Attaining GOD Beyond Honest Majority With Friends and Foes

Aditya Hegde[1]*, Nishat Koti[2], Varsha Bhat Kukkala[2], Shravani Patil[2], Arpita Patra[2], and Protik Paul[2]

[1] Johns Hopkins University ahegde@cs.jhu.edu
[2] Indian Institute of Science, Bangalore
{kotis,varshak,shravanip,arpita,protikpaul}@iisc.ac.in

Abstract. In the classical notion of multiparty computation (MPC), an honest party learning private inputs of others, either as a part of protocol specification or due to a malicious party's unspecified messages, is not considered a potential breach. Several works in the literature exploit this seemingly minor loophole to achieve the strongest security of guaranteed output delivery via a trusted third party, which nullifies the purpose of MPC. Alon et al. (CRYPTO 2020) presented the notion of *Friends and Foes* (`FaF`) security, which accounts for such undesired leakage towards honest parties by modelling them as semi-honest (friends) who do not collude with malicious parties (foes). With real-world applications in mind, it's more realistic to assume parties are semi-honest rather than completely honest, hence it is imperative to design efficient protocols conforming to the `FaF` security model.
Our contributions are not only motivated by the practical viewpoint, but also consider the theoretical aspects of `FaF` security. We prove the necessity of semi-honest oblivious transfer for `FaF`-secure protocols with optimal resiliency. On the practical side, we present QuadSquad, a ring-based 4PC protocol, which achieves fairness and GOD in the `FaF` model, with an optimal corruption of 1 malicious and 1 semi-honest party. QuadSquad is, to the best of our knowledge, the first practically efficient `FaF` secure protocol with optimal resiliency. Its performance is comparable to the state-of-the-art dishonest majority protocols while improving the security guarantee from abort to fairness and GOD. Further, QuadSquad elevates the security by tackling a stronger adversarial model over the state-of-the-art honest-majority protocols, while offering a comparable performance for the input-dependent computation. We corroborate these claims by benchmarking the performance of QuadSquad. We also consider the application of liquidity matching that deals with highly sensitive financial transaction data, where `FaF` security is apt. We design a range of `FaF` secure building blocks to securely realize liquidity matching as well as other popular applications such as privacy-preserving machine learning (PPML). Inclusion of these blocks makes QuadSquad a comprehensive framework.

Keywords: Friends and Foes · Multiparty Computation · Oblivious Transfer

* Work done while at International Institute of Information Technology Bangalore.

1 Introduction

The classical notion of multiparty computation (MPC) enables n mutually distrusting parties to compute a function over their private inputs, such that an adversary controlling up to t parties does not learn anything other than the output. Depending on its behaviour, the adversary can be categorized as *semi-honest* or *malicious*. A maliciously-secure MPC protocol may offer security guarantee with *abort, fairness* or *guaranteed output delivery (GOD)*. While security with abort may allow the adversary alone to receive the output (leaving out the honest parties), fairness makes sure either all or none receive the output. The strongest guarantee of GOD ensures that all honest participants receive the output irrespective of the adversarial behaviour. It is well known that honest majority is necessary to achieve GOD, whereas a dishonest majority setting can at best offer security with abort for general functionalities [29]. GOD is undoubtedly one of the most attractive features of an MPC protocol. Preventing repeated failures, it upholds the trust and interest of participants in the deployed protocol and saves a participant's valuable time and resources. Moreover, it also captures unforeseeable scenarios such as machine crashes and network delay.

It is well-known that the honest majority setting lends itself well for constructing efficient protocols for a large number of parties [34,2,1,18,47] and has been shown to be practical [68,6]. In this setting, MPC for a small number of parties [5,41,4,61,28,67,65,51,67,23] has gained popularity over the last few years due to applications such as financial data analysis [17] and privacy-preserving statistical studies [15] which typically involve 3 parties. This is corroborated by the popularity of MPC framework such as Sharemind [16] which works over 3 parties. In the literature, of all MPC protocols for a small population, several achieve the highest security guarantee of GOD [51,22,46,20,24,58,59]. In most of these protocols, when any malicious behaviour is detected, parties identify an *honest* party, referred to as Trusted Third Party (TTP) and make their inputs available to it in clear. Thereafter, TTP computes the desired function on parties' private inputs and returns the respective outputs. Such learning of inputs by an honest party is allowed in the traditional definition of security, although it nullifies the main purpose of MPC. In many real-world applications that deal with highly sensitive data, such as those in financial and healthcare sectors, information leak, even to an honest party, is unacceptable. Further, in the secure outsourced computation setting, where servers (typically run by reputed companies such as Amazon, Google, etc.) are hired to carry out the computation, it may be unacceptable to reveal private inputs to the server identified as a TTP.

Another issue that persists in traditionally secure MPC protocols is the following. The malicious adversary can potentially breach privacy of protocols by sending its view to some of the honest parties. However, traditional definitions do not acknowledge this view-leakage as an attack as honest parties are assumed to discard any non-protocol messages. In this way, traditional definition fails to account for the possibly curious nature of honest parties, which is a given in real-world scenarios. Consequently, many well-known protocols relying on threshold secret sharing (such as BGW [14]), satisfying traditional security against t malicious corruptions, immediately fall prey to this view-leakage attack. Indeed, an honest party on receiving the view of any t corrupt parties can learn the inputs of all the parties. Note that the traditional MPC protocols are vulnerable to this view-leakage attack which are not just restricted to GOD protocols but also protocols with weaker security notion of fairness. We emphasize that such a view-leakage attack is not irrational on the part of adversary's behaviour as it can be motivated by monetary incentives.

We showcase how reliance on a TTP and the view-leakage attack inherent in traditionally secure MPC is detrimental to data privacy in real-world applications via the example of liquidity matching. Consider a set of banks that have outstanding transactions that need to be settled among themselves. Liquidity matching enables settlement of inter-bank transactions while ensuring that each bank has sufficient liquidity. Here, liquidity means the balance of a bank, and matching requires that each bank, upon processing of the outstanding transactions, has non-negative balance. Since transactions comprise sensitive financial data, it is required to perform liquidity matching in a privacy-preserving manner. Hence, when designing MPC protocols for the same. It is imperative for the protocol to provide GOD, owing to the real-time nature of such transactions. That is, aborting the execution is not an acceptable option as it may lead to an indefinite delay in processing the transactions. The work of [7] has explored this application in the traditionally secure MPC setting. However, given the sensitive nature of the application, reliance on a TTP to attain GOD, and the view-leakage attack, render the traditionally secure MPC solution futile.

Inspired by the above compelling concerns of reliance on a TTP and view-leakage, [3] proposed a new MPC security definition, Friends & Foes (FaF). In this definition, an honest party's input is required to be safeguarded from quorums of other honest parties, in addition to the standard security against an adversary. This dual need is modelled through a decentralized adversary. Specifically, there is one malicious adversary that corrupts at most t out of n parties *(Foes)* and another semi-honest adversary, controlling at most h^* parties *(Friends)* out of the remaining $n-t$ parties. A protocol secure against such adversaries is said to be (t, h^*)-FaF secure. Technically, in the FaF model, not only should the views of t malicious parties, but also the views of every (disjoint) subset of h^* semi-honest parties, be simulatable separately (details appear in §A). Moreover, FaF requires security to hold even when the malicious adversary sends its view to some of the other parties (semi-honest). Thus, FaF-security is a better fit for applications that deal with highly sensitive data, as in the case of liquidity matching.

Alon et al. showed in [3] that any functionality can be computed with fairness and GOD in the (t, h^*)-FaF model, iff $2t + h^* < n$ holds. Since protocols with a small number of parties are pragmatic, from the above condition it is evident that a minimum of 4 parties is necessary to achieve the desired level of FaF-security. This implies that $t = 1, h^* = 1$. While the sufficiency of $t = 1$ is well established in the literature [62,71,58,59,32,46,20,22,24], we trust that $h^* = 1$ also suffices for most practical purposes, assuming honest parties do not collude. Thus, we design protocols in the 4PC setting providing $(1, 1)$-FaF security. It is worth noting that relying on a 4PC protocol with 2 malicious corruptions to achieve this goal is insufficient, since GOD is known to be impossible in this setting. On the other hand, although the 4 party honest-majority setting tackling a single corruption can offer GOD security, it is susceptible to the view-leakage attack.

Keeping practicality in mind, for the optimal 4PC setting considered above, we describe the design choices made to attain an efficient protocol. To obtain a fast-response time as required for real-time applications, we operate in the preprocessing paradigm which has been extensively explored [36,9,35,56,58,59]. Here, the protocols are partitioned into two phases, a function dependent (input independent) *preprocessing phase* and an input dependent *online phase*. Also, following recent works [16,35,37,33] we build our protocols over 32 or 64 bit rings to leverage CPU optimizations. Further, to aid resource constrained clients in performing computationally intensive tasks, the paradigm of secure outsourced computation (SOC) has gained popularity. In this setting, clients can avail the computationally powerful servers on a 'pay-per-use' basis from Cloud service providers. In this work, we provide secure protocols for performing computations in the 4-server SOC setting. The servers here are mapped to the parties of our 4PC.

When designing FaF-secure protocols in a given setting, it is both theoretically profound and practically important to know, whether information-theoretic security is possible to be achieved. If not, it is important to identify what cryptographic assumption is required. [3] shows impossibility of information-theoretic FaF-secure MPC with less than $2t + 2h^*$ parties and presents a protocol relying on semi-honest oblivious transfer (OT) with at least $2t + h^* + 1$ parties. However, the necessity of OT in the latter setting was not known. We settle this question, showing the necessity of semi-honest OT. This proves the tightness of the protocol of [3] in terms of assumption, and implies that any 4PC in $(1, 1)$-FaF setting requires semi-honest OT. This requirement puts FaF security closer to the dishonest majority setting where the same necessity holds [45,52], than the honest majority setting which is known to offer even the strongest security of GOD information-theoretically.

1.1 Related Work

We restrict the discussion to practically-efficient secret-sharing based (high throughput regime) MPC protocols over small population for arithmetic and Boolean world, since this is the regime of focus in this work.

In the honest-majority setting, we restrict to protocols achieving fairness and GOD over rings. The GOD protocol offering the best *overall* communication cost is that of [20]. [25,71,58], present 3PC protocols in the preprocessing paradigm, and thus have faster online phase than [20]. Of these, [58] elevates the security of the former two, from fairness to GOD. In the 4PC regime, [59] presents the best GOD protocol improving over the previously best-known fair protocol of [26] and GOD protocol of [22,58].

The work that comes closest to the FaF notion in terms of security in the four party setting is that of Fantastic Four [32] which is devoid of function dependent preprocessing. It attempts to offer a variant of GOD, referred to as *private robustness* without the honest party learning other parties' inputs. However, this work does not capture the behaviour of a malicious adversary which allows it to send its complete view to an honest party, thus falling short of satisfying the FaF security notion. This aspect is captured in the recent work PentaGOD [57] which achieves $(1,1)$-FaF security with GOD in the 5-party setting. However, we know that 4 parties are sufficient for $(1,1)$-FaF secure fair/GOD protocol. [57] lets go of the optimal resiliency and considers an additional party to move to honest majority and hence achieves better efficiency. However, keeping the focus on honest majority, their construction is specifically customised for $(1,1)$-FaF setting. Hence they do not account for a few other FaF corruption scenarios for which GOD is possible in the 5-party setting.

In the dishonest-majority setting, the study of practically-efficient protocols started with the work of [36] which was followed by [55,56]. This line of work culminated with [13] which has the fastest online phase. However, these protocols work over fields. The works that extend over rings are [31,69] and of these the latter is a better performer. In this regime, all the protocols work in preprocessing paradigm, where the common trend had been to generate Beaver multiplication triples [11] in the preprocessing and consume them in the online phase for multiplication. The majority of the works focus on bettering the preprocessing and choose either Oblivious Transfer (OT) [55] or Somewhat Homomorphic Encryption (SHE) [36,31,69] to enable triple generation.

1.2 Our Contribution

QuadSquad: A $(1,1)$-FaF Secure 4PC. We propose the first, efficient, $(1,1)$-FaF secure, 4PC protocol in the preprocessing paradigm, over rings (both $\mathbb{Z}_{2^\lambda}$ and $\mathbb{Z}_2$), that achieves fairness and GOD. Casting our protocol in the preprocessing paradigm allows us to obtain a fast online phase, with a cost comparable to the best-known dishonest as well as honest majority protocols. Furthermore, we achieve GOD, without incurring any additional overhead in the online phase, in comparison to our fair protocol. This is depicted in Table 1.

Here, with respect to honest-majority protocols, we compare QuadSquad's multiplication with the best-known 4PC of Tetrad [59] which relies on a TTP, and the protocol of Fantastic Four [32] which offers *private robustness* without relying on a TTP. With respect to dishonest-majority protocols, we compare with the best-known OT-based protocol of MASCOT [55] since our protocol also relies on OTs in the preprocessing. While QuadSquad, [32] and [59] work over ring, [55] exploits

Ref.	Preproc.	Online		Model	Security
	Comm.	Rounds	Comm.		
Tetrad ($\mathbb{Z}_{2^\lambda}$)	2	1	3	HM	GOD
Fantastic Four ($\mathbb{Z}_{2^\lambda}$)	NA	1	6	HM	GOD
MASCOT ($\mathbb{F}$)	7713	2	12	DM	abort
QuadSquad ($\mathbb{Z}_{2^\lambda}$)	1558	3	7	FaF	Fair
QuadSquad ($\mathbb{Z}_{2^\lambda}$)	3110	3	7	FaF	GOD

– The comm. complexity is given in terms of elements from $\mathbb{Z}_{2^\lambda}/\mathbb{F}$ (of size 2^{64}), as applicable. HM: Honest majority; DM: Dishonest majority.

Table 1: Comparison of mult of MASCOT, Fantastic Four and Tetrad with QuadSquad

field ($\mathbb{F}$) structure. Further, the protocol in [32] does not have a separate preprocessing phase. We indicate this in Table 1 by "NA" (Not Applicable). As per the table, QuadSquad is comparable to both the honest-majority and dishonest-majority protocols in the online phase and outperforms [55] in the preprocessing. Our offer over [59], [32] is stronger security against an additional semi-honest corruption, with a comparable online cost. Our offer over [55] is the stronger guarantee of fairness/GOD with comparable online cost (and better preprocessing cost).

Necessity of OT. FaF is closer to dishonest majority (with 2 corruptions out of 4), and hence, public-key primitives are inevitable. We back this up by proving the necessity of OT. We prove the necessity of semi-honest OT for (t, h^*)-FaF (abort) secure protocol with $n \leq 2t + 2h^*$ (by constructing the former from the latter). The goal of this result is to justify that a protocol, including ours, in FaF-model will require public-key primitives. Given this, we use semi-honest OT, but restrict its use to preprocessing alone[3].

