the challenges to be odd, for example. In this case, the attacker simply chooses $e_i = 2^{E_{i-1}}$ and $c_i = 2^{E_{i-1}}$, so that $e_i \cdot r_i + c_i = 2^{E_{i-1}}(r_i + 1)$. The attack still works by looking for $2^{c \cdot \log \lambda} \mid r_i + 1$.

[16] Note that a Galois ring extension of degree ℓ is equivalent to $\mathbb{Z}_{2^k}^{\ell}$ for communication purposes.

product. This means we take $p = 1$ and $d = \delta\binom{n-1}{t}^2$, where δ is the number of multiplications across all inner-products to be checked by $\mathcal{F}_{\text{VrfySSIP}}$. Furthermore, they do not need Step 1 where the prover inputs p extra "correcting" values. Hence, the total communication in the multi-prover case becomes

$$n \cdot \left(2(q-1)\log_q(d)k\ell t + (3nk\ell + (\log_q(d)+1)n\sigma)\binom{n-1}{t+1}\right) + (n\sigma + nk\ell)\binom{n-1}{t+1}.$$

Comparing this to the communication complexity of our instantiation, given in Eq. (2), we see that the leading term that depends on $d = \delta\binom{n-1}{t}^2$, namely $\log_q(d)$, is multiplied in our case by $(q^2-1)(k+s)t$, while using ring extensions this factor is $2(q-1)k\ell t$. The ratio between these two terms is roughly $\frac{q(k+\lambda)}{k\lambda} = q(\frac{1}{\lambda} + \frac{1}{k})$. The term q is typically taken to be a constant (*e.g.* 2 or 8), so this ratio decreases (*i.e.* our communication is better) as either λ or k increases.

To see more concretely what our improvement in terms of communication is, let us consider some concrete parameters sets. For $\lambda = 40$ and three parties, and taking $q = 4$, verifying $\delta = 2^{20} \approx$ 1 million secret-shared products with our protocol requires 142.7 kB, while using ring extensions this requires 636.1 kB, about ×5 more communication. For other parameter regimes of interest this factor tends to range between 3 and 5.

F Achieving Full Security

Applying the techniques used in [12] for achieving full security (i.e., guaranteed output delivery), our distributed product check protocol can be adapted to construct a fully secure MPC protocol as well. At a high level, the core idea of lifting security with abort to full security is cheating identification – whenever some party outputs abort in the protocol, we can identify a so-called *semi-corrupt* pair of parties where at least one of them is guaranteed to be corrupted; the two parties will be eliminated, and the remaining active parties will restart the computation again (after a potential update of input sharings). Pair elimination and recomputation will be repeated whenever there is an abort, until the remaining parties successfully finish the computation, or eventually one honest party is identified and then finishes the computation using the parties' inputs.

F.1 Review of the Fully Secure Protocol in [12]

We now briefly review how previous work [12] achieves full security. It utilizes an authenticated secret sharing scheme, and works as follows.

- At first, the parties secret-share their inputs using the RSS scheme; then for the parties in each subset $|T|$ of size $t+1$, they compute an authentication tag of the additive shares of the inputs belonging to this subset using random authentication keys that are secret-shared via an authenticated secret sharing scheme.
- Next, the parties evaluate the circuit using the RSS-based semi-honest protocol, also compute authentication tags for shares of output values, and then conduct the distributed verification procedure to verify the semi-honest multiplication triples. As mentioned above, once abort is detected in any previous step, cheating identification and pair elimination will be triggered and the computation will be restarted.
- Finally, the parties reconstruct the output values, and reveal the authentication keys for the parties to check the correctness of the received shares w.r.t. the previously computed tags. After obtaining enough shares that pass the authentication check (which will always happen since honest parties' shares will always pass the authentication check), the parties can recover their outputs correctly.

In the above fully secure protocol from [12], the verification of distributed multiplication triples with cheating identification is captured by a functionality $\mathcal{F}^{\text{full}}_{\text{vrfy}}$, which is black-box used in this protocol. Following this, we can also define a functionality $\mathcal{F}^{\text{full}}_{\text{vrfy}}$ that verifies distributed inner-product triples with the ability of cheating identification. By making a black-box use of this functionality, we can obtain a fully secure protocol as in [12].

F.2 Instantiating $\mathcal{F}^{\textbf{full}}_{\textbf{vrfy}}$

We first give a formal definition of $\mathcal{F}^{\text{full}}_{\text{vrfy}}$ in Functionality F.2.1.

Below we demonstrate how to instantiate $\mathcal{F}^{\text{full}}_{\text{vrfy}}$ given the secure-with-abort verification protocol Π_{VrfySSIP}. Observe that in Π_{VrfySSIP}, aborting only occurs in the following procedures: (1) functionality $\mathcal{F}_{\text{VrfyIP}}$ returns an abort, (2) reconstruction of o_i for $i \in [\lambda]$ fails due to inconsistency, and (3) there exists some $i \in [\lambda]$ s.t. $o_i \neq_k 0$. We analyze the three cases in the following.

- Case (1): In this case, we require that whenever $\mathcal{F}_{\text{VrfyIP}}$ returns an abort, it also outputs a pair of conflicting parties, and then $\mathcal{F}^{\text{full}}_{\text{vrfy}}$ just takes this pair of conflicting parties as the semi-corrupt pair. We augment $\mathcal{F}_{\text{VrfyIP}}$ with the cheating identification ability, which is captured by the functionality $\mathcal{F}^{\text{CheatIdntfy}}_{\text{VrfyIP}}$ (and will be elaborated later).
- Case (2): Utilizing the "replicated" property of the RSS scheme, the parties are able to identify two inconsistent parties with ease.
- Case (3): This case implies that, each additive share $c_i'^{(j)}$ is honestly computed using claimed $[\![\boldsymbol{a}_i']\!]_k$ and $[\![\boldsymbol{b}_i']\!]_k$ and shared by each party P_j (except with negligible probability), but c_i' reconstructed from the additive shares $\{c_i'^{(j)}\}_{j\in[n]}$ is inconsistent with the one reconstructed from the RSS sharings $[\![c_i']\!]_k$, which further implies that there exists some incorrect RSS-shared inner-product triple in the semi-honest phase. To locate such an incorrect triple, the parties apply the binary search method as in [12]. Specifically, they first divide the triples into two halves, and then perform the distributed product check procedure on the halved triples; if the current check is not passed, then they apply the binary search method on the current half of the triples, otherwise they turn to the other half and apply the process, until finally obtaining an incorrect triple. After obtaining this incorrect triple, the parties can check the computation of this triple and find a pair of conflicting parties, which is captured by the functionality $\mathcal{F}_{\text{miniMPC}}$ as in [12].

FUNCTIONALITY F.2.1. ($\mathcal{F}_{vrfy}^{full}$ - *Verifying Secret-Shared Inner-Product Triples with Cheating Identification).*

Let $\mathcal{S}$ be the ideal world adversary.

(1) $\mathcal{F}^{\text{full}}_{\text{vrfy}}$ receives m from all parties. Then for all $i \in [m]$, $\mathcal{F}^{\text{full}}_{\text{vrfy}}$ receives honest parties' shares of $(\llbracket \boldsymbol{a}_i \rrbracket_k, \llbracket \boldsymbol{b}_i \rrbracket_k, \llbracket c_i \rrbracket_k)$. For each replicated secret sharing, $\mathcal{F}^{\text{full}}_{\text{vrfy}}$ reconstructs the whole sharings $(\llbracket \boldsymbol{a}_i \rrbracket_k, \llbracket \boldsymbol{b}_i \rrbracket_k, \llbracket c_i \rrbracket_k)$ for all $i \in [m]$, and sends the shares of corrupted parties to $\mathcal{S}$. In addition, $\mathcal{F}^{\text{full}}_{\text{vrfy}}$ computes $\epsilon_i \equiv_k c_i - \boldsymbol{a}_i \cdot \boldsymbol{b}_i$ and sends ϵ_i to $\mathcal{S}$.

(2) $\mathcal{F}^{\text{full}}_{\text{vrfy}}$ checks if the equation $c_i \equiv_k \boldsymbol{a}_i \cdot \boldsymbol{b}_i$ holds for all $i \in [m]$.
- If it holds for all $i \in [m]$, $\mathcal{F}^{\text{full}}_{\text{vrfy}}$ sends accept to $\mathcal{S}$ and receives a command out $\in$ {accept, abort} from $\mathcal{S}$. If out = abort, then $\mathcal{S}$ is required to send a pair of indices (j_1, j_2) to $\mathcal{F}^{\text{full}}_{\text{vrfy}}$ with at least one of them being a corrupted party, then $\mathcal{F}^{\text{full}}_{\text{vrfy}}$ hands (j_1, j_2) to all honest parties.
- Otherwise, $\mathcal{F}^{\text{full}}_{\text{vrfy}}$ sends abort to $\mathcal{S}$. Then $\mathcal{S}$ chooses one of the following two options:
  - $\mathcal{S}$ sends a pair of indices (j_1, j_2) to $\mathcal{F}^{\text{full}}_{\text{vrfy}}$ with at least one of them being a corrupted party, then $\mathcal{F}^{\text{full}}_{\text{vrfy}}$ hands (j_1, j_2) to all honest parties.
  - $\mathcal{S}$ asks $\mathcal{F}^{\text{full}}_{\text{vrfy}}$ to find a pair of conflicting parties in the $\hat{i}^\star$-th inner-product triple for some $\hat{i}^\star \in [m]$. Then, $\mathcal{F}^{\text{full}}_{\text{vrfy}}$ commands the honest parties to send their inputs, randomnesses as well as views in the execution to compute the $\hat{i}^\star$-th triple and the messages that should have been sent by each corrupted party. Then $\mathcal{F}^{\text{full}}_{\text{vrfy}}$ finds a pair of parties (P_{j_1}, P_{j_2}) where P_{j_2} received an incorrect message from P_{j_1}, and hands (j_1, j_2) to all honest parties and $\mathcal{S}$.

In this way, we can securely instantiate functionality $\mathcal{F}^{\text{full}}_{\text{vrfy}}$ in the $(\mathcal{F}^{\text{CheatIdntfy}}_{\text{VrfyIP}}, \mathcal{F}_{\text{miniMPC}})$-hybrid model. Note that we can directly apply the previous instantiation of $\mathcal{F}_{\text{miniMPC}}$ as in [12]. In the following we mainly elaborate on how to instantiate $\mathcal{F}^{\text{CheatIdntfy}}_{\text{VrfyIP}}$ atop our secure-with-abort protocol Π_{VrfyIP}.

F.3 Instantiating $\mathcal{F}^{\textbf{CheatIdntfy}}_{\textbf{VrfyIP}}$

Functionality F.3.1 gives the formal definition of $\mathcal{F}^{\text{CheatIdntfy}}_{\text{VrfyIP}}$, which verifies distributed inner-product triples known by a single prover with the ability of cheating identification.

FUNCTIONALITY F.3.1. ($\mathcal{F}^{CheatIdntfy}_{VrfyIP}$ - *Verifying Secret-Shared Inner-Product Triples Known by A Single Party with Cheating Identification).*

Let $\mathcal{S}$ be the ideal world adversary.

(1) $\mathcal{F}_{\text{VrfyIP}}$ receives the prover's identity j, a parameter p, and honest parties' shares of $\{(\llbracket \boldsymbol{\mu}_i \rrbracket_k, \llbracket \boldsymbol{\nu}_i \rrbracket_k, \llbracket w_i \rrbracket_k)\}_{i=1}^{p}$. For each replicated secret sharing, $\mathcal{F}_{\text{VrfyIP}}$ reconstructs the whole sharings $(\llbracket \boldsymbol{\mu}_i \rrbracket_k, \llbracket \boldsymbol{\nu}_i \rrbracket_k, \llbracket w_i \rrbracket_k)$ for all $i \in [p]$, and sends the identity j and the shares of corrupted parties to $\mathcal{S}$. In addition, if P_j is corrupted, $\mathcal{F}_{\text{VrfyIP}}$ also sends the whole sharings $\{(\llbracket \boldsymbol{\mu}_i \rrbracket_k, \llbracket \boldsymbol{\nu}_i \rrbracket_k, \llbracket w_i \rrbracket_k)\}_{i=1}^{p}$ to $\mathcal{S}$.