Building blocks and applications. We consider the application of liquidity matching where FaF security is more apt. We design a range of FaF secure building blocks to securely realize liquidity matching, as well as other popular applications such as privacy-preserving machine learning (PPML). The description of the building blocks appears in Table 2. Although these can be naively obtained by extending techniques from the literature, the resultant building blocks have a heavy communication overhead. We therefore go one step ahead and design customised building blocks which are efficient and help in improving the response time of these applications.

Protocol	Input	Output	Description
$[\![\cdot]\!]$-$\mathsf{Sh}^{\mathsf{SOC}}$	v	$[\![\mathsf{v}]\!]$	User $[\![\cdot]\!]$-shares input v with the servers
$[\![\cdot]\!]$-$\mathsf{Rec}^{\mathsf{SOC}}$	$[\![\mathsf{v}]\!]$	v	Servers reconstruct v to U
BitExt	$[\![\mathsf{v}]\!]$	$[\![\mathsf{msb}(\mathsf{v})]\!]^{\mathbf{B}}$	Extracts most significant bit of an arithmetic shared value v
Bit2A	$[\![\mathsf{b}]\!]^{\mathbf{B}}$	$[\![\mathsf{b}]\!]$	Converts boolean sharing of a bit b to arithmetic sharing
BitInj	$[\![\mathsf{b}]\!]^{\mathbf{B}}$, $[\![\mathsf{v}]\!]$	$[\![\mathsf{b} \cdot \mathsf{v}]\!]$	Outputs $[\![\cdot]\!]$-shares of $\mathsf{b} \cdot \mathsf{v}$, where bit b is $[\![\cdot]\!]^{\mathbf{B}}$-shared and v is $[\![\cdot]\!]$-shared
DotPTr	$\{[\![\mathsf{x}^s]\!], [\![\mathsf{y}^s]\!]\}_{s \in [\mathsf{n}]}$	$[\![\sum_{s \in [\mathsf{n}]} \mathsf{x}^s \cdot \mathsf{y}^s]\!]$	Outputs $[\![\cdot]\!]$-shares of dot product of $[\![\cdot]\!]$-shared vectors $\{\mathsf{x}^s\}_{s \in \mathsf{n}}$, $\{\mathsf{y}^s\}_{s \in \mathsf{n}}$

Table 2: Build blocks for various applications

Benchmarks. We showcase the practicality of QuadSquad by benchmarking its MPC, as well as the performance of secure liquidity matching and PPML inference for two Neural Networks (NN). We implement and benchmark our 4PC protocol over a WAN network using the ring $\mathbb{Z}_{2^{64}}$, and report the latency, throughput and communication costs in the preprocessing and online phase. We

[3] As mentioned in §1.1, SHE offers an alternative to OT. However, relying on the heels of recent interesting work on OT [76] and the huge effort on improving OT in the last decade [19,54], we opt for OT based approach. Translating our approach in the SHE regime is left for future exploration.

observe that the throughput of our GOD protocol is comparable to that of the fair protocol, and has an overhead of up to 4.5× in the online phase over [59] and [32]. This overhead indicates the cost to achieve the stronger notion of FaF-security. On the other hand, QuadSquad outperforms [55] by a factor of up to 4.5× in the online phase. With respect to the applications, we observe a runtime of 6 and 10 seconds for the two NNs, and a runtime of 15 seconds for liquidity matching. The reported runtime for both applications is practical.

1.3 Technical Highlights

In this section, we elaborate on the design choices of our protocol, the challenges involved and the approach taken to tackle them. One approach to achieving $(1,1)$-FaF security in the 4PC setting is via a 4-party identifiable abort protocol, where upon detecting a misbehaving party, the protocol can be re-run with a default input for the identified corrupt party. However, we deviate from this approach and choose dispute pair identification for achieving the desired security due to the following reasons. First, note that there is no customised 4PC identifiable abort protocol in the literature. Moreover, since the threshold of corruption in $(1,1)$-FaF considering malicious as well as semi-honest parties corresponds to a dishonest majority setting, we have to consider identifiable abort protocols in the same setting to prevent susceptibility to view-leakage attack. This would inherently require us to consider generic n-party dishonest majority identifiable abort protocols, instantiated for the specific case of $n = 4$ and $t = 2$, which do not offer a practically efficient solution. Specifically, the state-of-the-art protocol in this setting [10] requires online communication of 24 elements per multiplication-gate, which is significantly higher than the online communication cost of our protocol. Designing a customised 4PC identifiable abort protocol is an orthogonal question which is left as an open problem.

Necessity of OT. To prove the necessity of semi-honest OT for a generic n-party (t, h^*)-FaF secure (abort) protocol with $t + h^* < n \leq 2t + 2h^*$, we construct the former from the latter. Recall that the necessity of $n > t + h^*$ for abort security and $n > 2t+h^*$ for GOD in the FaF model is known from [3]. Note that our proof holds up to $n \leq 2t + 2h^*$, which subsumes the optimal bound on n for the GOD setting. We show that an n-party (t, h^*)-FaF secure protocol π_f for computing the function $f((m_0, m_1), \perp, \ldots, \perp, b) = (\perp, \perp, \ldots, \perp, m_b)$, where $n \leq 2t + 2h^*$, can be used to construct a semi-honest OT. We give the formal proof in §3.

QuadSquad: Robust $(1,1)$-FaF Secure 4PC. The core idea of our 4PC construction lies in designing the sharing, reconstruction and multiplication primitives.

Sharing: To facilitate operating over rings and to ensure privacy in FaF model with 1 malicious and 1 semi-honest party, we rely on Replicated Secret Sharing (RSS) with a threshold of 2. This requires 6 components where each pair of parties holds a common component. This is higher compared to the 4 components for RSS with threshold 1 and 3 which are typically used in honest and dishonest majority settings respectively. In QuadSquad, each party has only 3 components of a sharing which poses the challenge in ensuring a communication efficient reconstruction.

Reconstruction: Although a naive reconstruction towards all would require a communication of 12 elements, our protocol reduces this to an *amortized* cost of 7 elements. Both our sharing and reconstruction protocols extensively rely on primitives which leverage the honest behaviour of at least 3 parties to ensure dispute pair (DP) identification.

Multiplication: The higher number of components in our sharing semantics makes our multiplication protocol non-trivial. At a high level, the multiplication of 2 shared values results in 36 summands, which we broadly categorize into 3 types based on the number of parties which can locally compute each summand. We give separate treatment to each category, of which the summands that can be computed by a single party and those which cannot be computed by any party are of particular interest. The main challenge in the former is ensuring the correctness of a party's computation, for which we build upon the distributed Zero-Knowledge (ZK) protocol of [20]. The latter requires a new *distributed multiplication* protocol where two distinct pairs of parties hold the inputs to the multiplication and the goal is to additively share the product between the pairs. This primitive relies on OT. Here, the main challenge is ensuring the correctness of inputs to OT, for which we leverage the (semi) honest behaviour of at least 3 parties and the fact that every pair of parties holds a common component. Apart from several optimization techniques, the primary technical highlight in this part includes the new batch reconstruction and the distributed multiplication, both of which contribute to a highly efficient multiplication protocol.

Online: To ensure efficiency, we follow the paradigm of masked evaluation by tweaking the RSS sharing as follows. We share a value by using a mask which is RSS shared and a masked value which is public. The evaluation of the circuit is then performed on these publicly held masked values which are required to be reconstructed in the online phase [46,13,70].

Fair to GOD: In the optimistic run (where all parties behave honestly) of our 4PC protocol the function output is computed correctly. However, in case any malicious behaviour is detected during protocol execution, a dispute pair (DP) is identified which is assured to include the malicious party. The protocol that we obtain by terminating at the earliest point of dispute discovery, offers fairness. Note that the fair protocols existing in the literature [71,58,59] are susceptible to the view-leakage attack and thus are not FaF secure. Further, to extend the security guarantee to GOD without incurring additional communication overhead in the online phase, we follow the commonly used approach of segmented evaluation of a circuit. Specifically, we segment the circuit and execute the above protocol in a segment-by-segment manner. In case malicious behaviour is detected in any segment, as in our fair protocol, we identify a DP. Following this, for computation of the remaining segments, we resort to a single instance of a semi-honest 2PC which is executed by parties outside DP, which we refer to as the trusted pair (TP). We use the semi-honest 2PC in a black-box manner, and this can be instantiated with the state-of-the-art protocol. We use ABY2.0 [70], for this purpose, which is also designed in the preprocessing paradigm. To extend support for the online phase of [70], each pair of parties executes an instance of the preprocessing of [70], along with the preprocessing of QuadSquad. This ensures that in case DP is identified during the online phase, parties have the necessary preprocessed data for the 2PC.

Key differences from Tetrad, Fantastic Four and MASCOT. The best known honest-majority 4PC given in Tetrad differs from our construction in many aspects starting with reliance on RSS with threshold 1. This ensures every party misses a single (as opposed to 3 for us) component, offering a very efficient reconstruction. They further utilize high redundancy (every component is held by 3 parties) and heavily rely on isolating one of the parties from most of the computation. This, together with the threshold of 1 guarantees that, in case malicious behaviour is detected during the computation, the isolated party is honest. This honest party is then elevated to a TTP. The protocol of [58] follows a similar approach for efficiency. In FaF-model, we fall short of the first and the latter paradigm fails due to the presence of an additional semi-honest party. Thus, our multiplication protocol involves all four parties and enforces different mechanisms to detect and handle malicious behaviour compared to the Tetrad protocol. Similar to [59,58], the efficiency of Fantastic Four can be attributed to the benefits of redundancy offered by RSS with threshold 1. Their work achieves a variant of GOD referred to as *private robustness* by first identifying a dispute pair in the execution involving all 4 parties, followed by reducing the computation to a 3-party malicious protocol. For this, their work eliminates one party from the dispute pair arbitrarily. Any malicious behaviour hereafter, asserts that the party from the dispute pair included in the 3PC is malicious. To achieve robustness, they execute a semi-honest 2-party protocol using the parties guaranteed to be honest. Although their approach circumvents revealing private inputs to a TTP for achieving robustness, it falls short of offering FaF-security. In particular, it is susceptible to the view-leakage attack in all the instances of its sub-protocols involving $2, 3$ and 4 parties. Moreover, in [32], the switch from 4PC to 3PC upon identifying malicious behaviour is non-interactive. This can be attributed to the threshold of 1 which ensures that any three parties together possess all the components of the sharing. However, in our case, if any malicious behaviour is detected we fall back on a semi-honest 2PC. The sharing semantics of our protocol (required to prevent view-leakage attack) are such that a pair of parties does not hold all the shares. Hence we need additional interaction for converting from 4PC sharing to a 2PC sharing.

On the other hand, MASCOT [55] relies on RSS with threshold 3 (same as additive sharing). Though every party misses 3 shares like our case, riding on the advantage of shooting for a weaker guarantee of abort, they are able to leverage king-based approach [34] for reconstruction (only one party/king is enabled to reconstruct, which later sends the value to the rest) which only ensures detection, but falls short of recovery, from a malicious behaviour. [55] delegates checks to detect malicious behaviour to the end of the protocol whereas we need to verify correct behaviour at each step to ensure fairness/GOD.

Our work leaves open several interesting questions. We elaborate on these and the challenges involved therein in §D.

2 Preliminaries

Setting and Security. We consider a set of four parties $\mathcal{P} = \{P_1, P_2, P_3, P_4\}$ which are connected by pair-wise private and authenticated channels in a synchronous network. The function to be computed is expressed as a circuit whose topology is public and is evaluated over a ring $\mathbb{Z}_{2^\lambda}$ of size 2^λ. Our protocols are designed in the `FaF` model with a static malicious adversary and a (different) semi-honest adversary each corrupting at most one (distinct) party. We make use of broadcast channel for simplicity of presentation, which can be instantiated using any protocol such as [39]. Our constructions achieve the strongest security guarantee of GOD, wherein parties receive the protocol output irrespective of the malicious adversary's strategy. We prove the security of our protocols in the ideal world/real world simulation paradigm, the details appear in §A.1.

In the SOC setting, the four servers execute our protocol. For client-server based computation, a client secret-shares its data with the servers. Servers perform the required operations on secret-shared data and obtain the secret-shared output. Finally, to provide the client's output, servers reconstruct the output towards it. The underlying assumption here is that the corrupt server can collude with a corrupt client. We consider computation over $\mathbb{Z}_{2^\lambda}$ and $\mathbb{Z}_{2^1}$. To deal with decimal values, we use Fixed-Point Arithmetic (FPA) [66,64,25,71] in which a value is represented as a λ-bit integer in signed 2's complement representation. The most significant bit (msb) denotes the sign bit, and d least significant bits are reserved for the fractional part. The λ-bit integer is then viewed as an element of $\mathbb{Z}_{2^\lambda}$, and operations are performed modulo 2^λ. We set $\lambda = 64, d = 13$, leaving $\lambda - d - 1$ bits for the integer part. Our protocols are cast in the preprocessing paradigm, wherein a protocol is divided into (a) function dependent (input independent) *preprocessing phase* and (b) input dependent *online phase*.

Notation 1 *Wherever necessary, we denote $\mathcal{P}$ by the unordered set $\{P_i, P_j, P_k, P_m\}$ and $\{P_i, P_{i+1}, P_{i+2}, P_{i+3}\}$. Note that $i, j, k, m \in [4]$ do not correspond to any fixed ordering, only constraint being $i \neq j \neq k \neq m$. Similarly for $i, i+1, i+2, i+3$, corresponding to a P_i, say P_2, $P_{i+1} = P_3$, $P_{i+2} = P_4$, $P_{i+3} = P_1$.*

Standard Building Blocks. Parties make use of a one-time key setup captured by functionality $\mathcal{F}_{\mathsf{setup}}$ (Fig. 7), to establish pre-shared random keys for pseudo-random functions (PRF) among them. This functionality incurs a one-time cost, and thus can be instantiated using any `FaF`-secure protocol such as that of [3].

We make use of a *collision-resistant* hash function H and a commitment scheme Com. Details appear in §A.2.

Advanced Building Blocks. Here we discuss 4 primitives at a high-level: (a) 3-party joint message passing (jmp) from [58], with minor modifications (b) a related 4-party jmp primitive, (c) oblivious product evaluation (OPE) and (d) distributed zero-knowledge protocol. The details, including functionalities, protocols and proofs are deferred to §A.2.

3-Party Joint Message Passing (jmp3). The jmp primitive from [58] allows two parties P_i, P_j holding a common value v, to send it to a party P_k such that either P_k receives the correct v, or TTP is identified. For our purpose, we trivially modify their protocol to give out a dispute pair (DP) instead of a TTP to all the 4 parties. In [58], the jmp primitive is invoked for sending each value independently and the verification is amortized over many sends. Their protocol allows for such a decoupling due to its asymmetry and a pre-specified order of verification. For our protocol however, postponing verification causes security issues. Specifically, batching the verification of different layers of the circuit together allows an adversary to follow a strategy which ensures that DP comprises of two (semi) honest parties. This is contrary to the requirement that DP must include the malicious party. To avoid this problem, we compress the send and verification of jmp so that an optimistic (no error) run takes one round and batch them together for many instances corresponding to a pair of senders. That is, a pair of parties, say P_i, P_j invoke jmp to send a vector $\overrightarrow{v}$ to P_k, and in parallel verification of correctness takes place. We call the modified variant as jmp3. It requires an amortized communication of 1 element.