(2) $\mathcal{F}_{\text{VrfyIP}}$ checks if the equation $w_i \equiv_k \boldsymbol{\mu}_i \cdot \boldsymbol{\nu}_i$ holds for all $i \in [p]$. If it doesn't hold for some $i \in [p]$, $\mathcal{F}_{\text{VrfyIP}}$ sends abort to all honest parties and $\mathcal{S}$. Otherwise, $\mathcal{F}_{\text{VrfyIP}}$ receives a command out $\in$ {accept, abort} from $\mathcal{S}$, and sends out to all honest parties.

(3) If the command sent to the honest parties is abort, then:
- If P_j is corrupted, then $\mathcal{S}$ sends an index $j' \in [n]$ to $\mathcal{F}^{\text{CheatIdntfy}}_{\text{VrfyIP}}$; if P_j is honest, then $\mathcal{S}$ sends an index j' where $P_{j'}$ is corrupted to $\mathcal{F}^{\text{CheatIdntfy}}_{\text{VrfyIP}}$.
- $\mathcal{S}$ sends the pair of indices (j, j') to the honest parties.

Recall that in the protocol Π_{VrfyIP}, the single prover knows in clear the inner-product triples to be checked as well as the sharings of triples held by the verifiers. Taking advantage of this fact, we can instantiate $\mathcal{F}^{\text{CheatIdntfy}}_{\text{VrfyIP}}$ by augmenting Π_{VrfyIP} with the cheating identification ability easily. Specifically, we let the prover "accuse" one verifier who may cause abort, and output the prover and the accused verifier as a semi-corrput pair. Consider the following two cases:

- Case (1): The prover is corrupted. In this case, no matter which verifier the prover chooses, the output semi-corrupt pair always contains the malicious prover.
- Case (2): The prover is honest. In this case, the honest prover needs to accurately find a malicious verifier that causes abort. To this end, we utilize the "recomputable verification" property of Π_{VrfyIP} as in [12] – the prover itself can recompute the expected messages of the verifiers. To see this, note that each message sent by each verifier can be represented as a deterministic function of 1) its inputs to the protocol, 2) the messages received from the functionalities $\mathcal{F}_{\text{input}}$, $\mathcal{F}_{\text{coin}}$, as well as 3) the messages received from the prover. For 1), note that each verifier's inputs (i.e., shares of the triples) are known by the prover at the beginning of the protocol. For 2), the prover can deduce the verifier's messages (i.e. shares of the prover's secrets) received from $\mathcal{F}_{\text{input}}$ (using the secrets it sends to $\mathcal{F}_{\text{input}}$ and the shares received back from $\mathcal{F}_{\text{input}}$); the prover also knows the randomnesses generated by $\mathcal{F}_{\text{coin}}$ since all the randomnesses are public. For 3), they are already known to the prover. Therefore, the prover can

individually compute each message that should be sent by the verifiers, which enables it to find exactly which verifier is malicious and further output a correct semi-corrupt pair.

G Security Analysis and Proofs

G.1 Reduction from $\mathcal{A}$ in our Protocol to $\mathcal{A}_g$ in $\mathcal{G}ame(k, s, T)$.

We now briefly show the reduction from the adversary $\mathcal{A}$ of our protocol to $\mathcal{A}_g$ in the game $\mathcal{G}ame(k, s, T)$. Assume that the prover party P_j is corrupted and at least one of the input inner-product triples is incorrect. Intuitively, we use E_i to denote an upper bound on the smallest number of 2-factors of the additive errors in Round i. This holds for Round 0 since at least one of the inner-product triples is incorrect, indicating that the additive error of that triple has number of 2-factors no more than $k-1$.

Round 1: Suppose the additive errors of the inner-product triples after lifting to $\mathbb{Z}_{2^{k+s}}$ in Step 1 are denoted by $\epsilon_1, \dots, \epsilon_\lambda$. Then one of the inner-product triples, say the $i^\star$-th inner-product triple, has additive error $\epsilon_{i^\star}$ such that $\mathsf{Po2}(\epsilon_{i^\star}) \le E_0 = k-1$. We set $\mathcal{A}_g$ to pick $e_1 = \epsilon_{i^\star}$ in the first round of the game. In Step 2, we merge all inner-product triples into a single inner-product triple. Then the additive error of the new inner-product triple can be computed by $\epsilon = \sum_{i=1}^{\lambda} \theta_i \cdot \epsilon_i = \theta_{i^\star} \cdot \epsilon_{i^\star} + \sum_{i \ne i^\star} \theta_i \cdot \epsilon_i$. Thus, $\mathcal{A}_g$ picks $c_1 = \sum_{i \ne i^\star} \theta_i \cdot \epsilon_i$ in the first round of game. Since $\theta_{i^\star}$ is uniform, we interpret the challenge r_1 by $\mathcal{C}_g$ as $r_1 = \theta_{i^\star}$, and thus $E_1 = \mathsf{Po2}(r_1 \cdot e_1 + c_1) = \mathsf{Po2}(\epsilon)$.

Round 2 and Round 3: Now we proceed to Step 3 of our protocol.

In Step 3.1, we may define $\epsilon_{i,i'} = z_{i,i'} - \boldsymbol{x}_i \cdot \boldsymbol{y}_{i'}$. In particular, we have $\sum_{i=1}^{q} \epsilon_{i,i} = z - \boldsymbol{x} \cdot \boldsymbol{y} = \epsilon$, which satisfies that $\mathsf{Po2}(\sum_{i=1}^{q} \epsilon_{i,i}) \le E_1$. Thus, there exists an index $i^\star$ such that $\mathsf{Po2}(\epsilon_{i^\star,i^\star}) \le E_1$. We let $\mathcal{A}_g$ pick $e_2 = \epsilon_{i^\star,i^\star}$ in the second round of the game.

In Step 3.3, the additive error $\epsilon' = z' - \boldsymbol{x}' \cdot \boldsymbol{y}'$ can be computed as $\epsilon' = \sum_{i'=1}^{q} \beta_{i'} \cdot (\sum_{i=1}^{q} \alpha_i \cdot \epsilon_{i,i'})$. Let $\epsilon'_{i'} = \sum_{i=1}^{q} \alpha_i \cdot \epsilon_{i,i'}$. Then $\epsilon' = \sum_{i'=1}^{q} \beta_{i'} \cdot \epsilon'_{i'}$.

Observe that $\epsilon'_{i^\star} = \sum_{i=1}^{q} \alpha_i \cdot \epsilon_{i,i^\star} = \alpha_{i^\star} \cdot \epsilon_{i^\star,i^\star} + \sum_{i \ne i^\star} \alpha_i \cdot \epsilon_{i,i^\star}$. We let $\mathcal{A}_g$ pick $c_2 = \sum_{i \ne i^\star} \alpha_i \cdot \epsilon_{i,i^\star}$ in the second round of the game. Since $\alpha_{i^\star}$ is uniform, we interpret the second challenge $\mathcal{C}_g$ samples, r_2, as $r_2 = \alpha_{i^\star}$, and thus $E_2 = \mathsf{Po2}(r_2 \cdot e_2 + c_2) = \mathsf{Po2}(\epsilon'_{i^\star})$.

We let $\mathcal{A}_g$ pick $e_3 = \epsilon'_{i^\star}$ in the third round of the game. Observe that $\epsilon' = \sum_{i'=1}^{q} \beta_{i'} \cdot \epsilon'_{i'} = \beta_{i^\star} \cdot \epsilon'_{i^\star} + \sum_{i' \ne i^\star} \beta_{i'} \cdot \epsilon'_{i'}$. $\mathcal{A}_g$ picks $c_3 = \sum_{i' \ne i^\star} \beta_{i'} \cdot \epsilon'_{i'}$ and we interpret the third challenge by $\mathcal{C}_g$ as $r_3 = \beta_{i^\star}$. Thus, $E_3 = \mathsf{Po2}(r_3 \cdot e_3 + c_3) = \mathsf{Po2}(\epsilon')$. The similar reduction from the strategy of P_j to $\mathcal{A}_g$ works for the subsequent repetitions of Step 3 and Step 4.

Last Round: From the above, the additive error of the final multiplication triple has a number of 2-factors no more than $E_{2\log_q(p \cdot d)+1}$. In particular, if all parties accept the check of the final multiplication triple, it implies that the additive error of the final multiplication triple is 0. Then $E_{2\log_q(p \cdot d)+1} = k + s$, indicating that $\mathcal{A}_g$ wins the above game.

G.2 Proof of Lemma 5.1

Proof. Consider a fixed adversary $\mathcal{A}_g$. We start by analyzing the probability of the event $E_i \ge q$ for any round $i \in [T]$ and any positive integer $q \le k + s$. We first have the following proposition.

Proposition G.1. *For any positive integer q where $q \le k + s$,*

$$\mathsf{Pr}[E_1 \ge q \mid E_0 = k-1] \le \frac{1}{2^{q-k+1}}. \tag{5}$$

For any $2 \le i \le T$ and positive integer $p \le q$,

$$\mathsf{Pr}[E_i \ge q \mid E_{i-1} = p, E_0 = k-1] \le \frac{1}{2^{q-p}}. \tag{6}$$

Proof. Consider the first round where $i = 1$. Note that when $q < k-1$, the inequality always holds. We only consider the case where $q \ge k-1$. Assume $e_1 = 2^u \cdot v$ where v is an odder integer. Under the requirement that $E_0 = k-1$ and $\mathsf{Po2}(e_1) \le E_0$, we have $u \le k-1 \le q$. Let $\Phi_1 \equiv_{k+s} r_1 \cdot e_1 + c_1$. It follows that $r_1 \cdot 2^u \cdot v \equiv_{k+s} \Phi_1 - c_1$. And the event $E_1 \ge q$ (i.e., $\mathsf{Po2}(\Phi_1) \ge q$) implies that $r_1 \equiv_{q-u} \frac{(\Phi_1 - c_1)}{2^u} \cdot v^{-1}$. Since $q - u \le k + s$, this further implies determining the lowest $q-u$ bits of r_1. As r_1 is uniformly random over $\mathbb{Z}_{2^{k+s}}$, the probability of this event is bounded by $\frac{1}{2^{q-u}}$. Since $u \le k-1$, the probability is at most $\frac{1}{2^{q-k+1}}$.

Now consider the following rounds where $2 \le i \le T$. We now assume $e_i = 2^u \cdot v$ where v is an odder integer. In this case, conditioned on $E_{i-1} = p$, we have $u = \mathsf{Po2}(e_i) \le E_{i-1} = p \le q$. Let $\Phi_i \equiv_{k+s} r_i \cdot e_i + c_i$. Similarly, the event $E_i \ge q$ implies that $r_i \equiv_{q-u} \frac{(\Phi_i - c_i)}{2^u} \cdot v^{-1}$. which happens with probability bounded by $\frac{1}{2^{q-u}}$. As $u \le p$, this probability is at most $\frac{1}{2^{q-p}}$. □

Given Proposition G.1, we now have the following induction inequality:

Proposition G.2. *For any positive integer $q \le k+s$ and $2 \le i \le T$,*

$$\mathsf{Pr}[E_i \ge q \mid E_0 = k-1] \le \frac{1}{2^q} + \sum_{p=1}^{q} \frac{1}{2^{q-p+1}} \mathsf{Pr}[E_{i-1} \ge p \mid E_0 = k-1]. \tag{7}$$

Proof. By applying the law of total probability over all possible values of E_{i-1}, we have

$$\begin{aligned}
&\mathsf{Pr}[E_i \ge q \mid E_0 = k-1] \\
&= \sum_{p=0}^{k+s} \mathsf{Pr}[E_i \ge q \mid E_{i-1} = p, E_0 = k-1] \cdot \mathsf{Pr}[E_{i-1} = p \mid E_0 = k-1] \\
&\le \sum_{p=0}^{q-1} \mathsf{Pr}[E_i \ge q \mid E_{i-1} = p, E_0 = k-1] \cdot \mathsf{Pr}[E_{i-1} = p \mid E_0 = k-1] \\
&\quad + \sum_{p=q}^{k+s} \mathsf{Pr}[E_{i-1} = p \mid E_0 = k-1] \\
&= \sum_{p=0}^{q-1} \mathsf{Pr}[E_i \ge q \mid E_{i-1} = p, E_0 = k-1] \cdot \mathsf{Pr}[E_{i-1} = p \mid E_0 = k-1] \\
&\quad + \mathsf{Pr}[E_{i-1} \ge q \mid E_0 = k-1].
\end{aligned}$$