4-Party Joint Message Passing (jmp4). Similar to jmp3, jmp4 allows two parties P_i, P_j holding a common value v, to send it to the other two parties P_k, P_m such that, either both the parties receive the correct v or all the parties identify DP.

Notation 2 *We refer to the invocation of* jmp3(P_i, P_j, v, P_k) *as "*P_i, P_j jmp3-send v *to* P_k*" and* jmp4(P_i, P_j, v, P_k, P_m) *as "*P_i, P_j jmp4-send v *to* P_k, P_m*".*

Oblivious Product Evaluation (OPE). OPE (adapted from [55]) allows two parties holding $x \in \mathbb{Z}_{2^\lambda}$ and $y, z \in \mathbb{Z}_{2^\lambda}$ respectively, to compute an additive sharing of the product xy, such that one party holds $xy + z \in \mathbb{Z}_{2^\lambda}$ and the other holds $z \in \mathbb{Z}_{2^\lambda}$. We rely on techniques from [43,55] to obtain an OPE for λ-bit strings by running a total of λ 1-out-of-2 OTs on λ bits strings (see §A.2). In this work, we instantiate OTs using the protocol from Ferret [76], which incurs an (amortized) cost of 0.44 bits for generating one random correlated OT (amortized over batch generation of 10^7 correlated OTs). We can obtain an input-dependent OT (using techniques from [12,50]) at an additional cost of 2 elements and 1 bit. This results in a cost of $2\lambda + 1.44$ bits per OT. So an instantiation of OPE requires an amortised cost of $\lambda(2\lambda + 1.44)$ bits and 4 rounds. Note that we use OT in a black-box manner; thus, any improvement in OT, will improve the efficiency of our construction. Further, although OPE can be realised with oblivious linear evaluation (OLE), we opt for the approach of [55] due to better efficiency of

OT. Hence, any improvements in OLE that surpasses OT can be translated to improving our protocol by replacing OPE with OLE.

Distributed Zero-knowledge (ZK). To verify a party P_i's correct behaviour, we extend the distributed zero-knowledge proofs introduced first in [18] offering *abort* security, and further optimized by Boyle *et al.* [20] to provide *robust* verification of degree-two relations. Such proofs involve a single prover and multiple verifiers, where the prover intends to prove the correctness of its (degree-two) computation over data which is *additively* distributed among the verifiers. In [20], the authors provide a distributed ZK protocol with sub-linear proof size, which is adapted for the verification of messages sent in a 3PC protocol with one corruption. Their ZK protocol extends in a straightforward manner to the 4-party case with one malicious corruption and one semi-honest corruption in the FaF model where a dispute pair is identified in case the verification fails. This is identical to extending the distributed ZK protocol to the case of 4 parties with 1 malicious corruption in the classical model and does not incur any overhead in our setting. Since the protocol in [20], and correspondingly ours, is constructed over fields, to support verification over rings, as in [20] verification operations are carried out on the extended ring $\mathbb{Z}_{2^\lambda}/f(x)$, which is the ring of all polynomials with coefficients in $\mathbb{Z}_{2^\lambda}$ modulo a polynomial f, of degree η, irreducible over $\mathbb{Z}_{2^1}$. Each element in $\mathbb{Z}_{2^\lambda}$ is lifted to a η-degree polynomial in $\mathbb{Z}_{2^\lambda}[x]/f(x)$ (which results in blowing up the communication by a factor η).

3 Necessity of Oblivious Transfer

Here, we show that semi-honest OT is necessary for a FaF-secure protocol. Our claim holds for $n \leq 2t + 2h^*$ which subsumes the case of n-party (t, h^*)-FaF security with optimal threshold of $t + h^* + 1$ and $2t + h^* + 1$ for abort and GOD [3] respectively, and the special case of 4-party $(1, 1)$-FaF security. The theorem and proof sketch are given below.

Theorem 3. *An n-party (t, h^*)-FaF secure (abort) protocol with $n \leq 2t + 2h^*$ implies 2-party semi-honest OT.*

Proof. Without loss of generality, we consider $n = 2t+2h^*$. Let π_f be an n-party (t, h^*)-FaF secure abort protocol for computing the function $f((m_0, m_1), \perp, \ldots, \perp, b) = (\perp, \perp, \ldots, \perp, m_b)$. We construct a 2-party semi-honest OT protocol π_{OT} (Fig. 15) between a sender P_S with inputs (m_0, m_1) and a receiver P_R with input b using π_f. In π_{OT}, P_S emulates the role of $Q_S = \{P_1, P_2, \ldots P_{t+h^*}\}$ while P_R emulates the role of $Q_R = \{P_{t+h^*+1}, \ldots, P_n\}$ to run π_f. P_R outputs the same m_b as output by party P_n which it emulates while P_S outputs $\perp$. To prove the security of π_{OT}, we construct simulators $\mathcal{S}_S$ and $\mathcal{S}_R$ that generate the view of P_S and P_R respectively from their inputs.

Let P_S be corrupted by the semi-honest adversary $\mathcal{A}_{\mathsf{OT}}$ and let $H = \{P_1, \ldots, P_{h^*}\}$ and $I = Q_S \backslash H$. We now map $\mathcal{A}_{\mathsf{OT}}$ to an adversarial strategy against π_f as follows. Consider a malicious adversary $\mathcal{A}$ for π_f that corrupts parties in I but does not deviate from the protocol (since $\mathcal{A}_{\mathsf{OT}}$ is semi-honest). However, it sends the random tape, inputs and messages of all parties in I to every other party in H at the end of the protocol execution. Note that such an attack of leaking the view of the maliciously corrupted parties to the semi-honest adversary is valid in the FaF model. The semi-honest adversary $\mathcal{A}_{\mathcal{H}}$ for π_f runs $\mathcal{A}_{\mathsf{OT}}$ on the joint view of the parties in $I \cup H$ ($\mathcal{A}_{\mathcal{H}}$ receives the view of parties in I from $\mathcal{A}$) and outputs the same value as $\mathcal{A}_{\mathsf{OT}}$. Since $|I| = t$ and $|H| = h^*$, the security of π_f ensures that there exist simulators $\mathcal{S}_{\mathcal{A}}$ and $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ corresponding to the adversaries $\mathcal{A}$ and $\mathcal{A}_{\mathcal{H}}$. We construct the simulator $\mathcal{S}_S$ (Fig. 16) to run $\mathcal{S}_{\mathcal{A}}$ followed by $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ on P_S's input (m_0, m_1) and output the view generated by $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$. Since $\mathcal{A}_{\mathcal{H}}$ receives the view of parties in I, the view generated by $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ includes the view of parties in $I \cup H$. Note that although $\mathcal{A}$ considered is malicious in π_f, it is emulated by a semi-honest adversary in the outer π_{OT} protocol and hence does not deviate from the protocol. Corresponding to such adversarial strategy of $\mathcal{A}$, the simulator $\mathcal{S}_{\mathcal{A}}$ may need to choose the input on behalf of $\mathcal{A}$. A simulator for a semi-honest adversary is not allowed to choose the input on behalf of the adversary, as discussed in [48]. However, since the parties in I controlled by the adversary $\mathcal{A}$ do not have inputs for f, this does not pose a problem in the proof. $\mathcal{S}_S$ can thus use $\mathcal{S}_{\mathcal{A}}$ without any issues.

This proves the necessity of semi-honest-OT for (t, h^*)-FaF secure protocol where $t+h^* < n \leq 2t+2h^*$. Moreover, the sufficiency of OT for the same is given in [3, Theorem 4.1]. The detailed constructions of the OT protocol, simulators and the corresponding indistinguishabilty argument appears in §B.

Corollary 1. *An n-party (t, h^*)-FaF secure abort protocol with $n = t + h^* + 1$ implies 2-party semi-honest* OT.

Corollary 2. *An n-party (t, h^*)-FaF secure GOD protocol with $n = 2t + h^* + 1$ implies 2-party semi-honest* OT.

Both Corollary 1 and 2 follow directly from Theorem 3. For Corollary 1, the sender emulates $t + h^*$ parties and the receiver emulates 1 party. For the corrupt receiver we consider $I = \phi$ and $H = \{P_n\}$. For Corollary 2, the sender emulates $t + h^*$ parties and the receiver emulates $t + 1$ parties. For the corrupt receiver we consider $I = \{P_{t+h^*+1}, \ldots, P_{2t+h^*}\}$ and $H = \{P_n\}$.

4 Input Sharing and Reconstruction

To enforce security, we perform computation on secret-shared data. This section starts with the various sharing semantics we use, followed by a sharing and a reconstruction protocol for secret-shared computation. We further present an efficient batch reconstruction for a second type of sharing, which in turn, will act as the primary building block for our efficient (batch) multiplication protocol.

We begin with the motivation for the choice of our sharing semantics. As explained earlier, we rely on RSS with threshold 2 to tackle view-leakage attack where the semi-honest adversary may receive the view of the malicious adversary. Instead of using RSS directly, we slightly augment our sharing to RSS-share a random mask and make the masked secret available to all. This sharing style makes the online cost of a multiplication one reconstruction instead of two. If we use RSS directly for sharing a secret, then relying on the Beaver's multiplication triple technique [11], we would need reconstructing $\mathsf{x}+\alpha_\mathsf{x}$ and $\mathsf{y}+\alpha_\mathsf{y}$, where x,y are the inputs and $\alpha_\mathsf{x}, \alpha_\mathsf{y}$ are the corresponding random masks. However, as per the latter sharing, we include the masked values $\beta_\mathsf{x} = \mathsf{x}+\alpha_\mathsf{x}$, $\beta_\mathsf{y} = \mathsf{y}+\alpha_\mathsf{y}$ along with RSS shares of α_x and α_y respectively in our sharing semantics. So the only reconstruction needed now is that of the masked valued of xy. This idea goes back to [70]. We now describe the sharing semantics.

1. $[\cdot]$-*sharing:* A value $\mathsf{v} \in \mathbb{Z}_{2^\lambda}$ is said to be $[\cdot]$-shared (additively shared) among parties P_i, P_j, if P_i holds $[\mathsf{v}]_i \in \mathbb{Z}_{2^\lambda}$ and P_j holds $[\mathsf{v}]_j \in \mathbb{Z}_{2^\lambda}$ such that $\mathsf{v} = [\mathsf{v}]_i + [\mathsf{v}]_j$.
2. $\langle\cdot\rangle$-*sharing:* A value $\mathsf{v} \in \mathbb{Z}_{2^\lambda}$ is said to be $\langle\cdot\rangle$-shared among $\mathcal{P}$ if, each pair of parties (P_i, P_j), where $1 \le i < j \le 4$, holds $\langle\mathsf{v}\rangle_{ij} \in \mathbb{Z}_{2^\lambda}$ such that $\mathsf{v} = \sum_{(i,j)} \langle\mathsf{v}\rangle_{ij}$. This is equivalent to RSS of a value among 4 parties with threshold 2. Note that since $\langle\mathsf{v}\rangle_{ij}$ represents the common share held by P_i, P_j, throughout the protocol we assume the invariant that $\langle\mathsf{v}\rangle_{ij} = \langle\mathsf{v}\rangle_{ji}$, for all $1 \le i < j \le 4$. $\langle\mathsf{v}\rangle_i$ denotes P_i's share in the $\langle\cdot\rangle$-sharing of v.
3. $[\![\cdot]\!]$-*sharing:* A value $\mathsf{v} \in \mathbb{Z}_{2^\lambda}$ is $[\![\cdot]\!]$-shared if
   - there exists $\alpha_\mathsf{v} \in \mathbb{Z}_{2^\lambda}$ that is $\langle\cdot\rangle$-shared amongst $\mathcal{P}$ and
   - each $P_i \in \mathcal{P}$ holds $\beta_\mathsf{v} = \mathsf{v} + \alpha_\mathsf{v}$.

   Note that the value α_v acts as the mask for v. We denote by $[\![\mathsf{v}]\!]_i$, P_i's share in the $[\![\cdot]\!]$-sharing of v.

Note that all these sharings are linear i.e. given sharings of values $a_1, \ldots, a_m$ and public constants $c_1, \ldots, c_m$, sharing of $\sum_{i=1}^m c_i a_i$ can be computed non-interactively for an integer m.

4.1 $[\![\cdot]\!]$-sharing: Sharing and Reconstruction

Sharing. Protocol $[\![\cdot]\!]$-Sh either allows a party P_s to share a value v or allows for a dispute pair (DP) detection. To enable P_s to generate $[\![\mathsf{v}]\!]$, in the preprocessing phase, P_s together with every other party P_i, samples a random $\langle\alpha_\mathsf{v}\rangle_{si} \in \mathbb{Z}_{2^\lambda}$, while P_s samples a random $\langle\alpha_\mathsf{v}\rangle_{ij} \in \mathbb{Z}_{2^\lambda}$ with every pair of parties P_i, P_j. This allows P_s to learn α_v in clear. In the online phase, P_s computes $\beta_\mathsf{v} = \mathsf{v} + \alpha_\mathsf{v}$ and sends it to P_t. Parties P_s, P_t then use jmp4-send to send β_v to the rest. This step either allows the sharing to be completed or allows for DP detection. The protocol appears in Fig. 1.

Protocol $[\![\cdot]\!]$-Sh

- **Input, Output:** P_s has v. The parties output $[\![\mathsf{v}]\!]$.
- **Primitives:** jmp4-send (§2).

Preprocessing: P_s together with (a) P_i, for each $P_i \in \mathcal{P} \setminus P_s$ samples random $\langle\alpha_\mathsf{v}\rangle_{si} \in \mathbb{Z}_{2^\lambda}$; (b) $P_i, P_j \in \mathcal{P} \setminus P_s$, where $i \ne j$, samples random $\langle\alpha_\mathsf{v}\rangle_{ij} \in \mathbb{Z}_{2^\lambda}$.

Online: P_s computes $\beta_{\mathsf{v}} = \mathsf{v} + \sum_{(i,j)} \langle \alpha_{\mathsf{v}} \rangle_{ij}$ and sends it to P_t, where $s \neq t$. P_s, P_t jmp4-send β_{v} to $\mathcal{P} \backslash \{P_s, P_t\}$.

Fig. 1: $[\![\cdot]\!]$-sharing a value

Reconstruction. Protocol $[\![\cdot]\!]$-Rec allows parties to reconstruct v from $[\![\mathsf{v}]\!]$ such that either v is obtained by all the parties or a DP is identified. As observed, a party misses three shares of $\langle \alpha_{\mathsf{v}} \rangle$, which are needed for reconstructing v, each of which is held by two other parties. To reconstruct v towards a party P_s, in the preprocessing each pair (P_i, P_j) jmp3-send a commitment of their common share $\mathsf{Com}(\langle \alpha_{\mathsf{v}} \rangle_{ij})$ to P_s. The common source of randomness (generated via the shared key setup) can be used for generating the commitments, so that it is identically generated by both the senders. Then in the online phase all the parties open the commitments sent in the preprocessing phase. P_s first reconstructs α_{v} from the consistent openings and then computes $\mathsf{v} = \beta_{\mathsf{v}} - \alpha_{\mathsf{v}}$. Due to the use of jmp3, the preprocessing may fail, however once it is successful the online phase is robust. Due to this feature, this reconstruction ensures fairness i.e. either all or none receives the output (in the latter case DP has been identified). In case the reconstruction protocol terminates with a dispute pair, to extend security to GOD, parties perform the circuit evaluation using a semi-honest 2PC protocol.

Protocol $[\![\cdot]\!]$-Rec

- **Input, Output:** The parties input $[\![\mathsf{v}]\!]$. The parties output v.
- **Primitives:** jmp4-send and Com (§2).

Preprocessing: Each P_i, P_j, $1 \leq i < j \leq 4$ compute $\mathsf{Com}(\langle \alpha_{\mathsf{v}} \rangle_{ij})$ and jmp4-send it to P_k, P_m.

Online: Each P_i, P_j, $1 \leq i < j \leq 4$ open $\mathsf{Com}(\langle \alpha_{\mathsf{v}} \rangle_{ij})$ to P_k and P_m. Each P_i accepts the opening consistent with the commitment received earlier and computes $\mathsf{v} = \beta_{\mathsf{v}} - \sum_{(i,j)} \langle \alpha_{\mathsf{v}} \rangle_{ij}$.

Fig. 2: Reconstructing a $[\![\cdot]\!]$-shared value

4.2 $\langle \cdot \rangle$-sharing: Reconstruction

In our MPC protocol, for each multiplication gate we require to reconstruct a $\langle \cdot \rangle$-shared value in the online phase. Note that a party misses three shares of $\langle \mathsf{v} \rangle$ needed for reconstruction, each of which is held by two other parties. For reconstructing v towards all the parties, naively, each pair can jmp4-send their common share to the other two parties. This requires 6 invocations of jmp4, thus a communication of 12 elements. Since reconstructing $\langle \cdot \rangle$-shared value is the only communication bottleneck in the online phase of our multiplication protocol, it is imperative to improve its efficiency.

Taking a step towards this, we allow two parties, say P_3, P_4 (w.l.o.g) to first reconstruct v and use jmp4-send to send it to the other two parties. Naively, the reconstruction towards P_3, P_4 requires 6 instances of $\mathsf{jmp3\text{-}send}$, three per party to send its missing shares. To improve the communication cost further, we improve the cost of the second instance of the reconstruction of v (towards P_4 in our case), to 2 $\mathsf{jmp3\text{-}send}$ instances, leveraging the communication already done for the reconstruction towards P_3. This reduces the communication cost to 7 elements. Our protocol appears in Fig. 3.

Since $\mathsf{jmp3}$ is defined for a vector of values, in $\langle\cdot\rangle$-Rec, parties execute reconstruction of multiple values together. The protocol is described for a single value. Extending it to a vector is straightforward. In our multiplication protocol, this translates to reconstruction of the output of all multiplication gates in a level of the circuit simultaneously.

Note that we can reconstruct v from $[\![\mathsf{v}]\!]$ using $\langle\cdot\rangle$-Rec to reconstruct α_{v}. However, while $[\![\cdot]\!]$-Rec offers fairness, $\langle\cdot\rangle$-Rec does not. This implies if we use $\langle\cdot\rangle$-Rec for the final output, it is possible that the adversary gets the output while the honest parties do not. Further, when the computation is rerun in 2PC mode, the adversary can use a different input and obtain another evaluation, thus breaching security.

Protocol $\langle\cdot\rangle$-Rec

- **Input, Output:** The parties input $\langle\mathsf{v}\rangle$. The parties output v.
- **Primitives:** $\mathsf{jmp3\text{-}send}$ and $\mathsf{jmp4\text{-}send}$ (§2).

Online:

– **(Reconstructing v to P_3.)** P_1, P_2 $\mathsf{jmp3\text{-}send}$ $\langle\mathsf{v}\rangle_{12}$ to P_3. P_1, P_4 $\mathsf{jmp3\text{-}send}$ $\langle\mathsf{v}\rangle_{14}$ to P_3. P_2, P_4 $\mathsf{jmp3\text{-}send}$ $\langle\mathsf{v}\rangle_{24}$ to P_3.

– **(Reconstructing v to P_4.)** P_1, P_3 $\mathsf{jmp3\text{-}send}$ $\langle\mathsf{v}\rangle_{13}$ to P_4. P_2, P_3 $\mathsf{jmp3\text{-}send}$ $\langle\mathsf{v}\rangle_{12} + \langle\mathsf{v}\rangle_{23}$ to P_4.

– **(Reconstructing v to P_1, P_2.)** P_3, P_4 $\mathsf{jmp4\text{-}send}$ $\mathsf{v} = \sum_{(i,j)} \langle\mathsf{v}\rangle_{ij}$ to P_1, P_2.

Fig. 3: Reconstructing a $\langle\cdot\rangle$-shared value

5 Multiplication

In this section, we present a multiplication protocol. Taking a top-down approach, we first present our multiplication protocol relying on a triple generation protocol in a black-box way. We then conclude with a triple generation protocol. To gain efficiency, several layers of amortisation are used. We mention them on the go and summarise at the end of the section.

5.1 Multiplication Protocol

The multiplication protocol (Fig. 4) allows parties to compute $[\![\mathsf{z}]\!]$, given $[\![\mathsf{x}]\!]$ and $[\![\mathsf{y}]\!]$, where $\mathsf{z} = \mathsf{x} \cdot \mathsf{y}$. We reduce this problem to that of reconstructing a $\langle\cdot\rangle$-shared value, assuming that the parties have access to (a) $\langle\cdot\rangle$-sharing of a multiplication triple $(\alpha_{\mathsf{x}}, \alpha_{\mathsf{y}}, \alpha_{\mathsf{x}}\alpha_{\mathsf{y}})$ for random α_{x}, α_{y} and (b) $\langle\cdot\rangle$-sharing of a random α_{z}. Both the requirements are input (i.e. x, y) independent and can be fulfilled during the preprocessing phase. The former requirement is obtained via a triple generation protocol tripGen (Fig. 6), discussed subsequently. The latter requirement can be achieved non-interactively using the shared key setup. The reduction works as follows. The random and independent secret α_{z} is taken as the mask for the $[\![\cdot]\!]$-sharing of product z. Since α_{z} is already $\langle\cdot\rangle$-shared, to complete $[\![\mathsf{z}]\!]$, parties only need to obtain the masked value $\beta_{\mathsf{z}} = \mathsf{z} + \alpha_{\mathsf{z}}$. Since β_{z} takes the following form $\beta_{\mathsf{z}} = \mathsf{z} + \alpha_{\mathsf{z}} = \mathsf{xy} + \alpha_{\mathsf{z}} = (\beta_{\mathsf{x}} - \alpha_{\mathsf{x}})(\beta_{\mathsf{y}} - \alpha_{\mathsf{y}}) + \alpha_{\mathsf{z}} = \beta_{\mathsf{x}}\beta_{\mathsf{y}} - \beta_{\mathsf{x}}\alpha_{\mathsf{y}} - \beta_{\mathsf{y}}\alpha_{\mathsf{x}} + \alpha_{\mathsf{x}}\alpha_{\mathsf{y}} + \alpha_{\mathsf{z}}$ and the parties hold $\langle\alpha_{\mathsf{x}}\rangle$, $\langle\alpha_{\mathsf{y}}\rangle$, $\langle\alpha_{\mathsf{x}}\alpha_{\mathsf{y}}\rangle$, $\langle\alpha_{\mathsf{z}}\rangle$, and $\beta_{\mathsf{x}}, \beta_{\mathsf{y}}$ in clear, the parties hold $\langle\beta_{\mathsf{z}}\rangle$. Parties thus need to reconstruct β_{z}. In order to leverage the amortised cost of $\langle\cdot\rangle$-Rec, we batch many multiplications together. While for simplicity, we present the protocol in Fig. 4 for a single multiplication, our complexity analysis accounts for amortization.

Protocol mult

- **Input and Output:** The input is $[\![\mathsf{x}]\!], [\![\mathsf{y}]\!]$ and the output is $[\![\mathsf{xy}]\!]$.
- **Primitives:** tripGen (§5.2; Fig. 6) and $\langle\cdot\rangle$-Rec (§4.2; Fig. 3).

Preprocessing:

– Each P_i, P_j where $1 \le i < j \le 4$ sample random $\langle\alpha_{\mathsf{z}}\rangle_{ij} \in \mathbb{Z}_{2^\lambda}$.

– Parties invoke tripGen with inputs $\langle\alpha_{\mathsf{x}}\rangle, \langle\alpha_{\mathsf{y}}\rangle$ to obtain $\langle\alpha_{\mathsf{x}}\alpha_{\mathsf{y}}\rangle$.

Online:

– Each P_i, P_j for $1 \le i < j \le 4$ and $(i,j) \ne (1,2)$ compute $\langle\beta_{\mathsf{z}}\rangle_{ij}$ such that $\langle\beta_{\mathsf{z}}\rangle_{ij} = -\beta_{\mathsf{x}}\langle\alpha_{\mathsf{y}}\rangle_{ij} - \beta_{\mathsf{y}}\langle\alpha_{\mathsf{x}}\rangle_{ij} + \langle\alpha_{\mathsf{x}}\alpha_{\mathsf{y}}\rangle_{ij} + \langle\alpha_{\mathsf{z}}\rangle_{ij}$.

– P_1, P_2 compute $\langle\beta_{\mathsf{z}}\rangle_{12} = \beta_{\mathsf{x}}\beta_{\mathsf{y}} - \beta_{\mathsf{x}}\langle\alpha_{\mathsf{y}}\rangle_{12} - \beta_{\mathsf{y}}\langle\alpha_{\mathsf{x}}\rangle_{12} + \langle\alpha_{\mathsf{x}}\alpha_{\mathsf{y}}\rangle_{12} + \langle\alpha_{\mathsf{z}}\rangle_{12}$.

– Parties invoke $\langle\cdot\rangle$-Rec to obtain β_{z}.

Fig. 4: Multiplication Protocol

5.2 Triple Generation Protocol

As a building block to our triple generation protocol, we first present a distributed multiplication protocol, where two distinct pairs of parties hold inputs to the multiplication and the goal is to additively share the product between the pairs. We build on this protocol to complete our triple generation.

Distributed Multiplication Protocol Let P_i, P_j hold a and P_k, P_m hold b. The goal of a distributed multiplication is to allow P_i, P_j compute c^1 and P_k, P_m to compute c^2 such that $c^1 + c^2 = ab$. To achieve this, P_k and P_m locally sample c^2 (using one-time key setup; Fig. 7) then parties engage in an instance of OPE (§2) where P_i, P_j and respectively P_k, P_m enact the receiver's and sender's role.

– P_i, P_j as the receivers input a and output either c^1 or DP.

– P_k, P_m as the senders input b, $-c^2$ and output either $\perp$ or DP.

Since the pair of receivers $\{P_i, P_j\}$ hold identical inputs and use a shared source of randomness, their corresponding messages in the underlying protocol for OPE realisation will be identical. They send their messages to the senders via an instance of jmp4. Recall that the jmp4 primitive ensures that a message commonly known to two sender parties is either communicated correctly to both the receiving parties, or a dispute pair DP is identified. In the former case, the pair of senders $\{P_k, P_m\}$, having the same input and receiver's message, will prepare identical sender messages as a part of OPE and communicate to the receivers via another instance of jmp4 primitive, resulting in either a successful communication of the sender message to the receivers $\{P_i, P_j\}$ or identification of DP. In the former case, OPE is concluded successfully. Note that the verification of jmp4 tackles any malicious behaviour, thus relying on semi-honest OPE suffices. Otherwise, DP is identified and the pair is guaranteed to include the malicious party. If fairness is the end goal, the protocol can terminate at this stage. Otherwise, it switches to an execution of a semi-honest 2PC (such as ABY2.0 [70]) with the parties outside DP to achieve the stronger guarantee of GOD.

Protocol disMult

- **Input and Output:** P_i, P_j hold a. P_k, P_m hold b. The first pair outputs c^1, the second pair c^2 such that $c^1 + c^2 = ab$. Otherwise the parties output DP.
- **Primitives:** OPE and jmp4 (§2).

– P_k and P_m locally sample a value c^2, using their shared key.
– P_i, P_j execute OPE with input a using jmp4 to send messages to P_k, P_m.
– P_k, P_m execute OPE with inputs $(b, -c^2)$ using jmp4 to send messages to P_i, P_j.

Fig. 5: Distributed Multiplication Protocol

Triple Generation Protocol The triple generation protocol allows parties holding $\langle\alpha_{\mathsf{x}}\rangle, \langle\alpha_{\mathsf{y}}\rangle$ to generate $\langle\alpha_{\mathsf{x}}\alpha_{\mathsf{y}}\rangle$. We write the product $\alpha_{\mathsf{x}}\alpha_{\mathsf{y}}$ as below, consisting of 36 summands, categorizing them into three types as below and as shown in Table 3.