From Proposition G.1, we can obtain

$$\begin{aligned} &\Pr[E_i \ge q \mid E_0 = k-1] \\ \le& \sum_{p=0}^{q-1} \frac{1}{2^{q-p}} \cdot \Pr[E_{i-1} = p \mid E_0 = k-1] + \Pr[E_{i-1} \ge q \mid E_0 = k-1] \\ =& \sum_{p=0}^{q-1} \frac{1}{2^{q-p}} \cdot (\Pr[E_{i-1} \ge p \mid E_0 = k-1] \\ &- \Pr[E_{i-1} \ge p+1 \mid E_0 = k-1]) + \Pr[E_{i-1} \ge q \mid E_0 = k-1] \\ =& \frac{1}{2^q} \cdot \Pr[E_{i-1} \ge 0 \mid E_0 = k-1] \\ &+ \sum_{p=1}^{q-1} (\frac{1}{2^{q-p}} - \frac{1}{2^{q-(p-1)}}) \cdot \Pr[E_{i-1} \ge p \mid E_0 = k-1] \\ &- \frac{1}{2}\Pr[E_{i-1} \ge q \mid E_0 = k-1] + \Pr[E_{i-1} \ge q \mid E_0 = k-1] \\ =& \frac{1}{2^q} \cdot \Pr[E_{i-1} \ge 0 \mid E_0 = k-1] \\ &+ \sum_{p=1}^{q} \frac{1}{2^{q-p+1}} \Pr[E_{i-1} \ge p \mid E_0 = k-1]. \end{aligned}$$

As $0 \le E_{i-1} \le k+s$, we have $\Pr[E_{i-1} \ge 0 \mid E_0 = k-1] = 1$, and thus

$$\Pr[E_i \ge q \mid E_0 = k-1] \le \frac{1}{2^q} + \sum_{p=1}^{q} \frac{1}{2^{q-p+1}} \Pr[E_{i-1} \ge p \mid E_0 = k-1]. \tag{8}$$

□

We now claim that Proposition G.2 implies the following inequality for any $i \in [T]$ and $q \in [k, k+s]$.

$$\Pr[E_i \ge q \mid E_0 = k-1] \le \sum_{j=0}^{i-1} \binom{q-k+j}{q-k} \cdot \frac{1}{2^{q-k+1+j}}. \tag{9}$$

Below we prove the correctness of this inequality.

- Consider the first round where $i = 1$. It's clear to see that in this case the above inequality is consistent with Proposition G.1. Specifically, in the case of $i = 1$, Inequality 9 becomes

$$\Pr[E_1 \ge q \mid E_0 = k-1] \le \binom{q-k}{q-k} \cdot \frac{1}{2^{q-k+1}} = \frac{1}{2^{q-k+1}}. \tag{10}$$

  which is consistent with Inequality 5 claimed in Proposition G.1.
- Now we assume that Inequality 9 holds in any $(i-1)$-th round where $2 \le i \le T$, and prove that under this assumption, the inequality still holds in the i-th round. Starting from Proposition G.2, we have

$$\begin{aligned} &\Pr[E_i \ge q \mid E_0 = k-1] \\ \le& \frac{1}{2^q} + \sum_{p=1}^{q} \frac{1}{2^{q-p+1}} \Pr[E_{i-1} \ge p \mid E_0 = k-1] \\ \le& \frac{1}{2^q} + \sum_{p=1}^{k-1} \frac{1}{2^{q-p+1}} + \sum_{p=k}^{q} \frac{1}{2^{q-p+1}} \Pr[E_{i-1} \ge p \mid E_0 = k-1] \\ \le& \frac{1}{2^{q-k+1}} + \sum_{p=k}^{q} \frac{1}{2^{q-p+1}} \Pr[E_{i-1} \ge p \mid E_0 = k-1]. \end{aligned}$$

  By the assumption that Inequality 9 holds in the $(i-1)$-th round, we have

$$\Pr[E_{i-1} \ge p \mid E_0 = k-1] \le \sum_{j=0}^{i-2} \binom{p-k+j}{p-k} \cdot \frac{1}{2^{p-k+1+j}}.$$

  And thus we obtain

$$\begin{aligned} &\Pr[E_i \ge q \mid E_0 = k-1] \\ \le& \frac{1}{2^{q-k+1}} + \sum_{p=k}^{q} \frac{1}{2^{q-p+1}} \sum_{j=0}^{i-2} \binom{p-k+j}{p-k} \cdot \frac{1}{2^{p-k+1+j}} \\ =& \frac{1}{2^{q-k+1}} + \sum_{p=k}^{q} \sum_{j=0}^{i-2} \frac{1}{2^{q-k+2+j}} \binom{p-k+j}{p-k} \\ =& \frac{1}{2^{q-k+1}} + \sum_{j=0}^{i-2} \frac{1}{2^{q-k+2+j}} \sum_{p=k}^{q} \binom{p-k+j}{p-k}. \end{aligned}$$

  Due to the fact that

$$\sum_{p=k}^{q} \binom{p-k+j}{p-k} = \binom{q-k+1+j}{q-k},$$

  we have

$$\begin{aligned} &\Pr[E_i \ge q \mid E_0 = k-1] \\ \le& \frac{1}{2^{q-k+1}} + \sum_{j=0}^{i-2} \frac{1}{2^{q-k+2+j}} \binom{q-k+1+j}{q-k} \\ =& \frac{1}{2^{q-k+1}} + \sum_{j=1}^{i-1} \frac{1}{2^{q-k+1+j}} \binom{q-k+j}{q-k} \\ =& \binom{q-k+0}{q-k} \cdot \frac{1}{2^{q-k+1}} + \sum_{j=1}^{i-1} \frac{1}{2^{q-k+1+j}} \binom{q-k+j}{q-k} \\ =& \sum_{j=0}^{i-1} \frac{1}{2^{q-k+1+j}} \binom{q-k+j}{q-k}. \end{aligned}$$

  This indicates that Inequality 9 holds for any round $i \in [T]$.

Note that $\mathcal{A}_g$ wins $\mathcal{G}ame(k, s, T)$ if and only if in the last round T, $E_T = k+s$. Given Inequality 9, we set $q = k+s$, $i = T$, and get

$$\Pr[E_T \ge k+s \mid E_0 = k-1] \le \sum_{j=0}^{T-1} \binom{s+j}{s} \cdot \frac{1}{2^{s+1+j}}. \tag{11}$$

By definition, we have $0 \leq E_T \leq k{+}s$, and thus $\mathsf{Pr}[E_T \geq k{+}s \mid E_0 = k-1] = \mathsf{Pr}[E_T = k+s \mid E_0 = k-1]$. Therefore we obtain

$$\mathsf{Pr}[E_T = k+s \mid E_0 = k-1] \leq \sum_{j=0}^{T-1} \binom{s+j}{s} \cdot \frac{1}{2^{s+1+j}}, \tag{12}$$

which is exactly the upper bound of the winning probability of $\mathcal{A}_g$ claimed in Lemma 5.1. □

Tightness of Lemma 5.1. We note that the bound we obtained in Lemma 5.1 can be met by the following adversary $\mathcal{A}_g$: In the i-th iteration, $\mathcal{A}_g$ sets $e_i = 2^{E_{i-1}}$ which satisfies that $\mathsf{Po2}(e_i) \leq E_{i-1}$, and sets $c_i = 0$. One can verify that such an adversary indeed matches the bound in Lemma 5.1.

G.3 Proof of Lemma 5.2

Proof. Let $\mathcal{S}$ be the ideal world adversary and $\mathcal{A}$ the real world adversary controlling $t = \frac{n-1}{2}$ corrupted parties. $\mathcal{S}$ is invoked by receiving the prover's identity j from $\mathcal{F}_{\text{VrfyIP}}$. We consider the following two cases.

Case 1: the prover P_j is corrupted. In this case, $\mathcal{S}$ also receives the shares of the corrupted parties and the whole sharings $\{(\llbracket \boldsymbol{\mu}_i \rrbracket_k, \llbracket \boldsymbol{\nu}_i \rrbracket_k, \llbracket w_i \rrbracket_k)\}_{i=1}^{p}$ of the corrupted prover from $\mathcal{F}_{\text{VrfyIP}}$. $\mathcal{S}$ works as follows.

(1) $\mathcal{S}$ emulates $\mathcal{F}_{\text{input}}$ and receives the shares of $h_i/2^k$ the honest parties should hold from the corrupted prover P_j for each $i \in [p]$. $\mathcal{S}$ locally multiplies the shares by 2^k over $\mathbb{Z}_{2^{k+s}}$ and obtains $\llbracket h_i \rrbracket_{k+s}$ held by the honest parties.
(2) $\mathcal{S}$ plays the role of $\mathcal{F}_{\text{coin}}$ by sampling and handing random $\theta_1, \cdots, \theta_p \in \mathbb{Z}_{2^{k+s}}$ to $\mathcal{A}$.
(3) As $\mathcal{S}$ is given the corrupted parties' shares and the whole sharings $\{(\llbracket \boldsymbol{\mu}_i \rrbracket_k, \llbracket \boldsymbol{\nu}_i \rrbracket_k, \llbracket w_i \rrbracket_k)\}_{i=1}^{p}$ of the prover, it computes the honest parties' shares of these input triples. Then it deduces the sharings $(\llbracket \boldsymbol{x} \rrbracket_{k+s}, \llbracket \boldsymbol{y} \rrbracket_{k+s}, \llbracket z \rrbracket_{k+s})$ held by the honest parties using the randomness θ_i as described in the protocol.
(4) $\mathcal{S}$ repeats the following procedure. It emulates $\mathcal{F}_{\text{input}}$ and receives the shares of $z_{i,i'}$ the honest parties should have from $\mathcal{A}$ for all $i, i' \in [q]$ and $(i, i') \neq (1,1)$ from $\mathcal{A}$. Then it simulates $\mathcal{F}_{\text{coin}}$ by sending random $\{\alpha_i\}_{i=1}^{q}, \{\beta_i\}_{i=1}^{q}$ in $\mathbb{Z}_{2^{k+s}}$ to $\mathcal{A}$, and computes $(\llbracket \boldsymbol{x}' \rrbracket_{k+s}, \llbracket \boldsymbol{y}' \rrbracket_{k+s}, \llbracket z' \rrbracket_{k+s})$ using the randomness α_i, β_i to update the sharings $(\llbracket \boldsymbol{x} \rrbracket_{k+s}, \llbracket \boldsymbol{y} \rrbracket_{k+s}, \llbracket z \rrbracket_{k+s})$ held by the honest parties. $\mathcal{S}$ then goes to the next iteration until the dimension of the vectors is at most q.
(5) In the final check step, $\mathcal{S}$ emulates $\mathcal{F}_{\text{input}}$ and receives the shares of x_0, y_0 and then the shares of $z_{i,i'}$ for all $i, i' \in [q]$ and $(i, i') \neq (1,1)$ the honest parties should hold from $\mathcal{A}$. Then it simulates $\mathcal{F}_{\text{coin}}$ by handing random $\{\alpha_i\}_{i=1}^{q}, \{\beta_i\}_{i=1}^{q}$ in $\mathbb{Z}_{2^{k+s}}$ to $\mathcal{A}$.
(6) $\mathcal{S}$ computes the sharings of the final multiplication triple $(\llbracket x' \rrbracket_{k+s}, \llbracket y' \rrbracket_{k+s}, \llbracket z' \rrbracket_{k+s})$ held by the honest parties, and then reconstruct the triple (x', y', z') from the computed shares. Then $\mathcal{S}$ simulates the $\mathsf{reconstruct}$ procedure by handing the honest parties' shares of the multiplication triple to each corrupted party controlled by $\mathcal{A}$.
(7) If $\mathcal{F}_{\text{VrfyIP}}$ doesn't send abort to $\mathcal{S}$, then $\mathcal{S}$ sends accept to $\mathcal{F}_{\text{VrfyIP}}$ if $(\llbracket x' \rrbracket_{k+s}, \llbracket y' \rrbracket_{k+s}, \llbracket z' \rrbracket_{k+s})$ is a correct multiplication triple, and sends abort otherwise. Otherwise if $\mathcal{S}$ receives abort from $\mathcal{F}_{\text{VrfyIP}}$ and the final multiplication triple is incorrect, i.e., $x' \cdot y' \neq_{k+s} z'$, then $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs. Otherwise, if it receives abort from $\mathcal{F}_{\text{VrfyIP}}$ but the final multiplication triple is correct, then $\mathcal{S}$ outputs fail and halts.