For the summands in type S_0, no single party holds the two constituent shares of $\alpha_{\mathsf{x}}, \alpha_{\mathsf{y}}$. For the summands in S_1, exactly one party holds the two constituent shares, and lastly for the summands in S_2, exactly two parties hold the the two constituent shares. Note that there are 6 summands each, of the types S_0 and S_2 and 24 summands of type S_1. To generate $\langle\alpha_{\mathsf{x}}\alpha_{\mathsf{y}}\rangle$, we generate $\langle\cdot\rangle$-sharing of each summand of $\alpha_{\mathsf{x}}\alpha_{\mathsf{y}}$ and then sum them up to obtain $\langle\alpha_{\mathsf{x}}\alpha_{\mathsf{y}}\rangle$. The task of generating $\langle\cdot\rangle$-sharing for an individual summand differs based on the class it belongs to.

$$\begin{aligned}\alpha_{\mathsf{x}} \cdot \alpha_{\mathsf{y}} &= \sum_{\substack{(i,j)\\1\le i<j\le 4}} \langle \alpha_{\mathsf{x}} \rangle_{ij} \cdot \sum_{\substack{(k,m)\\1\le k<m\le 4}} \langle \alpha_{\mathsf{y}} \rangle_{km} \\ &= \underbrace{\sum_{\substack{(i,j)\\1\le i<j\le 4}} \langle \alpha_{\mathsf{x}} \rangle_{ij} \langle \alpha_{\mathsf{y}} \rangle_{ij}}_{S_2} + \underbrace{\sum_{\substack{(i,j,k)\\i,j,k\in[4]}} \langle \alpha_{\mathsf{x}} \rangle_{ij} \langle \alpha_{\mathsf{y}} \rangle_{ik}}_{S_1} + \underbrace{\sum_{\substack{(i,j),(k,m)\\1\le i,k<j,m\le 4}} \langle \alpha_{\mathsf{x}} \rangle_{ij} \langle \alpha_{\mathsf{y}} \rangle_{km}}_{S_0}\end{aligned} \tag{1}$$

Summands of S_2. Each summand in this type can be computed locally by 2 parties. For instance, $\langle \alpha_{\mathsf{x}} \rangle_{ij} \langle \alpha_{\mathsf{y}} \rangle_{ij}$ can be computed by P_i and P_j. Denoting $\langle \alpha_{\mathsf{x}} \rangle_{ij} \langle \alpha_{\mathsf{y}} \rangle_{ij}$ as τ_{ij}, $\langle \tau_{ij} \rangle$ is computed as follows:

$$\begin{aligned}&P_i, P_j \text{ set } \langle \tau_{ij} \rangle_{ij} = \langle \alpha_{\mathsf{x}} \rangle_{ij} \langle \alpha_{\mathsf{y}} \rangle_{ij} \text{ and} \\ &P_u, P_v \text{ set } \langle \tau_{ij} \rangle_{uv} = 0, \forall (u,v) \neq (i,j)\end{aligned} \tag{2}$$

Summands of S_1. Each summand here can be computed locally by a single party. For instance, $\langle \alpha_{\mathsf{x}} \rangle_{ij} \langle \alpha_{\mathsf{y}} \rangle_{ik}$ can be computed by P_i alone. Then P_i's goal is to share this amongst the four parties so that one share is held by both P_i, P_k and the other by P_j, P_m. That is, for $\delta_i, \delta_i^1, \delta_i^2$ with $\delta_i = \delta_i^1 + \delta_i^2 = \langle \alpha_{\mathsf{x}} \rangle_{ij} \langle \alpha_{\mathsf{y}} \rangle_{ik}$, P_i, P_k intend to obtain δ_i^1 and P_j, P_m intend to obtain δ_i^2. The pairings $\{P_i, P_k\}$ and $\{P_j, P_m\}$ for various parties are done to balance the share count across the parties. We say that $\{P_i, P_k\}$ and respectively $\{P_j, P_m\}$ pair up for P_i's instance. Given this, $\langle \delta_i \rangle$ can be computed as (we set $k = i + 3$):

$$\begin{aligned}&P_i, P_k \text{ set } \langle \delta_i \rangle_{ik} = \delta_i^1,\ P_j, P_m \text{ set } \langle \delta_i \rangle_{jm} = \delta_i^2 \\ &P_u, P_v \text{ set } \langle \delta_i \rangle_{uv} = 0,\ \text{for all } (u,v) \neq (i,k), (j,m)\end{aligned} \tag{3}$$

Now to achieve the above distribution of additive shares (δ_i^1, δ_i^2), P_i, P_j, P_m first locally sample δ_i^2 (using the shared key setup) and further, P_i computes and sends δ_i^1 to P_k. To keep P_i's misbehaviour in check, P_i is made to prove in zero-knowledge the correctness of its computation. With this high-level idea, we introduce two cost-cutting techniques.

	$\langle \alpha_{\mathsf{x}} \rangle_{12}$	$\langle \alpha_{\mathsf{x}} \rangle_{13}$	$\langle \alpha_{\mathsf{x}} \rangle_{14}$	$\langle \alpha_{\mathsf{x}} \rangle_{23}$	$\langle \alpha_{\mathsf{x}} \rangle_{24}$	$\langle \alpha_{\mathsf{x}} \rangle_{34}$
$\langle \alpha_{\mathsf{y}} \rangle_{12}$	S_2	S_1	S_1	S_1	S_1	S_0
$\langle \alpha_{\mathsf{y}} \rangle_{13}$	S_1	S_2	S_1	S_1	S_0	S_1
$\langle \alpha_{\mathsf{y}} \rangle_{14}$	S_1	S_1	S_2	S_0	S_1	S_1
$\langle \alpha_{\mathsf{y}} \rangle_{23}$	S_1	S_1	S_0	S_2	S_1	S_1
$\langle \alpha_{\mathsf{y}} \rangle_{24}$	S_1	S_0	S_1	S_1	S_2	S_1
$\langle \alpha_{\mathsf{y}} \rangle_{34}$	S_0	S_1	S_1	S_1	S_1	S_2

Table 3: The summands of $\alpha_{\mathsf{x}} \cdot \alpha_{\mathsf{y}}$ with category $\{S_0, S_1, S_2\}$

First, recall that there are 24 summands in S_1 and every P_i is capable of locally computing 6 of them. We combine the above procedure for 6 summands together. That is, δ_i^1, δ_i^2 are additive shares of $\delta_i = \sum_{(j,k)} \langle \alpha_{\mathsf{x}} \rangle_{ij} \langle \alpha_{\mathsf{y}} \rangle_{ik}$. This cuts our cost by 1/6th. Next, leveraging the malicious-minority and non-collusion of the malicious and semi-honest adversaries (implied by FaF model), we customise disZK of [20] (see §A; Fig. 14) to prove that $\sum_{(j,k)} \langle \alpha_{\mathsf{x}} \rangle_{ij} \langle \alpha_{\mathsf{y}} \rangle_{ik} - \delta_i^1 - \delta_i^2 = 0$. As per the need of such ZK, each term in the statement is additively shared amongst P_j, P_k, P_m and is possessed in entirety by the prover P_i. For instance,

$\langle\alpha_{\mathsf{x}}\rangle_{ij}$ is additively shared amongst P_j, P_k, P_m with P_j's share as $\langle\alpha_{\mathsf{x}}\rangle_{ij}$ and the shares of the rest set to 0. Similarly for other shares of α_{x} and α_{y}. δ_i^2 is additively shared amongst P_j, P_k, P_m with P_j's share as δ_i^2 and the shares of the rest set to 0. Lastly, δ_i^1 is additively shared amongst P_j, P_k, P_m with P_k's share as δ_i^1 and the shares of the rest set to 0. If the disZK is successful, then P_i, P_k output δ_i^1 and P_j, P_m output δ_i^2, using which $\langle\delta_i\rangle$ an be computed as above. Otherwise, the disZK returns a dispute pair. This is executed for every party's collection of S_1 summands.

Summands of S_0. No single party can compute the summands in this category. For instance, $\langle\alpha_{\mathsf{x}}\rangle_{ij}\langle\alpha_{\mathsf{y}}\rangle_{km}$ cannot be computed by any of the parties locally. We invoke the distributed multiplication protocol $\mathsf{disMult}$ (Fig. 5) for each such term, where the common input of $\{P_i, P_j\}$ and $\{P_k, P_m\}$ are $\langle\alpha_{\mathsf{x}}\rangle_{ij}$ and $\langle\alpha_{\mathsf{y}}\rangle_{km}$ respectively and their respective outputs are $\gamma^1_{ij,km}, \gamma^2_{ij,km}$, in case of success, or a dispute pair. Denoting $\gamma_{ij,km} = \gamma^1_{ij,km} + \gamma^2_{ij,km} = \langle\alpha_{\mathsf{x}}\rangle_{ij}\langle\alpha_{\mathsf{y}}\rangle_{km}$, the parties can now generate $\langle\gamma_{ij,km}\rangle$ as:

$$\begin{aligned} &P_i, P_j \text{ set } \langle\gamma_{ij,km}\rangle_{ij} = \gamma^1_{ij,km},\ P_k, P_m \text{ set } \langle\gamma_{ij,km}\rangle_{km} = \gamma^2_{ij,km} \\ &P_u, P_v \text{ set } \langle\gamma_{ij,km}\rangle_{uv} = 0,\ \text{for all } (u,v) \neq (i,j),(k,m) \end{aligned} \tag{4}$$

Protocol $\mathsf{tripGen}$

- **Input and Output:** The parties input $\langle\alpha_{\mathsf{x}}\rangle, \langle\alpha_{\mathsf{y}}\rangle$. The output is $\langle\alpha_{\mathsf{x}}\alpha_{\mathsf{y}}\rangle$.
- **Primitives:** Protocol $\mathsf{disMult}$ (§5.2) and Protocol disZK (§2).

– For each of the 6 summands of the form $\langle\alpha_{\mathsf{x}}\rangle_{ij}\langle\alpha_{\mathsf{y}}\rangle_{km}$ for unordered pairs $\{P_i, P_j\}$ and $\{P_k, P_m\}$ in S_0, the parties execute $\mathsf{disMult}$ with the inputs of $\{P_i, P_j\}$, $\{P_k, P_m\}$ as $\langle\alpha_{\mathsf{x}}\rangle_{ij}$ and $\langle\alpha_{\mathsf{y}}\rangle_{km}$ respectively. The parties either output $\mathtt{DP}$ or $\{P_i, P_j\}$, $\{P_k, P_m\}$ output $\gamma^1_{ij,km}$ and $\gamma^2_{ij,km}$ respectively. In the latter case, parties compute $\langle\gamma_{ij,km}\rangle$ as shown in Equation 4.

– For every i, consider *all* the 6 summands of the form $\langle\alpha_{\mathsf{x}}\rangle_{ij}\langle\alpha_{\mathsf{y}}\rangle_{ik}$ for unordered pairs $\{P_i, P_j\}$ and $\{P_i, P_k\}$ in S_1.

1. The parties P_i, P_j, P_m locally sample δ_i^2 (using the shared key setup).
2. P_i computes and sends $\delta_i^1 = \sum_{(j,k)}\langle\alpha_{\mathsf{x}}\rangle_{ij}\cdot\langle\alpha_{\mathsf{y}}\rangle_{ik} - \delta_i^2$ to P_k.
3. Parties invoke disZK to verify if $\sum_{(j,k)}\langle\alpha_{\mathsf{x}}\rangle_{ij}\langle\alpha_{\mathsf{y}}\rangle_{ik} - \delta_i^1 - \delta_i^2 = 0$. If disZK returns success, then P_i, P_j, P_k, P_m output $\langle\delta_i\rangle$ as shown in Equation 3. Otherwise, output the $\mathtt{DP}$ returned by disZK.

– For each of the 6 summands of S_2, of the form $\langle\alpha_{\mathsf{x}}\rangle_{ij}\langle\alpha_{\mathsf{y}}\rangle_{ij}$, parties compute $\langle\tau_{ij}\rangle$-sharing as shown in Equation 2.

– Every P_r for every $s \neq r$ computes

$$\langle\alpha_{\mathsf{x}}\alpha_{\mathsf{y}}\rangle_{rs} = \sum_{u,v:u\neq v}\langle\tau_{u,v}\rangle_{rs} + \sum_{1\leq\ell\leq 4}\langle\delta_\ell\rangle_{rs} + \sum_{\substack{u,v:u\neq v\\ p,q:p\neq q}}\langle\gamma_{uv,pq}\rangle_{rs}$$

Fig. 6: Triple Generation Protocol

5.3 Summary

Amortizations We summarise the various layers of amortization we use to get the best efficiency of our protocols. First, given a circuit with ℓ multiplication gates, the triple generation protocol creates $\langle\cdot\rangle$-sharing of ℓ triples at one go. All the summands of the form $\langle\alpha_{\mathsf{x}}\rangle_{ij}\langle\alpha_{\mathsf{y}}\rangle_{km}$ from S_0 category across all the ℓ instances use jmp4 for communication, whose verification is inherently batched for amortization. Next, the distributed ZK used for tackling the summands in S_1 can be used in an amortized sense as well. Recall that corresponding to a single triple generation, every P_i runs a single instance of distributed ZK to tackle 6 summands in its possession. However, we can extend this to accommodate 6ℓ summands across all the ℓ triples to achieve 40 bits of statistical security while working over a ring, by performing verification on the extended ring [20,1]. This means that we need to run overall 4 distributed ZK, one for every party. These cover all the amortizations done in the triple sharing protocol which constitutes the preprocessing of the multiplication protocol. The online phase of the multiplication protocol too exploits amortization of the batch $\langle\cdot\rangle$-reconstruction protocol. In the MPC protocol, we thus proceed level by level and execute all the multiplications placed in a level at one go.

Achieving Fairness. To obtain fairness, we can stop immediately after sensing a dispute. This means, in some cases, the effort needed for identifying a dispute pair, beyond sensing a dispute (which only says something is wrong and nothing beyond), can be slashed. For instance, in jmp4 parties can terminate immediately upon detecting conflict without identifying a dispute pair.

6 $(1,1)$-FaF Secure 4PC Protocol

Our complete protocol for evaluating a circuit in the $(1,1)$-FaF security model with fairness and GOD is described here as a composition of the protocols discussed so far and appears in Fig. 34. Recall that our protocol is cast in the preprocessing paradigm with a function dependent preprocessing phase and an online phase. In the preprocessing phase, for each input gate u, parties execute the preprocessing of $[\![\cdot]\!]$-Sh to precompute $\langle\alpha_{\mathsf{u}}\rangle$. Further, for each multiplication gate with input wires u,v and output wire w, parties run the preprocessing of mult to obtain $\langle\alpha_{\mathsf{w}}\rangle$ and $\langle\alpha_{\mathsf{u}}\alpha_{\mathsf{v}}\rangle$ corresponding to the output. This computation is done in parallel for all the multiplication gates. Finally, for each output gate of the circuit, parties execute the preprocessing phase of $[\![\cdot]\!]$-Rec. This completes the preprocessing.

In the online phase, parties evaluate the circuit gate-by-gate in the predetermined topological order. For each input gate u, they execute the online phase of protocol $[\![\cdot]\!]$-Sh to obtain β_{u}. Addition gates are performed locally. For each multiplication gate with input wires u,v and output wire w, parties perform the online phase of mult to compute β_{w}. Finally, parties reconstruct the value of an output wire w, by invoking the online phase of $[\![\cdot]\!]$-Rec. Recall that, as mentioned in §2, we batch the verification of all the parallel instances of jmp3 and jmp4 respectively for every pair of parties, and perform it simultaneously with the send in the same round. In case of malicious behaviour in these instances, additionally at most 2 rounds are required to identify a dispute pair (§A.2). The above protocol either succeeds or a dispute pair is identified, which includes the malicious party. This construction achieves fairness.