Observe that $\mathcal{S}$ knows exactly the honest parties' shares in this case, and thus the above simulation is perfect. The only difference between the simulation and the real execution is the event that $\mathcal{S}$ outputs fail, where $\mathcal{F}_{\text{VrfyIP}}$ outputs abort to the honest parties but the honest parties in the real execution output accept. Note that this happens only when there exists some incorrect triple $c_i \neq_k \boldsymbol{a}_i \cdot \boldsymbol{b}_i$ for some $i \in [m]$ but the final multiplication triple is correct, i.e., $x' \cdot y' \equiv_{k+s} z'$. We now show the probability of this event is negligible in λ given $s = \max(3T, \lambda + T(1/2 + \log(5/2 + 3\lambda/T)))$ where $T = 2\lceil \log_q(p \cdot d) \rceil + 1$ under the assumption that $T \leq \lambda$ and $3T < s$.

Given $\mathcal{G}\mathit{ame}(k, s, T)$ in Section 5.1, we first have the following claim.

Claim G.3.1. *If $\mathcal{A}$ can cause $\mathcal{S}$ to output* fail *with probability q, then $\mathcal{S}$ can win the game with the same probability q.*

Proof. We use the definitions and notations in Section 5.1. Particularly, we denote the additive errors of the inner-product triples after lifting to $\mathbb{Z}_{2^{k+s}}$ by $\epsilon_1, \ldots, \epsilon_p$. Recall that the event $\mathcal{A}$ causes $\mathcal{S}$ to output fail implies at least of one of the input inner-product triples is incorrect but the final multiplication triple is correct. Thus here we can assume that there exists an index $i^\star \in [p]$ s.t. the $i^\star$-th inner-product triple has an non-zero additive error $\epsilon_{i^\star}$ over $\mathbb{Z}_{2^k}$, i.e., $\mathsf{Po2}(\epsilon_{i^\star}) \leq E_0 = k-1$. We let $\mathcal{S}$ work as follows.

- $\mathcal{S}$ initially has $E_0 = k-1$.
- $\mathcal{S}$ works as we described above until simulating $\mathcal{F}_{\text{coin}}$ in Step 2. It now picks $p-1$ random coefficients θ_i for $i \in [p], i \neq i^\star$, sets $e_1 = \epsilon_{i^\star}$, $c_1 = \sum_{i \in [p], i \neq i^\star} \theta_i \cdot \epsilon_i$, and sends (e_1, c_1) to $\mathcal{C}_g$ in the first round of the game. Then it receives a random $r_1 \in \mathbb{Z}_{2^{k+s}}$ from $\mathcal{C}_g$ and defines $\theta_{i^\star} = r_1$. Now $\mathcal{S}$ simulates $\mathcal{F}_{\text{coin}}$ sending these p random values $\{\theta_i\}_{i \in [p]}$ to the adversary $\mathcal{A}$. Additionally, $\mathcal{S}$ defines and computes $\epsilon = r_1 \cdot e_1 + c_1$. Clearly $\epsilon = \theta_{i^\star} \cdot \epsilon_{i^\star} + \sum_{i \in [p], i \neq i^\star} \theta_i \cdot \epsilon_i = \sum_{i \in [p]} \theta_i \cdot \epsilon_i = z - \boldsymbol{x} \cdot \boldsymbol{y}$, which is exactly the additive error on z. Then $\mathcal{S}$ calculates $E_1 = \mathsf{Po2}(\epsilon)$.
- $\mathcal{S}$ proceeds as described above until simulating $\mathcal{F}_{\text{coin}}$ in the first iteration of the repetition in Step 4. It now works as follows.
  - $\mathcal{S}$ defines and computes each $\epsilon_{i,i'} = z_{i,i'} - \boldsymbol{x}_i \cdot \boldsymbol{y}_{i'}$ for $i, i' \in [p]$ (as it knows the honest parties' shares of $\boldsymbol{x}_i, \boldsymbol{y}_i, z_{i,i'}$ and can reconstruct them). Now we have $\epsilon = \sum_{i=1}^{p} \epsilon_{i,i}$, and it follows that there exists an index $i^\star$ s.t. $\mathsf{Po2}(\epsilon_{i^\star,i^\star}) \leq \mathsf{Po2}(\epsilon) = E_1$. $\mathcal{S}$ now picks $p-1$ randomnesses α_i for $i \in [p], i \neq i^\star$, sets $e_2 = \epsilon_{i^\star,i^\star}$, $c_2 = \sum_{i \in [p], i \neq i^\star} \alpha_i \epsilon_{i,i^\star}$ and sends (e_2, c_2) to $\mathcal{C}_g$ in the second round of the game. After $\mathcal{S}$ receives a random r_2 from $\mathcal{C}_g$, it defines $\alpha_{i^\star} = r_2$.
  - At this point, $\mathcal{S}$ defines and computes $\epsilon'_{i'} = \sum_{i \in [p]} \alpha_i \cdot \epsilon_{i,i'}$ for each $i \in [p]$. Then $\mathcal{S}$ picks another $p-1$ randomnesses

β_i for $i \in [p], i \neq i^\star$, sets $e_3 = \epsilon'_{i^\star}$, $c_3 = \sum_{i' \in [p], i' \neq i^\star} \beta_{i'} \cdot \epsilon'_{i'}$ and sends (e_3, c_3) to $\mathcal{C}_g$ in the third round of the game. $\mathcal{S}$ defines the received randomness r_3 as $\beta_{i^\star}$, now simulates $\mathcal{F}_{\text{coin}}$ handing these random values $\{\alpha_i\}_{i=1}^p, \{\beta_i\}_{i=1}^p$ to $\mathcal{A}$. Now $\mathcal{S}$ defines $\epsilon' = r_3 \cdot e_3 + c_3$. Clearly we have $\epsilon' = \beta_{i^\star} \cdot \epsilon'_{i^\star} + \sum_{i' \in [p], i' \neq i^\star} \beta_{i'} \cdot \epsilon'_{i'}) = \sum_{i' \in [p]} \beta_{i'} \cdot (\sum_{i=1}^q \alpha_i \cdot \epsilon_{i,i'}) = z' - \boldsymbol{x}' \cdot \boldsymbol{y}$. And then $\mathcal{S}$ computes $E_3 = \mathsf{Po2}(\epsilon')$.

$\mathcal{S}$ goes on simulating the honest parties as we described previously. In each iteration of the repetition in Step 4, it simulates $\mathcal{F}_{\text{coin}}$ in the same way above.

- In the final check step, the number of iterations v reaches $v = \lceil \log_q(p \cdot d) \rceil$. $\mathcal{S}$ simulates $\mathcal{F}_{\text{coin}}$ by picking random values $\{\alpha_i\}_{i \in [p], i \neq i^\star}, \{\beta_i\}_{i \in [p], i \neq i^\star}$, sending (e_{2v}, c_{2v}),(e_{2v+1}, c_{2v+1}) (defined in a similar way to that in the previous iterations) to $\mathcal{C}_g$ in two rounds, and taking the received randomnesses r_{2v}, r_{2_v+1} as $\alpha_{i^\star}, \beta_{i^\star}$. Note that now the number of interaction rounds in this game is $T = 2v + 1 = 2\lceil \log_q(p \cdot d) \rceil + 1$. $\mathcal{S}$ additionally defines $\alpha_0 = \beta_0 = 1$, and computes the final additive error $\epsilon' = r_{2v+1} \cdot e_{2v+1} + c_{2v+1} = \sum_{i' \in [0,p]} \beta_{i'} \cdot (\sum_{i' \in [0,p]} \alpha_i \cdot \epsilon_{i,i'}) = z' - x' \cdot y'$. $\mathcal{S}$ proceeds to simulate the following as described above until the simulation ends.

As we analyzed in Section 5.1, in each round the value r_i for $i \in [T]$ received from $\mathcal{C}_g$ is uniformly random, and thus the simulation of $\mathcal{F}_{\text{coin}}$ is perfect. Observe that at the end of the simulation, we have $E_T = \mathsf{Po2}(r_{2v+1} \cdot e_{2v+1} + c_{2v+1}) = \mathsf{Po2}(\epsilon')$. The event that $\mathcal{A}$ causes $\mathcal{S}$ to output fail indicates that $z' - x' \cdot y' \equiv_{k+s} 0$, i.e., $\mathsf{Po2}(\epsilon') = k+s$, which further means that $E_T = k + s$ and $\mathcal{S}$ now wins the game. Therefore, if $\mathcal{A}$ can make $\mathcal{S}$ fail with probability q, then with the same probability, $\mathcal{S}$ can win the game $\mathcal{G}ame(k, s, T)$. □

According to Lemma 5.1, we know that the probability for an adversary $\mathcal{A}_g$ winning the game $\mathcal{G}ame(k, s, T)$ is at most $\sum_{j=0}^{T-1} \binom{s+j}{s} \cdot \frac{1}{2^{s+1+j}}$. In our case, $T = 2\lceil \log_q(p \cdot d) \rceil + 1$. Then based on our analysis in Appendix G.6.1, when we set $s = \lambda + T(1/2 + \log(5/2 + 3\lambda/T))$ (assuming that $T \leq \lambda$ and $3T \leq s$) the winning probability of $\mathcal{A}_g$ is at most $2^{-\lambda}$. Given Claim G.3.1, we deduce that the probability of $\mathcal{S}$ outputting fail is also bounded by $2^{-\lambda}$, which is exactly the soundness error claimed in this lemma.

Case 2: the prover P_j is honest. In this case, $\mathcal{S}$ receives the corrupted parties' shares $\{(\llbracket \boldsymbol{\mu}_i \rrbracket_k, \llbracket \boldsymbol{\nu}_i \rrbracket_k, \llbracket w_i \rrbracket_k)\}_{i=1}^p$. It works as follows.

(1) $\mathcal{S}$ simulates $\mathcal{F}_{\text{input}}$ and receives the shares of $\llbracket h_i/2^k \rrbracket_s$ of corrupted parties.

(2) $\mathcal{S}$ plays the role of $\mathcal{F}_{\text{coin}}$ by handing random $\theta_1, \cdots, \theta_p \in \mathbb{Z}_{2^{k+s}}$ to $\mathcal{A}$.

(3) $\mathcal{S}$ repeats the following procedure. $\mathcal{S}$ simulates $\mathcal{F}_{\text{input}}$ and receives the shares of $\llbracket z_{i,i'} \rrbracket_{k+s}$ of corrupted parties for each $i, i' \in [q]$ and $(i, i') \neq (1, 1)$. Then it computes the shares of $\llbracket z_{1,1} \rrbracket_{k+s}$ that corrupted parties should hold. Next it simulates $\mathcal{F}_{\text{coin}}$ sending random $\{\alpha_i\}_{i=1}^q, \{\beta_i\}_{i=1}^q$ to $\mathcal{A}$. Finally, $\mathcal{S}$ follows the protocol and computes the shares $(\llbracket \boldsymbol{x} \rrbracket_{k+s}, \llbracket \boldsymbol{y} \rrbracket_{k+s}, \llbracket \boldsymbol{z} \rrbracket_{k+s}$ that corrupted parties should hold. $\mathcal{S}$ goes to the next iteration until the dimension of the vectors is at most q.