To attain GOD without incurring additional overhead in the online phase, we follow the approach of segmented evaluation described in [32]. Specifically, we divide the circuit into segments, and the protocol proceeds as described in a segment-by-segment manner with topological order. As in the case of our fair protocol, either the execution of a segment completes successfully, or a dispute pair is identified. In the latter case, the segment where the fault occurs and all the segments following it are evaluated using a semi-honest 2PC, which is executed by the parties outside the dispute pair. Using this approach, only the segment where the fault occurs incurs the cost of 2PC in addition to the cost of our fair protocol. Hence, this overhead which is limited to a single segment is insignificant. The cost of evaluating the subsequent segments is solely that of the semi-honest 2PC which we instantiate with [70]. Note that in segmented evaluation of the circuit, the output of a segment acts as the input to the following segment. Hence, rerunning the segment where malicious behaviour was detected requires the outputs from the prior segment with 4PC sharing semantics to be translated to 2PC sharing semantics. However, due to a threshold of 2 in the 4PC, no pair of parties hold all the components of sharing corresponding to any secret. This necessitates interaction among parties. Suppose S_m is the segment where malicious activity is detected and w.l.o.g. $\{P_3, P_4\}$ is identified as the dispute pair, which means the evaluation till segment S_{m-1} happened correctly. W.l.o.g let z be the output of the segment S_{m-1} which is also an input to the segment S_m. Since the evaluation of S_{m-1} happened correctly, all 4 parties have the correct $[\![\cdot]\!]$ sharing of z, which comprises of β_{z} and $\langle\alpha_{\mathsf{z}}\rangle$. But to rerun the segment with $\{P_1, P_2\}$, they need the 2PC sharing of z. However, $\{P_1, P_2\}$ miss the $\langle\alpha_{\mathsf{z}}\rangle_{34}$ component which is common to P_3, P_4 and hence cannot obtain the 2PC sharing of z from locally. Making P_3, P_4 send this value directly to P_1 or P_2 or both does not suffice. Since either P_3 or P_4 is malicious, the malicious party can send a wrong value which will lead to an inconclusive state for $\{P_1, P_2\}$, thus failing to achieve the end goal of 2PC sharing. To address the above problem, we resort to the same idea as that of $[\![\cdot]\!]$-Rec. That is, for each output wire z of all the segments, all pairs of parties P_i, P_j commit to their common share $\langle\alpha_{\mathsf{z}}\rangle_{ij}$ in the preprocessing phase and jmp4-send the commitment to the other two parties. Now with the commitments established, parties in the dispute pair can send the opening corresponding to their respective commitments to the remaining two parties. In the above example, this corresponds to P_3, P_4 sending the opening of their commitments which contains $\langle\alpha_{\mathsf{z}}\rangle_{34}$ to P_1, P_2. Following this, P_1, P_2 can decide the correct value of $\langle\alpha_{\mathsf{z}}\rangle_{34}$ based on a valid opening, which is guaranteed to exist since one of P_3, P_4 is honest. Note that, sending the value $\langle\alpha_{\mathsf{z}}\rangle_{34}$ does not breach privacy since the malicious party can anyway send this value to any other party as a part of view-leakage, which is handled by our sharing semantics.

Now P_1 sets its 2PC additive share $[\alpha_{\mathsf{z}}]_1 = \langle \alpha_{\mathsf{z}} \rangle_{12} + \langle \alpha_{\mathsf{z}} \rangle_{13} + \langle \alpha_{\mathsf{z}} \rangle_{14}$ and P_2 sets $[\alpha_{\mathsf{z}}]_2 = \langle \alpha_{\mathsf{z}} \rangle_{23} + \langle \alpha_{\mathsf{z}} \rangle_{24} + \langle \alpha_{\mathsf{z}} \rangle_{34}$, where $\alpha_{\mathsf{z}} = [\alpha_{\mathsf{z}}]_1 + [\alpha_{\mathsf{z}}]_2$. Note that $(\beta_{\mathsf{z}}, [\alpha_{\mathsf{z}}]_1)$ and $(\beta_{\mathsf{z}}, [\alpha_{\mathsf{z}}]_2)$ is a valid 2PC sharing of z as per the semantics of [70]. However, as we describe below, this does not suffice to execute the 2PC.

Observe that the preprocessing of 2PC circuit is performed along with the preprocessing of our 4PC protocol. Therefore, the value of mask corresponding to a wire z may differ in these two scenarios. To perform the 2PC execution of the circuit, we need to rely on the mask values selected during preprocessing for the 2PC. Let α'_{z} be the mask corresponding to wire z in the 2PC and $[\alpha'_{\mathsf{z}}]_1$ and $[\alpha'_{\mathsf{z}}]_2$ be the shares corresponding to P_1, P_2 respectively. Thus, the sharing of z is required to be updated according to α'_{z}, which essentially means updating the corresponding masked value, say β'_{z} such that $\beta'_{\mathsf{z}} = \mathsf{z} + \alpha'_{\mathsf{z}} = (\beta_{\mathsf{z}} - \alpha_{\mathsf{z}}) + \alpha'_{\mathsf{z}}$. Towards this, P_1 computes $v_1 = \beta_{\mathsf{z}} - [\alpha_{\mathsf{z}}]_1 + [\alpha'_{\mathsf{z}}]_1$ and sends it to P_2. Similarly, P_2 computes $v_2 = [\alpha'_{\mathsf{z}}]_2 - [\alpha_{\mathsf{z}}]_2$ and sends it to P_1. Then P_1, P_2 locally obtain $\beta'_{\mathsf{z}} = v_1 + v_2$ to complete the required 2PC sharing of z. Note that since both P_1, P_2 are (semi) honest, they send the correct values. Furthermore, sending v_1 or v_2 does not breach privacy since they can anyway learn these values from their own shares (for example, P_1 can compute v_2 given its shares $\beta_{\mathsf{z}}, \beta'_{\mathsf{z}}, [\alpha_{\mathsf{z}}]_1, [\alpha'_{\mathsf{z}}]_1$). In other words, this is an allowed leakage. We refer to the conversion from 4PC to 2PC sharing semantics as **share conversion**. The protocol appears in Fig. 34, with its security stated below.

Theorem 4. *Assuming collision resistant hash functions and semi-honest OT exists, protocol* 4PC *(Fig. 34) realizes* $\mathcal{F}_{\mathsf{4PC\text{-}FaF}}$ *(Fig. 33) with computational* $(1,1)$-`FaF` *security.*

Security against a mixed adversary. A closely related notion of security in the literature is that of a mixed adversary [27,38,40,8,42,49] which can simultaneously corrupt a subset of t parties maliciously and additionally a disjoint subset of h^* parties in a semi-honest manner. In contrast to the `FaF` model, the adversary here is centralized. Consequently, the mixed security model allows the view of semi-honest parties to be available to the adversary while determining a strategy for the malicious parties. Although the mixed adversarial model might seem to subsume `FaF`, Alon et al. [3] showed that (t, h^*) mixed security does not necessarily imply (t, h^*)-`FaF` security. Given this, we constructed a 4PC protocol which is secure in the `FaF` model. However, we go a step beyond and show that our protocol is additionally secure against a $(1,1)$-mixed adversary. For this, the crucial observation is that our protocol can withstand the scenario where the malicious adversary is provided with the view of semi-honest parties, which essentially captures the mixed adversarial model. Details appear in §C.

7 Applications and Benchmarks

This section focuses on evaluating the performance of QuadSquad. We first evaluate the performance of the MPC and draw comparisons to concretely efficient traditional MPC protocols that come closest to our setting. We then establish the practicality of QuadSquad via the application of secure liquidity matching and PPML for neural network inference. The source code of our implementation is available at quadsquad.

Environment. Benchmarks are performed over WAN using n1-standard-32 instances of Google Cloud, with machines located in East Australia (M_0), South Asia (M_1), South East Asia (M_2), and West Europe (M_3). The machines are equipped with 2.2GHz Intel Xeon processors supporting hyper-threading and 128GB RAM. Average bandwidth and round-trip time (rtt) between pair of machines was observed to be 180 Mbps and 158.31 ms respectively; though these values vary depending on the regions where the machines are located (see §H for details).

Software. We implement our protocol in C++17 using EMP toolkit [75]. Since we are using OT as a black-box, it can be instantiated with any state-of-the-art OT protocol such as [30]. Since the public implementation of [30] is not available, we use EMP toolkit's Ferret OT [76]. We use the NTL library [72] for computation over ring extensions for disZK protocol. We will open source our code upon acceptance. [55] and [32] are benchmarked in the MP-SPDZ [53] framework. Due to the unavailability of implementation of [59], we estimate its performance from microbenchmarks. We instantiate the collision resistant hash function with SHA256 and the PRF with AES-128 in counter mode. Computation is performed over $\mathbb{Z}_{2^{64}}$ for [32,59] and QuadSquad, and over $\mathbb{Z}_p$ for [55] where p is a 64-bit prime. We set the computational security parameter to $\kappa = 128$ and ensure statistical security of at least 2^{-40} for all the protocols. In particular, we set the degree of the polynomial modulus of the extended ring $\eta = 47$. We report the average value over 20 runs for each experiment.

Benchmarking Parameters As a measure of performance, we report the online and overall (preprocessing + online) communication per party and latency for a single execution. To capture the combined effect of communication and round complexity, we additionally use *throughput* (tp) as a benchmark parameter, following prior works [59,64,71]. Here, tp denotes the number of operations (triples for 4PC preprocessing and multiplications for 4PC online protocol) that can be performed in one second.

7.1 Performance of 4PC QuadSquad

We compare the performance of our 4PC to Fantastic Four [32], Tetrad [59] and MASCOT [55]. We evaluate a circuit comprising 10^6 multiplication gates distributed over different depths. Recall that the online communication cost of our GOD protocol is almost similar to the fair protocol due to segment-wise evaluation. Hence, we only report the cost of the fair protocol for online comparison.

The performance of the online phase appears in Table 4. The latency of our protocol (fair and GOD) is up to 3.5× higher compared to honest majority protocols of [59] and the abort variant of [32]. This captures the overhead required to achieve the stronger notion of FaF-security. On the other hand, the dishonest majority protocol of [55] bears an overhead of 4.5× to 1.01× compared to ours.

Depth	Ref.	Online		
		Latency(s)	Comm. (MB)	tp
1	Fantastic Four	2.86	12.00	350066.51
	Tetrad	1.44	6.00	692947.87
	MASCOT	13.88	24.00	72023.80
	QS	2.94	14.00	340506.67
20	Fantastic Four	4.04	12.00	247286.04
	Tetrad	2.95	6.00	339321.22
	MASCOT	25.94	24.00	38554.22
	QS	7.42	14.00	134752.73
100	Fantastic Four	11.26	12.00	88771.32
	Tetrad	9.28	6.00	107764.43
	MASCOT	74.48	24.00	13425.63
	QS	30.92	14.00	32337.66
1000	Fantastic Four	87.82	12.00	11387.21
	Tetrad	80.52	6.00	12419.36
	MASCOT	289.69	24.00	3451.94
	QS	287.71	14.06	3475.69

Table 4: Online costs for evaluating circuits with 10^6 mult gates over various depths. (**QS** denotes QuadSquad.)

The performance of the preprocessing depends only on the number of multiplication gates, not on the circuit depth. Hence, only the communication cost and throughput are reported in Table 5. [32] does not have a preprocessing and is thus, not included. Further, unlike the online phase, Table 5 reports results with respect to both fair and GOD variants, independently, since their performance in the preprocessing phase is different.

The *communication* bottleneck in the preprocessing of QuadSquad is due to computing summands of S_0 which involves running six instances of disMult, while the *computational* bottleneck is due to computing the summands of S_1 which involves running four instances of disZK.

Ref.	Comm. (KB)	tp
Tetrad	0.004	958918.39
MASCOT	67.6	4548.64
QS (Fair)	3.115	8051.27
QS (GOD)	6.22	3934.01

Table 5: Preprocessing phase cost for generating a triple.

We implement disZK using recursion as in [20] (see §A.2 for details) which results in lower communication and computation costs at the expense of higher round complexity. Our benchmarks show that disMult always tends to have a higher latency than disZK and constitutes the performance bottleneck (see §H for further details). The GOD variant requires running the preprocessing of [70] for every pair of parties which has an overhead of around 3 KB per multiplication gate per party. This approximately halves the throughput in the preprocessing phase when compared to the fair variant since the combined preprocessing across all [70] instances is akin to running six instances of disMult which in turn is the main bottleneck in fair preprocessing. With respect to throughput, [59] has the highest tp owing to its low communication costs while the tp of QuadSquad Fair is around 1.8× that of [55]. The tp of QuadSquad GOD is comparable to that of [55] despite a significantly lower communication cost because the implementation of [55] distributes the evaluation of OT instances across the available threads while our implementation runs it in a single thread to allow running the disZK protocol in parallel.

7.2 Applications

We consider applications of secure liquidity matching and PPML inference. Before describing these and evaluating their performance via QuadSquad, we describe the building blocks designed for the same.

#banks	#transactions	Online		Fair Total*		GOD Total*	
		Latency(s)	Comm. (KB)	Latency(s)	Comm. (MB)	Latency(s)	Comm. (MB)
256	50	5.23	21.28	9.46	4.75	10.35	14.56
	100	5.46	23.71	10.22	5.53	10.64	16.11
	250	5.70	32.04	10.56	7.87	11.06	20.77
	500	5.94	47.97	10.95	11.77	11.61	28.53
	1000	6.18	81.76	11.49	19.56	12.45	44.07
1024	50	5.70	74.41	10.72	7.98	12.17	44.91
	100	5.94	76.59	10.99	8.76	12.47	46.46
	250	6.18	83.36	11.32	11.10	12.89	51.13
	500	6.42	96.13	11.71	15.0	13.43	58.88
	1000	6.66	124.36	12.26	22.79	14.28	74.41

Table 6: Liquidity matching

Building blocks Each of these applications requires designing new building blocks, as described in Table 2. Specifically, we develop the following building blocks: sharing and reconstruction for SOC setting, dot product (DotP), dot product with truncation (DotPTr), conversion to arithmetic sharing from a Boolean shared bit (Bit2A), bit extraction to obtain Boolean sharing of the most significant bit (msb) from an arithmetic shared value (BitExt), bit injection to obtain arithmetic sharing of $\mathsf{b} \cdot \mathsf{v}$ from a Boolean sharing of a bit b and the arithmetic sharing of v (BitInj). Inclusion of these blocks makes QuadSquad a comprehensive framework. The details of the constructions and the complexity analysis are deferred to §F.

Liquidity matching Secure liquidity matching involves executing a privacy-preserving variant of the **gridlock** algorithm. This algorithm identifies the set of transactions among banks which can be executed while ensuring that all the banks possess sufficient liquidity to process them. The gridlock algorithm can be considered for the following three scenarios (i) the source and the destination banks of the transactions are open (non-private) (sodoGR), (ii) the source is open, but the destination is hidden (secret) (sodsGR), and (iii) both the source and the destination are hidden (ssdsGR). A secure realization for liquidity matching was provided in the work of [7], albeit via traditionally secure MPC. Given the sensitive nature of financial data involved in liquidity matching, clearly, FaF-security is more apt. Hence, we focus on designing FaF-secure protocols for the same. Further, with respect to the three scenarios described above, note that in most practical cases hiding the transaction amount is sufficient. Hence, we consider only the sodoGR instance (see §G for the secure protocol). However, we note that extending our techniques to the other two scenarios is also possible.