(4) $\mathcal{S}$ simulates $\mathcal{F}_{\text{input}}$ and receives corrupted parties' shares of $\llbracket x_0 \rrbracket_{k+s}, \llbracket y_0 \rrbracket_{k+s}$ and $\llbracket z_{i,i'} \rrbracket_{k+s}$ for all $i, i' \in [0, q]$ and $(i, i') \neq (1, 1)$. Then it computes the shares of $\llbracket z_{1,1} \rrbracket_{k+s}$ that corrupted parties should hold. $\mathcal{S}$ simulates $\mathcal{F}_{\text{coin}}$ handing random $\{\alpha_i\}_{i=1}^q, \{\beta_i\}_{i=1}^q$ to $\mathcal{A}$. Now $\mathcal{S}$ follows the protocol and computes the shares of $(\llbracket x' \rrbracket_{k+s}, \llbracket y' \rrbracket_{k+s}, \llbracket z' \rrbracket_{k+s})$ that corrupted parties should hold. Then $\mathcal{S}$ picks a random triple (x', y', z') over $\mathbb{Z}_{2^{k+s}}$ s.t. $x' \cdot y' \equiv_{k+s} z'$. Since there are exactly $t = (n-1)/2$ corrupted parties, the shares of honest parties are fully determined by the secret and the shares of corrupted parties. Thus, $\mathcal{S}$ calculates the honest parties' shares of $(\llbracket x' \rrbracket_{k+s}, \llbracket y' \rrbracket_{k+s}, \llbracket z' \rrbracket_{k+s})$. Finally, it simulates the $\mathsf{reconstruct}$ procedure by sending the honest parties' shares to the corrupted parties honestly. Also, it receives the corrupted parties' shares of $(\llbracket x' \rrbracket_{k+s}, \llbracket y' \rrbracket_{k+s}, \llbracket z' \rrbracket_{k+s})$ from $\mathcal{A}$.
$\mathcal{S}$ follows the rest of the protocol to check the triple $(\llbracket x' \rrbracket_{k+s}, \llbracket y' \rrbracket_{k+s}, \llbracket z' \rrbracket_{k+s})$. If the received shares are inconsistent, it sends abort to $\mathcal{F}_{\text{VrfyIP}}$; otherwise it sends accept to $\mathcal{F}_{\text{VrfyIP}}$. Then $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs.

Observe that the adversary's view consists of (1) shares of $\llbracket h_i/2^k \rrbracket_s$ for $i \in [p]$, shares of $\llbracket z_{i,i'} \rrbracket_{k+s}$ for $i, i' \in [q]$ (or $[0, q]$) but $(i, i') \neq (1, 1)$, shares of $\llbracket x_0 \rrbracket_{k+s}, \llbracket y_0 \rrbracket_{k+s}$ from the prover, (2) the revealed triple (x', y', z'). Due to the secrecy of the RSS scheme, view (1) is the same in the simulation and the real execution. And as the prover is honest, the final triple in view (2) is always a random triple under the condition that $x' \cdot y' \equiv_{k+s} z'$, and thus view (2) is also the same in both executions. This concludes our proof. □

G.4 Proof of Lemma 3.1

Proof. Let $\mathcal{S}$ be the ideal world adversary and $\mathcal{A}$ the real world adversary controlling $t = \frac{n-1}{2}$ corrupted parties. Let $\mathcal{C}$ denote the set of corrupted parties, and $\mathcal{H}$ denotes the set of honest parties.

Construction of the Ideal World Adversary $\mathcal{S}$. In the beginning, $\mathcal{S}$ receives from $\mathcal{F}_{\text{VrfySSIP}}$ the shares of $(\llbracket \boldsymbol{a}_i \rrbracket_k, \llbracket \boldsymbol{b}_i \rrbracket_k, \llbracket c_i \rrbracket_k)$ of the corrupted parties and the additive errors $\epsilon_i \equiv_k c_i - \boldsymbol{a}_i \cdot \boldsymbol{b}_i$ for $i \in [m]$. Then $\mathcal{S}$ simulates the behaviors of honest parties and interacts with the real world adversary $\mathcal{A}$ as follows.

(1) In Step 2, $\mathcal{S}$ faithfully emulates the role of $\mathcal{F}_{\text{coin}}$ by choosing a random seed of size σ and handing the seed to all parties. $\mathcal{S}$ expands the seed to obtain random binary coefficients $\boldsymbol{\gamma}_1, \cdots, \boldsymbol{\gamma}_\lambda \in \{0, 1\}^m$. Then $\mathcal{S}$ follows the protocol in Step 3 and computes the shares of $(\llbracket \boldsymbol{a}'_i \rrbracket_k, \llbracket \boldsymbol{b}'_i \rrbracket_k, \llbracket c'_i \rrbracket_k)$ of corrupted parties for all $i \in [\lambda]$.

(2) In Step 4, $\mathcal{S}$ emulates $\mathcal{F}_{\text{input}}$ as follows.
    - For each corrupted party P_j, $\mathcal{S}$ receives the input ${c'}_i^{(j)}$ and the whole sharing $\llbracket {c'}_i^{(j)} \rrbracket_k$ for all $i \in [\lambda]$.
    - For each honest party P_j, $\mathcal{S}$ receives the shares of $\llbracket {c'}_i^{(j)} \rrbracket_k$ of corrupted parties for all $i \in [\lambda]$.

(3) In Step 5, $\mathcal{S}$ emulates $\mathcal{F}_{\text{VrfyIP}}$ as follows. For each corrupted party P_j, since $\mathcal{S}$ learns the shares of $(\llbracket \boldsymbol{a}'_i \rrbracket_k, \llbracket \boldsymbol{b}'_i \rrbracket_k, \llbracket c'_i \rrbracket_k)$ P_j should hold and the whole sharing $\llbracket {c'}_i^{(j)} \rrbracket_k$ for all $i \in [\lambda]$, $\mathcal{S}$ follows this step and computes the whole sharings $(\llbracket \boldsymbol{\mu}_i^{(j)} \rrbracket_k, \llbracket \boldsymbol{\nu}_i^{(j)} \rrbracket_k, \llbracket {c'}_i^{(j)} \rrbracket_k)$ for all $i \in [\lambda]$. Then $\mathcal{S}$ sends

the whole sharings to the ideal adversary of $\mathcal{F}_{\mathrm{VrfyIP}}$ and honestly emulates the Step 3 in $\mathcal{F}_{\mathrm{VrfyIP}}$.
For each honest party P_j, since $\mathcal{S}$ learns the shares of ($\llbracket \boldsymbol{a}'_i \rrbracket_k$, $\llbracket \boldsymbol{b}'_i \rrbracket_k$, $\llbracket c'_i \rrbracket_k$) and $\llbracket {c'}_i^{(j)} \rrbracket_k$ that corrupted parties should hold for all $i \in [\lambda]$, $\mathcal{S}$ follows this step and computes the shares of ($\llbracket \boldsymbol{\mu}_i^{(j)} \rrbracket_k$, $\llbracket \boldsymbol{\nu}_i^{(j)} \rrbracket_k$, $\llbracket {c'}_i^{(j)} \rrbracket_k$) of corrupted parties for all $i \in [\lambda]$. Then $\mathcal{S}$ sends these shares to the ideal adversary of $\mathcal{F}_{\mathrm{VrfyIP}}$. In Step 3 of $\mathcal{F}_{\mathrm{VrfyIP}}$, $\mathcal{S}$ assumes $c_i'^{(j)} \equiv_k \boldsymbol{\mu}_i^{(j)} \cdot \boldsymbol{\nu}_i^{(j)}$ for all $i \in [\lambda]$ and follows the rest of this step.

(4) In Step 6, for all $i \in [\lambda]$ $\mathcal{S}$ computes ${c'}_i^{(j)}$ for each corrupted party j by using the shares of $\llbracket \boldsymbol{a}'_i \rrbracket_k$, $\llbracket \boldsymbol{b}'_i \rrbracket_k$ that P_j should hold. Let $\tilde{c}_i^{(j)}$ denote the secret $\mathcal{S}$ received when emulating $\mathcal{F}_{\mathrm{input}}$ in Step 4. Then $\mathcal{S}$ computes $o_i = \gamma_{i,1} \cdot \epsilon_1 + \ldots + \gamma_{i,m} \cdot \epsilon_m + \sum_{j \in \mathcal{C}} ({c'}_i^{(j)} - \tilde{c}_i^{(j)})$. Note that $\gamma_{i,1} \cdot \epsilon_1 + \ldots + \gamma_{i,m} \cdot \epsilon_m$ is the additive error if all corrupted parties share $\{{c'}_i^{(j)}\}_{j \in \mathcal{C}}$ correctly, and $\sum_{j \in \mathcal{C}} ({c'}_i^{(j)} - \tilde{c}_i^{(j)})$ is the additive error due to the possibly incorrect $\{\tilde{c}_i^{(j)}\}_{j \in \mathcal{C}}$.
Then, $\mathcal{S}$ computes the shares of each $\llbracket o_i \rrbracket_k$ that corrupted parties should hold. From o_i and the shares of corrupted parties, $\mathcal{S}$ computes the shares of honest parties. Finally, $\mathcal{S}$ honestly follows the rest of this step.

(5) In Step 7, if $\mathcal{S}$ receives `accept` from $\mathcal{F}_{\mathrm{VrfySSIP}}$,
- If $\mathcal{F}_{\mathrm{VrfyIP}}$ returns `reject` or there exists some $o_i \equiv_k 0$, $\mathcal{S}$ aborts on behalf of honest parties and sends `reject` to $\mathcal{F}_{\mathrm{VrfySSIP}}$.
- Otherwise, $\mathcal{S}$ sends `accept` to $\mathcal{F}_{\mathrm{VrfySSIP}}$.

Otherwise, $\mathcal{S}$ aborts on behalf of honest parties no matter the result of $\mathcal{F}_{\mathrm{VrfyIP}}$ or whether $o_i \equiv_k 0$ for some i.

Hybrid Arguments. Consider the following hybrids.

$\mathbf{Hybrid}_0$: In this hybrid, $\mathcal{S}$ uses honest parties input and honestly emulates honest parties. This corresponds to the real world.

$\mathbf{Hybrid}_1$: In this hybrid, $\mathcal{S}$ computes the shares of corrupted parties and the sharings dealt by corrupted parties as described above. Then $\mathcal{S}$ emulates $\mathcal{F}_{\mathrm{VrfyIP}}$ in Step 5 as described above. Note that the shares computed by $\mathcal{S}$ are always consistent with the shares held by honest parties. Thus, the distribution of $\mathbf{Hybrid}_1$ is identically to the distribution of $\mathbf{Hybrid}_0$.

$\mathbf{Hybrid}_2$: In this hybrid, $\mathcal{S}$ simulates Step 2 as described above. Then $\mathcal{S}$ simulates the last step as described above. The only difference between $\mathbf{Hybrid}_2$ and $\mathbf{Hybrid}_1$ is that $\mathcal{S}$ will abort on behalf of honest parties if receiving `reject` from $\mathcal{F}_{\mathrm{VrfySSIP}}$ even if $\mathcal{F}_{\mathrm{VrfyIP}}$ returns `accept` and $o_i \equiv_k 0$ for all $i \in [\lambda]$. We argue that this happens with negligible probability in λ and σ assuming that G is a PRG.

First note that, when $\boldsymbol{\gamma}_i \in \{0,1\}^m$ are sampled from uniform distribution, and there exists $j \in [m]$ s.t. $c_j \not\equiv_k \boldsymbol{a}_j \cdot \boldsymbol{b}_j$, then with probability $1/2$, $c'_i \neq \boldsymbol{a}'_i \cdot \boldsymbol{b}'_i$. Thus, when $\boldsymbol{\gamma}_1, \cdots, \boldsymbol{\gamma}_\lambda$ are all sampled from uniform distribution, with all but negligible probability in λ, there exists $i \in [\lambda]$ s.t. $c'_i \neq \boldsymbol{a}'_i \cdot \boldsymbol{b}'_i$. Then either $\mathcal{F}_{\mathrm{VrfyIP}}$ returns `reject` in case some corrupted party P_j does not share correct ${c'}_i^{(j)}$, or $o_i \neq 0$ in case all corrupted parties share correct $\{{c'}_i^{(j)}\}_{j \in \mathcal{C}}$. Thus, the event that $\mathcal{F}_{\mathrm{VrfySSIP}}$ returns `reject` but $\mathcal{F}_{\mathrm{VrfyIP}}$ returns `accept` and $o_i \equiv_k 0$ for all $i \in [\lambda]$ happens with negligible probability in λ.

Now consider the scenario where $\boldsymbol{\gamma}_1, \cdots, \boldsymbol{\gamma}_\lambda$ are expanded by G from a uniform seed. Since G is a PRG, the output distribution is computationally indistinguishable from the case where $\boldsymbol{\gamma}_1, \cdots, \boldsymbol{\gamma}_\lambda$ are sampled from uniform distribution (Otherwise, we may use the above procedure to detect whether $\boldsymbol{\gamma}_1, \cdots, \boldsymbol{\gamma}_\lambda$ is from uniform distribution or generated by G).

Thus, the output of $\mathbf{Hybrid}_2$ is computationally indistinguishable from $\mathbf{Hybrid}_1$.

$\mathbf{Hybrid}_3$: In this hybrid, $\mathcal{S}$ computes o_i by using $\{\epsilon_j\}_{j=1}^m$ as described in Step 6 above and then computes and uses the shares of $\llbracket o_i \rrbracket_k$ of honest parties as described in Step 6. Note that we may write

$$\begin{aligned} o_i &= \gamma_{i,1} \cdot c_1 + \ldots + \gamma_{i,m} \cdot c_m - \sum_{j=1}^{n} \tilde{c}_i^{(j)} \\ &= \gamma_{i,1} \cdot (c_1 - \boldsymbol{a}_1 \cdot \boldsymbol{b}_1) + \ldots + \gamma_{i,m} \cdot (c_m - \boldsymbol{a}_m \cdot \boldsymbol{b}_m) \\ &\quad + \sum_{j=1}^{n} {c'}_i^{(j)} - \sum_{j=1}^{n} \tilde{c}_i^{(j)}, \end{aligned}$$

where we use ${c'}_i^{(j)}$ to denote the correct share P_j should compute from the j-th shares of $\llbracket \boldsymbol{a}'_i \rrbracket_k$, $\llbracket \boldsymbol{b}'_i \rrbracket_k$ and $\tilde{c}_i^{(j)}$ to denote the share P_j shares in Step 4. Note that we always have $\sum_{j=1}^n {c'}_i^{(j)} = \boldsymbol{a}'_i \cdot \boldsymbol{b}'_i = \gamma_{i,1} \cdot \boldsymbol{a}_1 \cdot \boldsymbol{b}_1 + \ldots + \gamma_{i,m} \cdot \boldsymbol{a}_m \cdot \boldsymbol{b}_m$. And for each honest party P_j, ${c'}_i^{(j)} = \tilde{c}_i^{(j)}$. Thus

$$o_i = \gamma_{i,1} \cdot \epsilon_1 + \cdots + \gamma_{i,m} \cdot \epsilon_m + \sum_{j \in \mathcal{C}} ({c'}_i^{(j)} - \tilde{c}_i^{(j)}),$$

which is exactly the one computed above.

Therefore, o_i computed in Step 6 is identical to that in $\mathbf{Hybrid}_2$, which means that o_i is consistent with the shares of $\llbracket o_i \rrbracket_k$ held by honest parties, and the shares of $\llbracket o_i \rrbracket_k$ that corrupted parties should hold. Note that when there are exactly $t = (n-1)/2$ corrupted parties, the shares of honest parties are fully determined by the shares of corrupted parties and the secret. Thus, the output distribution of $\mathbf{Hybrid}_3$ is identical to that of $\mathbf{Hybrid}_2$.

$\mathbf{Hybrid}_4$: In this hybrid, $\mathcal{S}$ simulates $\mathcal{F}_{\mathrm{input}}$ as described above. The distribution remains the same as $\mathbf{Hybrid}_3$. Note that in this hybrid, $\mathcal{S}$ does not need to use the shares of honest parties, but only the values received from $\mathcal{F}_{\mathrm{VrfySSIP}}$. Thus $\mathbf{Hybrid}_4$ corresponds to the ideal world. And we conclude that Π_{VrfySSIP} securely computes $\mathcal{F}_{\mathrm{VrfySSIP}}$ with negligible error in λ and σ.

□

G.5 Soundness Analysis of the Optimized Protocol $\Pi_{\mathrm{VrfyIP}}^{\mathrm{Opt}}$

Following a similar argument to Lemma 5.2, the soundness of the optimized protocol $\Pi_{\mathrm{VrfyIP}}^{\mathrm{Opt}}$ is the same as the winning probability of the following game $\mathcal{G}ame'(k, s, T)$.

(1) $\mathcal{A}_g, \mathcal{C}_g$ initially have $E_0 = k - 1$.
(2) In the first round,
   (a) $\mathcal{A}_g$ chooses arbitrary $e_1, c_1 \in \mathbb{Z}_{2^{k+s}}$ under the requirement that $\mathsf{Po2}(e_1) \leq E_0$, and sends the two values to $\mathcal{C}_g$.

(b) $\mathcal{C}_g$ picks a uniformly random value $r_1 \in \mathbb{Z}_{2^{k+s}}$ and responds r_1 to $\mathcal{A}_g$.
(c) $\mathcal{A}_g$ and $\mathcal{C}_g$ compute $E_1 = \mathsf{Po2}(r_1 \cdot e_1 + c_1)$ and set $E_1' = E_1$.

(3) In Round i where $2 \leq i \leq T$, $\mathcal{A}_g$ and $\mathcal{C}_g$ repeat the following:
(a) $\mathcal{A}_g$ chooses arbitrary $e_i, c_i \in \mathbb{Z}_{2^{k+s}}$ under the requirement that $\mathsf{Po2}(e_i) \leq E_{i-1}$ and chooses arbitrary $e_i', c_i' \in \mathbb{Z}_{2^{k+s}}$ under the requirement that $\mathsf{Po2}(e_i') \leq E_{i-1}'$. Then $\mathcal{A}_g$ sends (e_i, c_i) and (e_i', c_i') to $\mathcal{C}_g$.
(b) $\mathcal{C}_g$ picks uniformly random values $r_i, r_i' \in \mathbb{Z}_{2^{k+s}}$ and responds r_i, r_i' to $\mathcal{A}_g$.
(c) $\mathcal{A}_g$ and $\mathcal{C}_g$ compute $E_i = \mathsf{Po2}(r_i \cdot e_i + c_i)$ and $E_i' = \mathsf{Po2}(r_i' \cdot e_i' + c_i')$.

(4) $\mathcal{A}_g$ wins if and only if in the last round T, $E_T = E_T' = k + s$.

We show the following lemma about $\mathcal{G}ame'(k, s, T)$.

Lemma G.3. *Let k, s, T be positive integers. For any adversary $\mathcal{A}_g$, the probability that $\mathcal{A}_g$ wins $\mathcal{G}ame'(k, s, T)$ is at most*

$$\sum_{i=0}^{s} \frac{1}{2^{i+1}} \left(\sum_{j=0}^{T-2} \binom{s-i+j}{s-i} \cdot \frac{1}{2^{s-i+1+j}} \right)^2 + \frac{1}{2^{s+1}}.$$

We now give the proof of Lemma G.3.

Proof. Consider a fixed adversary $\mathcal{A}_g$ with a fixed random tape. We first have the following claim.

Proposition G.4. *For a fixed adversary $\mathcal{A}_g$ with a fixed random tape and for all u, the variables $E_i|_{E_1=u}$ and $E_i'|_{E_1=u}$ are independent for all $i \in [2, T]$.*

Proof. Recall that at the end of the first round of $\mathcal{G}ame'(k, s, T)$, we have $E_1 = E_1'$. Starting from the second round, in each round $i \in [2, T]$ the adversary $\mathcal{A}_g$ chooses two pairs of values (e_i, c_i) and (e_i', c_i') under the requirements $\mathsf{Po2}(e_i) \leq E_{i-1}$ and $\mathsf{Po2}(e_i') \leq E_{i-1}'$ respectively, and sends them to the challenger $\mathcal{C}_g$. Then $\mathcal{C}_g$ replies two random values r_i, r_i' chosen independently. Finally $E_i = \mathsf{Po2}(e_i \cdot r_i + c_i)$ and $E_i' = \mathsf{Po2}(e_i' \cdot r_i' + c_i)$.

As we can see, given the adversary $\mathcal{A}_g$, its random tape, and given $E_1 = E_1' = u$, the random variable E_i only depends on the randomness $r_2, \ldots, r_i$, and the random variable E_i' only depends on the randomness $r_2', \ldots, r_i'$. Thus, $E_i|_{E_1=u}$ and $E_i'|_{E_1=u}$ are independent for all $i \in [2, T]$. □

Following Inequality 9, we have the following claim.

Proposition G.5. *For any positive integer $u \leq k + s - 1$,*

$$\begin{aligned} &\mathsf{Pr}[E_T \geq k + s \mid E_1 = u, E_0 = k - 1] \\ =&\mathsf{Pr}[E_T' \geq k + s \mid E_1 = u, E_0 = k - 1] \\ \leq& \sum_{j=0}^{T-2} \binom{k+s-u+j-1}{k+s-u-1} \cdot \frac{1}{2^{k+s-u+j}}. \end{aligned}$$

The proof is similar with that of Inequality 9.

Let Ω be the event that $\mathcal{A}_g$ wins $\mathcal{G}ame'(k, s, T)$. Applying the law of total probability over all possible values of E_1 and according to Proposition G.4, we have

$$\begin{aligned} \mathsf{Pr}[\Omega] =& \sum_{u=0}^{k+s} \mathsf{Pr}[E_T = k + s, E_T' = k + s \mid E_1 = u, E_0 = k - 1] \\ &\cdot \mathsf{Pr}[E_1 = u \mid E_0 = k - 1] \\ =& \sum_{u=0}^{k+s} \mathsf{Pr}[E_T = k + s \mid E_1 = u, E_0 = k - 1] \\ &\cdot \mathsf{Pr}[E_T' = k + s \mid E_1 = u, E_0 = k - 1] \\ &\cdot \mathsf{Pr}[E_1 = u \mid E_0 = k - 1]. \end{aligned}$$

As $0 \leq E_1 \leq k+s$, the event $E_1 \geq k+s$ is equivalent to $E_1 = k+s$. Thus we have

$$\begin{aligned} \mathsf{Pr}[\Omega] =& \sum_{u=0}^{k+s} \mathsf{Pr}[E_T \geq k + s \mid E_1 = u, E_0 = k - 1] \\ &\cdot \mathsf{Pr}[E_T' \geq k + s \mid E_1 = u, E_0 = k - 1] \\ &\cdot \mathsf{Pr}[E_1 = u \mid E_0 = k - 1] \\ =& \sum_{u=0}^{k+s-1} \mathsf{Pr}^2[E_T \geq k + s \mid E_1 = u, E_0 = k - 1] \\ &\cdot (\mathsf{Pr}[E_1 \geq u \mid E_0 = k - 1] - \mathsf{Pr}[E_1 \geq u + 1 \mid E_0 = k - 1]) \\ &+ \mathsf{Pr}^2[E_T \geq k + s \mid E_1 = k + s, E_0 = k - 1] \\ &\cdot \mathsf{Pr}[E_1 \geq k + s \mid E_0 = k - 1]. \end{aligned}$$

For all $T \geq 2$, let

$$p(k + s, u, T) = \sum_{j=0}^{T-2} \binom{k+s-u+j-1}{k+s-u-1} \cdot \frac{1}{2^{k+s-u+j}}$$

for all $u \in [0, k+s-1]$ and let $p(k+s, k+s, T) = 1$. By Proposition G.5 and by the fact that any probability is upper-bounded by 1, we have

$$\mathsf{Pr}^2[E_T \geq k + s \mid E_1 = u, E_0 = k - 1] \leq p^2(k + s, u, T)$$

for all $u \in [0, k + s]$. Then

$$\begin{aligned} \mathsf{Pr}[\Omega] \leq& \sum_{u=0}^{k+s-1} p^2(k + s, u, T) \cdot (\mathsf{Pr}[E_1 \geq u \mid E_0 = k - 1] \\ &- \mathsf{Pr}[E_1 \geq u + 1 \mid E_0 = k - 1]) \\ &+ p^2(k + s, k + s, T) \cdot \mathsf{Pr}[E_1 \geq k + s \mid E_0 = k - 1] \\ =&\mathsf{Pr}[E_1 \geq 0 | \, E_0 = k - 1] \cdot p^2(k + s, 0, T) \\ &+ \sum_{u=1}^{k+s} \mathsf{Pr}[E_1 \geq u \mid E_0 = k - 1] \\ &\cdot (p^2(k + s, u, T) - p^2(k + s, u - 1, T)). \end{aligned}$$

We show that $p(k + s, u, T)$ is increasing in u for all $T \geq 2$.

Proposition G.6. *For all positive integers k, s, u s.t. $u \leq k + s$, and for all $T \geq 2$, $p(k + s, u, T) \geq p(k + s, u - 1, T)$.*

Proof. We first show that the statement is true when $T = 2$. In this case

$$p(k + s, u, T) = \frac{1}{2^{k+s-u}},$$

which is increasing in u.