At a high level, the protocol proceeds in a sequence of iterations where each iteration attempts to check the feasibility of clearing a subset of transactions. The protocol terminates with a feasible set or reports a deadlock where no transactions can be cleared. Since the communication and computation costs are identical across all iterations, we benchmark the performance for running one iteration of the protocol and report the results in Table 6. We see similar trends as observed while evaluating the performance of the MPC, where the GOD variant is on par with the fair variant with respect to the overall latency.

Further, we observe that the latency of an iteration for both variants is within 15s even for a large number of banks and set of transactions. This hints towards the practicality of using QuadSquad for real time liquidity matching systems, especially considering the advantages of the "stronger" `FaF` model.

PPML For the application of PPML inference, we consider the popularly used [59,58,74,71] Neural Network (NN) architectures, given below.

- *FCNN*: Fully-Connected NN consists of two hidden layers, each with 128 nodes followed by an output layer of 10 nodes. ReLU is applied after each layer.
- *LeNet*: This NN consists of 2 convolutional layers and 2 fully connected layers, each followed by ReLU activation function. Moreover, the convolutional layers are followed by an average-pooling layer.

The inference task is performed over the publicly available MNIST [60] dataset which is a collection of 28×28 pixel, handwritten digit images with a label between 0 and 9 for each. We note that our techniques easily extend to securely evaluating other NN architectures such as convolutional neural network (CNN) and VGG16 [73] used in other MPC-based PPML frameworks of [58,59,74].

Network	Ref.	Online			Total*	
		Latency (s)	Comm. (MB)	`tp` (queries/min)	Latency (s)	Comm. (MB)
FCN	Fantastic Four	48.06	27.71	43.75	48.06	27.71
FCN	Tetrad	1.66	0.006	47099.05	2.38	0.02
FCN	**QS Fair**	6.00	0.022	3176.65	29.77	371.15
FCN	**QS GOD**	6.00	0.022	3176.65	44.49	746.46
LeNet	Fantastic Four	220.17	134.28	84.22	220.17	134.28
LeNet	Tetrad	2.45	0.36	787.09	3.25	0.91
LeNet	**QS Fair**	10.36	1.27	64.24	308.89	7251.73
LeNet	**QS GOD**	10.36	1.27	64.24	607.53	14868.07

Table 7: NN inference where **QS** denotes QuadSquad.

We compare the performance of PPML inference via QuadSquad for the above mentioned NN with the honest majority protocols of [59] and [32]. PPML in the 4PC dishonest majority (malicious) setting has not been explored so far. The results of our experiments are summarised in Table 7. Note that the latency reported is obtained via a single instance of circuit evaluation, whereas the throughput is computed by running the inference on larger batches. Here, `tp` is the number of queries evaluated in a minute since inference over WAN requires more than a second to complete. Our fair and GOD variants have an overhead of 3x–4x in performance respectively. However we provide a stronger adversarial model compared to [59]. The numbers in Table 7 for [32] from MP-SPDZ [53] are unexpectedly high. We suspect that this anomaly is due to the preprocessing cost of [32]. However, the benchmarks seem consistent with those reported in [32] and pinpointing the exact cause is challenging due to the vast MP-SPDZ codebase. It is worth noting that the communication cost of [32] per query for larger batch sizes decreases to 0.93 MB per party for FCN and 0.46 MB per party for LeNet. The QuadSquad protocols have higher cost in the preprocessing phase from using more expensive primitives like OT and the feature dependent preprocessing phase for dot-product. However, the comparable online performance to [59] and [32] and the stronger security model make it a viable practical option despite the overhead in preprocessing.

Acknowledgements

Arpita Patra would like to acknowledge financial support from DST National Mission on Interdisciplinary Cyber-Physical Systems (NM-ICPS) 2020-2025 and SERB MATRICS (Theoretical Sciences) Grant 2020-2023. Varsha Bhat Kukkala would like to acknowledge financial support from National Security Council, India. Nishat Koti would like to acknowledge support from Centre for Networked Intelligence (a Cisco CSR initiative) at the Indian Institute of Science, Bengaluru. Shravani Patil would like to acknowledge financial support from DST National Mission on Interdisciplinary Cyber-Physical Systems (NM-ICPS) 2020-2025. The authors would also like to acknowledge the support from Google Cloud for benchmarking.

References

1. Abspoel, M., Cramer, R., Damgård, I., Escudero, D., Yuan, C.: Efficient information-theoretic secure multiparty computation over $\mathbb{Z}/p^k\mathbb{Z}$ via galois rings. In: TCC (2019)
2. Abspoel, M., Dalskov, A.P.K., Escudero, D., Nof, A.: An efficient passive-to-active compiler for honest-majority MPC over rings. In: ACNS (2021)
3. Alon, B., Omri, E., Paskin-Cherniavsky, A.: Mpc with friends and foes. In: CRYPTO (2020)
4. Araki, T., Barak, A., Furukawa, J., Lichter, T., Lindell, Y., Nof, A., Ohara, K., Watzman, A., Weinstein, O.: Optimized honest-majority MPC for malicious adversaries - breaking the 1 billion-gate per second barrier. In: IEEE S&P (2017)
5. Araki, T., Furukawa, J., Lindell, Y., Nof, A., Ohara, K.: High-throughput semi-honest secure three-party computation with an honest majority. In: ACM CCS (2016)
6. Archer, D.W., Bogdanov, D., Lindell, Y., Kamm, L., Nielsen, K., Pagter, J.I., Smart, N.P., Wright, R.N.: From keys to databases—real-world applications of secure multi-party computation. The Computer Journal (2018)
7. Atapoor, S., Smart, N.P., Alaoui, Y.T.: Private liquidity matching using mpc. IACR Cryptol. ePrint Arch. (2021)
8. Badrinarayanan, S., Jain, A., Manohar, N., Sahai, A.: Secure mpc: laziness leads to god. In: ASIACRYPT (2020)
9. Baum, C., Damgård, I., Toft, T., Zakarias, R.W.: Better preprocessing for secure multiparty computation. In: ACNS (2016)
10. Baum, C., Orsini, E., Scholl, P.: Efficient secure multiparty computation with identifiable abort. In: Theory of Cryptography Conference (2016)

11. Beaver, D.: Efficient multiparty protocols using circuit randomization. In: CRYPTO (1991)
12. Beaver, D.: Precomputing oblivious transfer. In: CRYPTO (1995)
13. Ben-Efraim, A., Nielsen, M., Omri, E.: Turbospeedz: Double your online spdz! improving spdz using function dependent preprocessing. In: ACNS (2019)
14. Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In: STOC (1988)
15. Bogdanov, D., Kamm, L., Kubo, B., Rebane, R., Sokk, V., Talviste, R.: Students and taxes: a privacy-preserving social study using secure computation. IACR Cryptology ePrint Archive (2015)
16. Bogdanov, D., Laur, S., Willemson, J.: Sharemind: A framework for fast privacy-preserving computations. In: ESORICS (2008)
17. Bogdanov, D., Talviste, R., Willemson, J.: Deploying secure multi-party computation for financial data analysis - (short paper). In: FC (2012)
18. Boneh, D., Boyle, E., Corrigan-Gibbs, H., Gilboa, N., Ishai, Y.: Zero-knowledge proofs on secret-shared data via fully linear pcps. In: CRYPTO (2019)
19. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Scholl, P.: Efficient pseudorandom correlation generators: Silent ot extension and more. In: CRYPTO (2019)
20. Boyle, E., Gilboa, N., Ishai, Y., Nof, A.: Practical fully secure three-party computation via sublinear distributed zero-knowledge proofs. In: ACM CCS (2019)
21. Boyle, E., Gilboa, N., Ishai, Y., Nof, A.: Efficient fully secure computation via distributed zero-knowledge proofs. In: ASIACRYPT (2020)
22. Byali, M., Chaudhari, H., Patra, A., Suresh, A.: FLASH: fast and robust framework for privacy-preserving machine learning. PETS (2020)
23. Byali, M., Hazay, C., Patra, A., Singla, S.: Fast actively secure five-party computation with security beyond abort. In: ACM CCS (2019)
24. Byali, M., Joseph, A., Patra, A., Ravi, D.: Fast secure computation for small population over the internet. In: ACM CCS (2018)
25. Chaudhari, H., Choudhury, A., Patra, A., Suresh, A.: ASTRA: High Throughput 3PC over Rings with Application to Secure Prediction. In: ACM CCSW@CCS (2019)
26. Chaudhari, H., Rachuri, R., Suresh, A.: Trident: Efficient 4PC Framework for Privacy Preserving Machine Learning. NDSS (2020)
27. Chaum, D.: The spymasters double-agent problem: Multiparty computations secure unconditionally from minorities and cryptographically from majorities; crypto'89, lncs 435 (1990)
28. Chida, K., Genkin, D., Hamada, K., Ikarashi, D., Kikuchi, R., Lindell, Y., Nof, A.: Fast large-scale honest-majority MPC for malicious adversaries. In: CRYPTO (2018)
29. Cleve, R.: Limits on the security of coin flips when half the processors are faulty (extended abstract). In: ACM STOC (1986)
30. Couteau, G., Rindal, P., Raghuraman, S.: Silver: Silent vole and oblivious transfer from hardness of decoding structured ldpc codes. In: Annual International Cryptology Conference. pp. 502–534. Springer (2021)
31. Cramer, R., Damgård, I., Escudero, D., Scholl, P., Xing, C.: SPD$\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for Dishonest Majority. In: CRYPTO (2018)
32. Dalskov, A., Escudero, D., Keller, M.: Fantastic four: Honest-majority four-party secure computation with malicious security. In: USENIX Security (2021)

33. Damgård, I., Escudero, D., Frederiksen, T.K., Keller, M., Scholl, P., Volgushev, N.: New primitives for actively-secure MPC over rings with applications to private machine learning. IEEE S&P (2019)
34. Damgård, I., Nielsen, J.B.: Scalable and unconditionally secure multiparty computation. In: CRYPTO (2007)
35. Damgård, I., Orlandi, C., Simkin, M.: Yet another compiler for active security or: Efficient MPC over arbitrary rings. In: CRYPTO (2018)
36. Damgård, I., Pastro, V., Smart, N.P., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: CRYPTO (2012)
37. Demmler, D., Schneider, T., Zohner, M.: ABY - A framework for efficient mixed-protocol secure two-party computation. In: NDSS (2015)
38. Dolev, D., Dwork, C., Waarts, O., Yung, M.: Perfectly secure message transmission. Journal of the ACM (JACM) (1993)
39. Dolev, D., Strong, H.R.: Authenticated algorithms for byzantine agreement. SIAM Journal on Computing (1983)
40. Fitzi, M., Hirt, M., Maurer, U.: Trading correctness for privacy in unconditional multi-party computation. In: CRYPTO (1998)
41. Furukawa, J., Lindell, Y., Nof, A., Weinstein, O.: High-throughput secure three-party computation for malicious adversaries and an honest majority. In: EUROCRYPT (2017)
42. Ghodosi, H., Pieprzyk, J.: Multi-party computation with omnipresent adversary. In: PKC (2009)
43. Gilboa, N.: Two party rsa key generation. In: CRYPTO (1999)
44. Goldreich, O.: Foundations of cryptography: volume 1, basic tools (2007)
45. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or A completeness theorem for protocols with honest majority. In: STOC (1987)
46. Gordon, S.D., Ranellucci, S., Wang, X.: Secure computation with low communication from cross-checking. In: ASIACRYPT (2018)
47. Goyal, V., Song, Y., Zhu, C.: Guaranteed output delivery comes free in honest majority mpc. In: CRYPTO (2020)
48. Hazay, C., Lindell, Y.: A note on the relation between the definitions of security for semi-honest and malicious adversaries. IACR Cryptol. ePrint Arch. (2010)
49. Hirt, M., Mularczyk, M.: Efficient mpc with a mixed adversary. LIPIcs (2020)
50. Ishai, Y., Kilian, J., Nissim, K., Petrank, E.: Extending oblivious transfers efficiently. In: CRYPTO (2003)
51. Ishai, Y., Kumaresan, R., Kushilevitz, E., Paskin-Cherniavsky, A.: Secure computation with minimal interaction, revisited. In: CRYPTO (2015)
52. Ishai, Y., Prabhakaran, M., Sahai, A.: Founding cryptography on oblivious transfer–efficiently. In: CRYPTO (2008)
53. Keller, M.: MP-SPDZ: A versatile framework for multi-party computation. In: ACM CCS (2020)
54. Keller, M., Orsini, E., Scholl, P.: Actively secure ot extension with optimal overhead. In: CRYPTO (2015)
55. Keller, M., Orsini, E., Scholl, P.: MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In: ACM CCS (2016)
56. Keller, M., Pastro, V., Rotaru, D.: Overdrive: Making SPDZ great again. In: EUROCRYPT (2018)
57. Koti, N., Kukkala, V.B., Patra, A., Gopal, B.R.: Pentagod: Stepping beyond traditional god with five parties. Cryptology ePrint Archive (2022)
58. Koti, N., Pancholi, M., Patra, A., Suresh, A.: SWIFT: Super-fast and Robust Privacy-Preserving Machine Learning. In: USENIX Security (2021)

59. Koti, N., Patra, A., Rachuri, R., Suresh, A.: Tetrad: Actively secure 4pc for secure training and inference. arXiv preprint arXiv:2106.02850 (2021)
60. LeCun, Y., Cortes, C.: MNIST handwritten digit database (2010), http://yann.lecun.com/exdb/mnist/
61. Lindell, Y., Nof, A.: A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In: ACM CCS (2017)
62. Mazloom, S., Le, P.H., Ranellucci, S., Gordon, S.D.: Secure parallel computation on national scale volumes of data. In: USENIX Security (2020)
63. Mazloom, S., Le, P.H., Ranellucci, S., Gordon, S.D.: Secure parallel computation on national scale volumes of data. In: USENIX Security (2020)
64. Mohassel, P., Rindal, P.: ABY^3: A mixed protocol framework for machine learning. In: ACM CCS (2018)
65. Mohassel, P., Rosulek, M., Zhang, Y.: Fast and secure three-party computation: The garbled circuit approach. In: ACM CCS (2015)
66. Mohassel, P., Zhang, Y.: Secureml: A system for scalable privacy-preserving machine learning. In: IEEE S&P (2017)
67. Nordholt, P.S., Veeningen, M.: Minimising communication in honest-majority MPC by batchwise multiplication verification. In: ACNS (2018)
68. Orlandi, C.: Is multiparty computation any good in practice? In: IEEE ICASSP (2011)
69. Orsini, E., Smart, N.P., Vercauteren, F.: Overdrive2k: Efficient secure mpc over $\mathbb{Z}/2^k$ from somewhat homomorphic encryption. In: CT-RSA (2020)
70. Patra, A., Schneider, T., Suresh, A., Yalame, H.: Aby2. 0: Improved mixed-protocol secure two-party computation. In: USENIX Security (2021)
71. Patra, A., Suresh, A.: BLAZE: Blazing Fast Privacy-Preserving Machine Learning. NDSS (2020)
72. Shoup, V.: NTL: A Library for doing Number Theory. https://libntl.org/ (2021)
73. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2014)
74. Wagh, S., Tople, S., Benhamouda, F., Kushilevitz, E., Mittal, P., Rabin, T.: Falcon: Honest-majority maliciously secure framework for private deep learning. arXiv preprint (2020)
75. Wang, X., Malozemoff, A.J., Katz, J.: EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit (2016)
76. Yang, K., Weng, C., Lan, X., Zhang, J., Wang, X.: Ferret: Fast extension for correlated ot with small communication. In: ACM CCS (2020)

Supplementary Material

A Appendix: Preliminaries

A.1 Security Model

We follow the standard ideal-world/real-world simulation paradigm to prove the security of our protocols [44]. This security notion is defined by considering an ideal functionality $\mathcal{F}$ wherein the corrupted and the uncorrupted parties send their inputs to the trusted third party over a perfectly secure channel, which performs the computation and sends the output to the parties. Informally, a real-world protocol is deemed to be secure, if whatever the adversary can do in the real world, can also be done in the ideal world. In the classical definition, this is captured by designing an ideal-world adversary (simulator) which can simulate the view of the real-world adversary corrupting a subset of the parties in $\mathcal{P}$. However, in the FaF-security model defined in [3], it is additionally required that the view of any subset of uncorrupted (or semi-honest) parties can be simulated. The security of a protocol is thus established by constructing two simulators in the ideal-world, one each for the malicious adversary and the semi-honest adversary respectively. Moreover, to explicitly capture the fact that the malicious adversary can arbitrarily deviate from the protocol in the real-world by sending messages to the uncorrupted (semi-honest) parties, in the ideal world, the malicious adversary is allowed to send its entire view to the semi-honest adversary.