Now assume that the statement is true for $T = T' - 1$ where $T' \geq 3$, we show that when $T = T'$, $p(k+s, u, T) \geq p(k+s, u-1, T)$

for all positive integers k, s, u s.t. $u \leq k + s$. To this end, we show the following relation:

$$p(k+s, u-1, T) = \frac{1}{2}(p(k+s, u, T) + p(k+s, u-1, T-1)).$$

By the fact that

$$\binom{m+1}{n} = \binom{m}{n} + \binom{m}{n-1},$$

we have

$$\begin{aligned}
& p(k+s, u-1, T) - \frac{1}{2}p(k+s, u, T) \\
=& \sum_{j=0}^{T-2} \left(\binom{k+s-u+j}{k+s-u} \cdot \frac{1}{2^{k+s-u+j+1}} \right. \\
& \left. - \binom{k+s-u+j-1}{k+s-u-1} \cdot \frac{1}{2^{k+s-u+j+1}} \right) \\
=& \sum_{j=1}^{T-2} \left(\binom{k+s-u+j-1}{k+s-u} \cdot \frac{1}{2^{k+s-u+j+1}} \right) \\
=& \sum_{j=0}^{T-3} \left(\binom{k+s-u+j}{k+s-u} \cdot \frac{1}{2^{k+s-u+j+2}} \right) \\
=& \frac{1}{2} p(k+s, u-1, T-1).
\end{aligned}$$

Now we use induction to show that $p(k+s, u, T) \geq p(k+s, u-1, T)$.

- When $u = k + s$, we have $p(k+s, k+s, T) = 1$ and

$$p(k+s, k+s-1, T) = \frac{1}{2}(p(k+s, k+s, T) + p(k+s, k+s-1, T-1)).$$

  According to the induction hypothesis for $T - 1$, we have $p(k+s, k+s-1, T-1) \leq p(k+s, k+s, T-1) = 1$. Thus $p(k+s, k+s-1, T) \leq 1 = p(k+s, k+s, T)$.
- Now assume that when $u = u' + 1$ where $u' < k + s$, $p(k + s, u-1, T) \leq p(k+s, u, T)$. When $u = u'$, we have

$$\begin{aligned}
& p(k+s, u-1, T) - p(k+s, u, T) \\
=& \frac{1}{2}(p(k+s, u, T) + p(k+s, u-1, T-1)) \\
& -\frac{1}{2}(p(k+s, u+1, T) + p(k+s, u, T-1)) \\
=& \frac{1}{2}(p(k+s, u, T) - p(k+s, u+1, T)) \\
& +\frac{1}{2}(p(k+s, u-1, T-1) - p(k+s, u, T-1)).
\end{aligned}$$

  By induction hypothesis for $T-1$, we have $p(k+s, u-1, T-1) - p(k+s, u, T-1) \leq 0$. By induction hypothesis for $u = u' + 1$, we have $p(k+s, u, T) - p(k+s, u+1, T) \leq 0$. Thus $p(k+s, u-1, T) - p(k+s, u, T) \leq 0$.
- By induction, $p(k+s, u, T) \geq p(k+s, u-1, T)$.

Thus, $p(k+s, u, T)$ is increasing for $T = T'$. By induction, the statement holds. □

From Proposition G.1, we can obtain

$$\Pr[E_1 \geq u \mid E_0 = k-1] \leq \frac{1}{2^{u-k+1}}.$$

On the other hand when $u \leq k - 1$, we have $\Pr[E_1 \geq u \mid E_0 = k-1] \leq 1$. Then according to Proposition G.6,

$$\begin{aligned}
\Pr[\Omega] \leq\ & \Pr[E_1 \geq 0 | \, E_0 = k-1] \cdot p^2(k+s, 0, T) \\
& + \sum_{u=1}^{k+s} \Pr[E_1 \geq u \mid E_0 = k-1] \cdot (p^2(k+s, u, T) \\
& - p^2(k+s, u-1, T)) \\
\leq\ & p^2(k+s, 0, T) + \sum_{u=1}^{k-1} (p^2(k+s, u, T) - p^2(k+s, u-1, T)) \\
& + \sum_{u=k}^{k+s} \frac{1}{2^{u-k+1}} \cdot (p^2(k+s, u, T) - p^2(k+s, u-1, T)) \\
=\ & \sum_{u=k}^{k+s} \frac{1}{2^{u-k+1}} p^2(k+s, u-1, T) + \frac{1}{2^{s+1}} p^2(k+s, k+s, T) \\
=\ & \sum_{i=0}^{s} \frac{1}{2^{i+1}} \left(\sum_{j=0}^{T-2} \binom{s-i+j}{s-i} \cdot \frac{1}{2^{s-i+1+j}} \right)^2 + \frac{1}{2^{s+1}}.
\end{aligned}$$

□

G.6 Analysis of Winning Probability in Lemma 5.1 and Lemma G.3

G.6.1 Analysis of Winning Probability in Lemma 5.1. In this section, we analyse the winning probability in Lemma 5.1.

Recall that for positive integers k, s, T, the winning probability of $\mathcal{A}_g$ is bounded by $\sum_{j=0}^{T-1} \binom{s+j}{s} \cdot \frac{1}{2^{s+1+j}}$. We discuss three cases depending on the relation between s and T.

Case 1: $s < T$. In this case, we note that the winning probability is bounded by 1.

Case 2: $T \leq s < 3T$. In this case, we note that for all $j \in \{0, \ldots, T-1\}$,

$$\begin{aligned}
\binom{s+j}{s} \cdot \frac{1}{2^{s+1+j}} &= \frac{(s+j)!}{s! \cdot j!} \cdot \frac{1}{2^{s+1+j}} \\
&= \frac{1}{2^{s+1}} \cdot \prod_{i=1}^{j} \frac{s+i}{2i} \\
&\leq \frac{1}{2^{s+1}} \cdot \prod_{i=1}^{T} \frac{s+i}{2i} \\
&= \binom{s+T}{s} \cdot \frac{1}{2^{s+1+T}}.
\end{aligned}$$

Thus, $\sum_{j=0}^{T-1} \binom{s+j}{s} \cdot \frac{1}{2^{s+1+j}} \leq T \cdot \binom{s+T}{s} \cdot \frac{1}{2^{s+1+T}}$.

Now we apply the Stirling's approximation to estimate $\binom{s+T}{s}$. Recall the Stirling's approximation:

$$n! \sim \sqrt{2\pi n}(n/e)^n$$

Thus

$$\begin{aligned}\binom{s+T}{s} &= \frac{(s+T)!}{s!\cdot T!}\\ &\approx \exp\left((s+T)\ln(s+T) - s\ln s - T\ln T\right)\\ &\quad \cdot\exp\left((\ln(s+T) - \ln s - \ln T - \ln(2\pi))/2\right)\\ &= \exp\left(s\ln(1+T/s) + T\ln(1+s/T)\right)\\ &\quad \cdot\exp\left((\ln(s+T) - \ln(2s) - \ln T - \ln\pi)/2\right)\\ &\le \exp\left(T(1+\ln(1+s/T)) - (\ln T)/2\right).\end{aligned}$$

Therefore

$$\begin{aligned}&T\cdot\binom{s+T}{s}\cdot\frac{1}{2^{s+1+T}}\\ \approx\ & \exp\left(\ln T - (s+1+T)\ln 2 + T(1+\ln(1+s/T)) - (\ln T)/2\right)\\ \le\ & \exp\left(-s\ln 2 + T(1-\ln 2+\ln(1+s/T)) + (\ln T)/2\right).\end{aligned}$$

Case 3: $s \ge 3T$. In this case, we note that for all $j \in \{0,\dots,T-1\}$,

$$\begin{aligned}\binom{s+j}{s}\cdot\frac{1}{2^{s+1+j}} &= \frac{(s+j)!}{s!\cdot j!}\cdot\frac{1}{2^{s+1+j}}\\ &= \frac{1}{2^{s+1}}\cdot\prod_{i=1}^{j}\frac{s+i}{2i}\\ &\le \frac{1}{2^{s+1}}\cdot\prod_{i=1}^{j}\frac{s+i}{2i}\prod_{i=j+1}^{T}\frac{s+i}{4i}\\ &= \frac{1}{2^{T-j}}\cdot\binom{s+T}{s}\cdot\frac{1}{2^{s+1+T}}.\end{aligned}$$

Thus, $\sum_{j=0}^{T-1}\binom{s+j}{s}\cdot\frac{1}{2^{s+1+j}} \le \binom{s+T}{s}\cdot\frac{1}{2^{s+1+T}}$.

Following a similar analysis, we have

$$\begin{aligned}&\binom{s+T}{s}\cdot\frac{1}{2^{s+1+T}}\\ \approx\ & \exp\left(-(s+1+T)\ln 2 + T(1+\ln(1+s/T)) - (\ln T)/2\right)\\ \le\ & \exp\left(-s\ln 2 + T(1-\ln 2+\ln(1+s/T))\right).\end{aligned}$$

Approximating s to Achieve λ-bit Security. In general, we assume that $T < \lambda$ and $3T \le s$. Thus, we focus on the third case. To achieve λ-bit security, it is sufficient to set s s.t.

$$\exp\left(-s\ln 2 + T(1-\ln 2+\ln(1+s/T))\right) \le 2^{-\lambda}.$$

For simplicity, we use $\log(\cdot)$ to denote $\log_2(\cdot)$. Then it is equivalent to $-s + T(1/\ln 2 - 1 + \log(1+s/T)) \le -\lambda$, or

$$s \ge \lambda + T(1/\ln 2 - 1 + \log(1+s/T)). \tag{13}$$

We note that when $s \ge 3T$, $T\log(1+s/T) = s\cdot(\frac{T}{s}\log(1+\frac{s}{T})) \le 2s/3$. Also note that $1/\ln 2 - 1 \le 1/2$. Thus, when $s \ge 3\lambda + 3T/2$, we have

$$s = s/3 + 2s/3 \ge \lambda + T/2 + 2s/3 \ge \lambda + T(1/\ln 2 - 1 + \log(1+s/T)).$$

Now we show that when $s \ge \lambda + T(1/2 + \log(5/2 + 3\lambda/T))$, the Equation 13 always holds. Consider the following two cases:

- If $s \ge 3\lambda + 3T/2$, by the above analysis, Equation 13 always holds.
- If $s < 3\lambda + 3T/2$, then

$$\begin{aligned}s &\ge \lambda + T(1/2 + \log(5/2 + 3\lambda/T))\\ &\ge \lambda + T(1/\ln 2 - 1 + \log(1 + (3\lambda + 3T/2)/T))\\ &> \lambda + T(1/\ln 2 - 1 + \log(1+s/T)).\end{aligned}$$

Thus, it is sufficient to set

$$s = \lambda + T(1/2 + \log(5/2 + 3\lambda/T)) = \lambda + O(T\cdot\log(\lambda/T)).$$

G.6.2 Analysis of Winning Probability in Lemma G.3. In this section, we analyse the winning probability in Lemma G.3.

Recall that for positive integers k, s, T, the winning probability of $\mathcal{A}_g$ is bounded by

$$\sum_{i=0}^{s}\frac{1}{2^{i+1}}\left(\sum_{j=0}^{T-2}\binom{s-i+j}{s-i}\cdot\frac{1}{2^{s-i+1+j}}\right)^2 + \frac{1}{2^{s+1}}.$$

Following a similar analysis to Appendix G.6.1,

- When $s - i \le T$, $\sum_{j=0}^{T-2}\binom{s-i+j}{s-i}\cdot\frac{1}{2^{s-i+1+j}} \le 1$.
- When $s - i \ge T$,

$$\begin{aligned}&\sum_{j=0}^{T-2}\binom{s-i+j}{s-i}\cdot\frac{1}{2^{s-i+1+j}}\\ \le\ & T\binom{s-i+T}{s-i}\cdot\frac{1}{2^{s-i+1+T}}\\ \le\ & \exp\left(-(s-i)\ln 2 + T(1-\ln 2 +\right.\\ & \left.\ln(1+(s-i)/T)) + (\ln T)/2\right).\end{aligned}$$

Thus,

$$\begin{aligned}&\sum_{i=0}^{s}\frac{1}{2^{i+1}}\left(\sum_{j=0}^{T-2}\binom{s-i+j}{s-i}\cdot\frac{1}{2^{s-i+1+j}}\right)^2 + \frac{1}{2^{s+1}}\\ \le\ & \sum_{i=0}^{s-T}\frac{1}{2^{i+1}}\cdot\exp\left(-2(s-i)\ln 2 + 2T(1-\ln 2\right.\\ & \left.+\ln(1+(s-i)/T)) + \ln T\right) + \sum_{i=s-T+1}^{s}\frac{1}{2^{i+1}} + \frac{1}{2^{s+1}}\\ =\ & \sum_{i=0}^{s-T}\exp\left(-(2s-i+1)\ln 2 + 2T(1-\ln 2\right.\\ & \left.+\ln(1+(s-i)/T)) + \ln T\right) + \frac{1}{2^{s-T+1}}.\end{aligned}$$

We view i as an variable and consider the function $f(x) = -(2s - x + 1)\ln 2 + 2T(1 - \ln 2 + \ln(1 + (s-x)/T)) + \ln T$. Then $f'(x) = \ln 2 - 2/(1 + (s-x)/T)$. When $x \le s - T$, $f'(x)$ is decreasing, and the solution of $f'(x) = 0$ is $x = s - T(2/\ln 2 - 1)$. Thus,

$$\begin{aligned}\max_{x\le s-T} f(x) &= -(s+1)\ln 2 - T(2-\ln 2)\\ &\quad +2T(1-\ln 2+\ln(2/\ln 2)) + \ln T\\ &= -(s+1)\ln 2 + T(\ln 2 - 2\ln\ln 2) + \ln T.\end{aligned}$$

We have

$$\begin{aligned}
&\sum_{i=0}^{s} \frac{1}{2^{i+1}} \left(\sum_{j=0}^{T-2} \binom{s-i+j}{s-i} \cdot \frac{1}{2^{s-i+1+j}} \right)^2 \\
\leq \quad & \sum_{i=0}^{s-T} \exp\left(-(2s-i+1)\ln 2 + 2T(1-\ln 2\right. \\
& \left. + \ln(1+(s-i)/T)) + (\ln T)\right) + \frac{1}{2^{s-T+1}} \\
\leq \quad & s \cdot \exp\left(\max_{x \leq s-T} f(x) \right) + \frac{1}{2^{s-T+1}} \\
\leq \quad & \exp\left(\ln s - (s+1)\ln 2 + T(\ln 2 - 2\ln\ln 2) + \ln T\right) + \frac{1}{2^{s-T+1}}.
\end{aligned}$$

Approximating s to Achieve λ-bit Security. To achieve λ-bit security, it is sufficient to set s s.t.

$$\exp\left(\ln s - (s+1)\ln 2 + T(\ln 2 - 2\ln\ln 2) + \ln T\right) \leq 2^{-\lambda-1}$$
$$\frac{1}{2^{s-T+1}} \leq 2^{-\lambda-1}$$

The first condition is equivalent to $s \geq \lambda + T(1-2\log\ln 2) + \log T + \log s$ and the second condition is equivalent to $s \geq \lambda + T$ (which is implied by the first condition).

We note that when $s \geq 4$, we always have $\log s \leq s/2$. Thus, if $s \geq 2(\lambda + T(1-2\log\ln 2) + \log T)$, then

$$s = s/2 + s/2 \geq \lambda + T(1-2\log\ln 2) + \log T + \log s.$$

Now we show that when $s \geq \lambda + T(1-2\log\ln 2) + \log T + 1 + \log(\lambda + T(1-2\log\ln 2) + \log T)$, the first condition always holds. Consider the following two cases:

- If $s \geq 2(\lambda + T(1-2\log\ln 2) + \log T)$, by the above analysis, the first condition always holds.
- If $s < 2(\lambda + T(1-2\log\ln 2) + \log T)$, then

$$\begin{aligned}
s \quad \geq \quad & \lambda + T(1-2\log\ln 2) + \log T + 1 \\
& + \log(\lambda + T(1-2\log\ln 2) + \log T) \\
= \quad & \lambda + T(1-2\log\ln 2) + \log T \\
& + \log 2(\lambda + T(1-2\log\ln 2) + \log T) \\
> \quad & \lambda + T(1-2\log\ln 2) + \log T + \log s.
\end{aligned}$$

Thus, it is sufficient to set

$$\begin{aligned}
s =& \lambda + T(1-2\log\ln 2) + \log T + 1 \\
& + \log(\lambda + T(1-2\log\ln 2) + \log T) \\
=& \lambda + O(T + \log\lambda).
\end{aligned}$$

H Related Works

We survey some relevant related works, specifically setting in the honest majority scenario, with active security and security with abort.

Protocols Using Shamir Secret Sharing over Fields. In the setting of the standard honest majority setting ($t < n/2$), the DN07 protocol [23] is the first protocol for honest majority with linear communication complexity, but in its most basic version it is not actively secure. Many works build on top of this protocol to achieve active security, all incurring in different costs. The work of [18] adds an overhead of 2× in communication, and the works of [10, 12, 32] use sublinear distributed product checks and show that one can achieve active security at essentially the same communication of semi-honest DN07. On the other hand, the recent work [30] improves the communication complexity of the semi-honest DN07 protocol by 33%, and shows how to use the techniques in [10, 12, 32] to achieve malicious security with the same communication complexity as the semi-honest protocol.

In the setting of $t < n/3$ (which is necessary for perfect security), a line of works focus on improving the communication complexity of perfectly secure MPC [1, 2, 7, 9, 31]. We point out that all these works focus on general n-party computation.

Protocols Using Shamir Secret Sharing over $\mathbb{Z}_{2^k}$. There are only a few protocols that use Shamir secret sharing over $\mathbb{Z}_{2^k}$, which requires Galois ring extensions. The first is the work of [3], which made the observation that this was in fact possible, and is mostly of theoretical interest since it builds on older, less efficient protocols. The next work that explored the use of Shamir secret sharing is [4], which compiled the basic passive protocol from [3] to active security by extending the ideas from [18] to the ring case, with the help of the SPDZ2k trick from [19].

The work of [10] presents a generic way of compiling passive protocols into actively secure protocols, using sublinear distributed product checks, which, as they show, can be made to work over $\mathbb{Z}_{2^k}$ by using ring extensions. One can obtain an actively secure protocol over rings using Shamir secret sharing, with better complexity than [4] by compiling the passive protocol from [3] using the ideas from [10] ([12] also provides results based on sublinear distributed product checks and Shamir secret sharing, which can be adapted as well).

It may be worth pointing out that Shamir secret sharing can be used over much more general rings, even non-commutative ones, and MPC protocols over these can be designed using this primitive. This was explored in [25].

Protocols Using Replicated Secret Sharing. The share size in replicated secret sharing scaled exponentially with the number of parties, and hence it is not appropriate for use in settings with a large number of parties. In spite of this, a few works have considered this primitive since, on one hand, it satisfies certain useful properties that Shamir secret sharing lacks, and on the other hand, for a constant number of parties, a careful design and implementation can lead to better performance than using Shamir's.

The work of [12] uses replicated secret sharing in conjunction to sublinear distribute product proofs to obtain active security with guaranteed output delivery. This is set in the context of finite fields and also $\mathbb{Z}_{2^k}$, but it uses large Galois ring extensions in the latter case. Finally, the work of [20], which is not honest majority but $t < n/3$, also considers replicated secret sharing for an arbitrary number of parties. This work shows that replicated secret sharing is useful for working over an arbitrary, possibly non-commutative rings (in particular, $\mathbb{Z}_{2^k}$), and it also shows experimentally that such scheme can be used practically for number of parties that range in the order of dozens.

Other works like [15, 39] have explored using (variants of) replicated secret sharing in the four-party setting, but they are in two-thirds honest majority where $t < n/3$, and they do not require sublinear distributed product checks.

Three-party Computation Protocols. It is common to use replicated secret sharing specifically for the case of three-party computation, since in this setting it can be particularly efficient. There are multiple works that study this setting. The work of [6] presents a *passively* secure 3PC protocol using replicated secret sharing, and their multiplication protocol only requires each party to send one ring element to another party. This protocol works for any ring, and as we will discuss next, it underlies many of the subsequent 3PC constructions. The works of [27] and [5] extend the protocol above by adding active security using techniques based on cut-and-choose, specifically for the binary case (*i.e.* the ring is $\mathbb{Z}_2$). However, these works do not extend to $\mathbb{Z}_{2^k}$ for $k > 1$.

The work of [18] shows how to compile any passive protocol over *fields*, and in particular, how to use the passive three-party protocol from [6] in conjunction with their framework to obtain an active version of it. The work of [4] generalizes the 3PC protocol in [18] from fields to $\mathbb{Z}_{2^k}$, which can be seen as adding MACs to the passive protocol from [6]. The resulting communication complexity per multiplication gate per party, is $2(k + \lambda)$ bits, where λ is roughly a statistical security parameter.

The ABY3 protocol [40] builds on top of the multiplication approach over $\mathbb{Z}_{2^k}$ in [6], and extends it with a set of primitives useful for machine learning computations.[17] Secure NN [44] is also set in the $\mathbb{Z}_{2^k}$ setting, but relies on additive secret sharing between two of the parties (instead of replicated secret sharing), and multiplication triples generated by the third party for the products.

An important work that changed the paradigm to achieve active security is that of [11]. That work also starts from the passive protocol in [6], but it adds active security not by using MACs as in [4], but by employing sublinear distributed product checks, introduced in [10]. Using these techniques, the actively secure protocol over $\mathbb{Z}_{2^k}$ in [11] achieved a communication complexity that remained the same as the passive counterpart, namely, k bits per party. Given this appealing feature, the subsequent works of [14, 38, 42], which also consider 3PC for $\mathbb{Z}_{2^k}$ and active security, used the protocol by [11] in a *black-box* way to implement their underlying primitives. Focusing on SWIFT [38], which is the state-of-the-art among these three protocols, their multiplication protocol requires 2 elements in $\mathbb{Z}_{2^k}$ per party, distributed half and half in the offline and online phases, which is twice the cost of [11].

It is worth mentioning the 3PC protocol over $\mathbb{Z}_{2^k}$ of [24], which is similar to the one in [4] in the use of MACs, but achieves a worse communication complexity of $3(k + \ell)$ bits per party per multiplication, and is indeed shown in [4] to perform worse in practice.

Relevant Mention. The compiler in [21] works for arbitrary secret-sharing-based passively secure protocols over an arbitrary ring to achieve actively security with abort. It aims at optimizing the online phase, for which the authors preprocess function-dependent multiplication triples which are then used online. Furthermore, it needs to perform a correctness check, for which the author propose either sacrificing-based techniques or sublinear distributed product checks. Using sacrificing requires two triples per multiplication gate, which makes the total cost at least four times that of the semi-honest protocol (not counting the cost of sacrificing itself), which is worse than [4]. On the other hand, using sublinear distributed product checks, the cost in the function-independent offline phase alone will be at least the cost from [11].

Finally, we mention the protocols of [33, 37] over $\mathbb{Z}_{2^k}$. Both works are based on replicated secret sharing and achieve fairness and G.O.D., but focus on different settings. The work of [33] is set in the four-party setting in the FaF (Friends and Foes) model, while the work of [37] is set in the five-party scenario with two corruptions, which puts it in the standard honest majority context. Both works employ distributed product checks as in [11, 12] to check the correctness of the computation, which requires them to use large degree Galois ring extensions. Our sublinear distributed product checks can potentially be used to improve the concrete efficiency of [33, 37].

I Experiments running with multiple threads

We summarize our experimental results of running the two protocols with 10 threads in the verification phase. In LAN, the two protocols show almost the same parallelization improvements on different circuits, and thus our protocol maintains similar $17.2 \sim 46.8\times$ speedup over [11] as in the single-thread setting. As for WAN where communication plays a more important role, computational parallelization will have less effect on end-to-end runtimes, and thus our advantage over [11] will decrease. However, the experimental results show that our protocol still has up to $4.5\times$ speedup over [11] for 10M multiplication gates with depth 10. Note that as the circuit size decreases, the speedup factor of our protocol over [11] gets smaller. This reflects from another perspective that our computational task is much lighter than [11] so that paralleling computational tasks of our protocol makes less effect on end-to-end runtimes.

[17] ABY3 claims active security using, for the product, the approach from [27]. However, it is not clear how this would work since the latter protocol uses cut-and-choose and triple sacrificing over $\mathbb{Z}_2$, and even though it can be generalized to $\mathbb{F}_p$, it cannot be made to work over $\mathbb{Z}_{2^k}$.