Let $\mathcal{A}$ denote the probabilistic polynomial time (PPT) malicious adversary in the real-world corrupting t parties in $\mathcal{I} \subset \mathcal{P}$, and $\mathcal{S}_{\mathcal{A}}$ denote the corresponding ideal-world simulator. Similarly, let $\mathcal{A}_{\mathcal{H}}$ denote the (PPT) semi-honest adversary corrupting h^* parties in $\mathcal{H} \subset \mathcal{P} \setminus \mathcal{I}$ in the real-world, and $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$, be the ideal-world simulator. Note that in the classical definition of ideal-world, $\mathcal{H} = \phi$. Let $\mathcal{F}$ be the ideal-world functionality. Let $\text{VIEW}^{\text{REAL}}_{\mathcal{A},\Pi}(1^\lambda, z_{\mathcal{A}})$ be the malicious adversary's ($\mathcal{A}$) view and $\text{OUT}^{\text{REAL}}_{\mathcal{A},\Pi}(1^\lambda, z_{\mathcal{A}})$ denote the output of the uncorrupted parties (in $\mathcal{P} \setminus \mathcal{I}$) during a random execution of Π, where $z_{\mathcal{A}}$ is the auxiliary input of $\mathcal{A}$. Similarly, let $\text{VIEW}^{\text{REAL}}_{\mathcal{A},\mathcal{A}_{\mathcal{H}},\Pi}(1^\lambda, z_{\mathcal{A}}, z_{\mathcal{A}_{\mathcal{H}}})$ be the semi-honest adversary's ($\mathcal{A}_{\mathcal{H}}$) view during an execution of Π running alongside $\mathcal{A}$, where $z_{\mathcal{A}_{\mathcal{H}}}$ is the auxiliary input of $\mathcal{A}_{\mathcal{H}}$. Note that $\text{VIEW}^{\text{REAL}}_{\mathcal{A},\mathcal{A}_{\mathcal{H}},\Pi}(1^\lambda, z_{\mathcal{A}}, z_{\mathcal{A}_{\mathcal{H}}})$ consists of the non-prescribed messages sent by the malicious adversary to the semi-honest parties. Correspondingly, let $\text{VIEW}^{\text{IDEAL}}_{\mathcal{A},\mathcal{F}}(1^\lambda, z_{\mathcal{A}})$ be the malicious adversary's simulated view with $\mathcal{A}$ corrupting parties in $\mathcal{I}$ and $\text{OUT}^{\text{IDEAL}}_{\mathcal{A},\mathcal{F}}(1^\lambda, z_{\mathcal{A}})$ denote the output of the uncorrupted parties (in $\mathcal{P} \setminus \mathcal{I}$) during a random execution of ideal-world functionality $\mathcal{F}$. Similarly, let $\text{VIEW}^{\text{IDEAL}}_{\mathcal{A},\mathcal{A}_{\mathcal{H}},\mathcal{F}}(1^\lambda, z_{\mathcal{A}}, z_{\mathcal{A}_{\mathcal{H}}})$ be the semi-honest adversary's simulated view with $\mathcal{A}_{\mathcal{H}}$ corrupting parties in $\mathcal{H}$ during an execution of $\mathcal{F}$ running alongside $\mathcal{A}$.

A protocol Π is said to compute $\mathcal{F}$ with (weak) computational (t, h^*)-FaF-security if

$$(\text{VIEW}^{\text{IDEAL}}_{\mathcal{A},\mathcal{F}}(1^\lambda, z_\mathcal{A}), \text{OUT}^{\text{IDEAL}}_{\mathcal{A},\mathcal{F}}(1^\lambda, z_\mathcal{A})) \equiv (\text{VIEW}^{\text{REAL}}_{\mathcal{A},\Pi}(1^\lambda, z_\mathcal{A}), \text{OUT}^{\text{REAL}}_{\mathcal{A},\Pi}(1^\lambda, z_\mathcal{A})),$$

$$(\text{VIEW}^{\text{IDEAL}}_{\mathcal{A},\mathcal{A}_\mathcal{H},\mathcal{F}}(1^\lambda, z_\mathcal{A}, z_{\mathcal{A}_\mathcal{H}}), \text{OUT}^{\text{IDEAL}}_{\mathcal{A},\mathcal{F}}(1^\lambda, z_\mathcal{A})) \equiv (\text{VIEW}^{\text{REAL}}_{\mathcal{A},\mathcal{A}_\mathcal{H},\Pi}(1^\lambda, z_\mathcal{A}, z_{\mathcal{A}_\mathcal{H}}), \text{OUT}^{\text{REAL}}_{\mathcal{A},\Pi}(1^\lambda, z_\mathcal{A})).$$

A.2 Building Blocks

Shared Key Setup Let $F : \{0,1\}^\kappa \times \{0,1\}^\kappa \to X$ be a secure pseudo-random function (PRF), with co-domain X being $\mathbb{Z}_{2^\lambda}$. The set of keys established between the parties for the 4PC protocol is as follows:

- One key shared between every pair– $k_{12}, k_{13}, k_{14}, k_{23}, k_{24}, k_{34}$ for parties (P_1, P_2), $(P_1, P_3), (P_1, P_4), (P_2, P_3), (P_2, P_4), (P_3, P_4)$ respectively.
- One key shared between every triple of parties– $k_{123}, k_{124}, k_{134}, k_{234}$ for parties $(P_1, P_2, P_3), (P_1, P_2, P_4), (P_1, P_3, P_4), (P_2, P_3, P_4)$ respectively.
- One shared key known to all the parties– $k_\mathcal{P}$.

Suppose P_1, P_2 wish to sample a random value $r \in \mathbb{Z}_{2^\lambda}$ non-interactively, they do so by invoking $F_{k_{12}}(id_{12})$ and obtain r. Here, id_{12} denotes a counter maintained by the parties, and is updated after every PRF invocation. The appropriate keys used to sample is implicit from the context, from the identities of the pair (or triple) that sample or from the fact that it is sampled by all, and, hence, is omitted.

The key setup is modelled via a functionality $\mathcal{F}_{\mathsf{setup}}$ (Fig. 7) that can be realised using any FaF-secure MPC protocol.

Functionality $\mathcal{F}_{\mathsf{setup}}$

$\mathcal{F}_{\mathsf{setup}}$ interacts with the parties in $\mathcal{P}$ and the adversaries $\mathcal{S}_\mathcal{A}$ and $\mathcal{S}_{\mathcal{A}_\mathcal{H}}$. $\mathcal{F}_{\mathsf{setup}}$ picks random keys $\mathsf{k}_\mathcal{T}$ for every set $\mathcal{T} \subseteq \mathcal{P}$.
– Set $\mathbf{y}_s = \{\mathsf{k}_\mathcal{T}\}_{\forall \mathcal{T} \subseteq \mathcal{P}}$, when $P_s \in \mathcal{T}$.

Output: Send $(\mathsf{Output}, \mathbf{y}_s)$ to $P_s \in \mathcal{P}$.

Fig. 7: Ideal functionality for shared-key setup

Collision Resistant Hash Function Consider a hash function family $\mathtt{H}(\cdot) : \mathcal{K} \times \mathcal{L} \to \mathcal{Y}$. The hash function $\mathtt{H}$ is said to be collision resistant if, for all probabilistic polynomial-time adversaries $\mathcal{A}$, given the description of $\mathtt{H}_k$ where $k \in_R \mathcal{K}$, there exists a negligible function $\mathsf{negl}()$ such that $\Pr[(x_1, x_2) \leftarrow \mathcal{A}(k) : (x_1 \neq x_2) \wedge \mathtt{H}_k(x_1) = \mathtt{H}_k(x_2)] \leq \mathsf{negl}(\kappa)$, where $m = \mathsf{poly}(\kappa)$ and $x_1, x_2 \in_R \{0,1\}^m$.

Commitment Scheme Let $\mathsf{Com}(x)$ denote the commitment of a value x. The commitment scheme $\mathsf{Com}(x)$ possesses two properties; *hiding* and *binding*. The former ensures privacy of the value v given just its commitment $\mathsf{Com}(\mathsf{v})$, while the latter prevents a corrupt server from opening the commitment to a different value $x' \neq x$. The practical realization of a commitment scheme is via a hash function $\mathtt{H}()$ given below, whose security can be proved in the random-oracle model (ROM)– for $(c, o) = (\mathtt{H}((x||r)), x||r) = \mathsf{Com}(x; r)$. Throughout the paper, we abuse the terminology to denote c by $\mathsf{Com}(x)$ and o by opening of $\mathsf{Com}(x)$.

3 Party Joint Message Passing: The jmp3 primitive enables two parties to relay a common message to a third party, such that either the relay is successful or a dispute pair is identified. The ideal functionality appears in Fig. 8 and the protocol in Fig. 9.

Functionality $\mathcal{F}_{\mathsf{jmp3}}$

$\mathcal{F}_{\mathsf{jmp3}}$ interacts with the parties in $\mathcal{P}$ and the adversaries $\mathcal{S}_{\mathcal{A}}$ and $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$.

- $\mathcal{F}_{\mathsf{jmp3}}$ receives $(\mathsf{Input}, \vec{\mathsf{v}}_s)$ from P_s for $s \in \{i, j\}$, while it receives $(\mathsf{Select}, \mathcal{D})$ from $\mathcal{S}_{\mathcal{A}}$. Here, $\mathcal{D}$ denotes the pair of parties that $\mathcal{S}_{\mathcal{A}}$ wants to choose as dispute pair. Let P^* be the party corrupted by $\mathcal{S}_{\mathcal{A}}$.
- If $\vec{\mathsf{v}}_i = \vec{\mathsf{v}}_j$ and $\mathcal{D} = \perp$, then $\mathsf{msg}_i = \mathsf{msg}_j = \perp$, $\mathsf{msg}_k = \vec{\mathsf{v}}_i$.
- Else if, $P^* \in \mathcal{D}$, then set $\mathtt{DP} = \mathcal{D}$ and $\mathsf{msg}_i = \mathsf{msg}_j = \mathsf{msg}_k = \mathtt{DP}$.
- Else set $\mathtt{DP} = \{P^*, P_t\}$, where $P_t \in_R \mathcal{D}$ and $\mathsf{msg}_i = \mathsf{msg}_j = \mathsf{msg}_k = \mathtt{DP}$.

Output: Send $(\mathsf{Output}, \mathsf{msg}_s)$ to P_s, where $t \in \{i, j, k\}$.

Fig. 8: Ideal functionality for jmp3 primitive

Protocol $\mathsf{jmp3}(P_i, P_j, \vec{\mathsf{v}}, P_k)$

- P_i sends v to P_k and P_j sends $\mathtt{H}(\mathsf{v})$ to P_k.
- P_k broadcasts $(\mathtt{H}(\mathsf{v}_i), \mathtt{H}(\mathsf{v}_j))$ if the received values are inconsistent, where v_i is the value that P_k received from P_i, and $\mathtt{H}(\mathsf{v}_j)$ is the value that P_k received from P_j.
- For each $s \in \{i, j\}$, if P_s disagrees with the broadcast, it broadcasts complaint against P_k. All the parties set dispute pair as $\mathtt{DP} = \{P_s, P_k\}$. If both P_i, P_j broadcast complaint, all parties set $\mathtt{DP} = \{P_i, P_k\}$.
- If neither P_i nor P_j broadcast complaint then parties set $\mathtt{DP} = \{P_i, P_j\}$.

Fig. 9: P_i, P_j send a common list of values to P_k

Lemma 1. *Protocol* jmp3 *(Fig. 9) requires* 1 *round and an amortized communication of* 1 *element for an honest execution. If any malicious behaviour is identified, it takes 3 rounds of communication.*

Proof. Party P_i sends value v to P_k while P_j sends hash of the same to P_k. This accounts for one round and communication of 1 element. In the optimistic case, the protocol terminates at this stage. Otherwise, P_k broadcasts $(\mathtt{H}(\mathsf{v}_i), \mathtt{H}(\mathsf{v}_j))$, which leads to identification of a dispute pair. If a malicious P_k falsely broadcast, then either of P_i or P_j complaint and $\mathtt{DP}$ contains P_k.

Note that if the jmp3 execution identifies any malicious activity, then it identifies a dispute pair ($\mathtt{DP}$), which assured to have the malicious party. So, this 3 rounds of communication is a one time overhead.

4 Party Joint Message Passing(jmp4)*:* jmp4 allows two parties P_i, P_j holding a common value v, to send it to the other two parties P_k, P_m such that, either both the parties receive the correct v or all the parties identify $\mathtt{DP}$. This protocol invokes jmp3 twice in parallel, with P_i, P_j as the senders in both. The receiver's role is performed by P_k in one execution and P_m in the other. An invocation of jmp3 can either be successful, or $\mathtt{DP}$ is identified. If at least one of the jmp3 executions identifies $\mathtt{DP}$, then the parties (deterministically) choose one of them as the output. Otherwise, both the executions of jmp3 succeed and so does jmp4. Similar to jmp3, for every pair of parties, we perform the send and verification of all the parallel executions of jmp4 simultaneously. The ideal functionality appears in Fig. 10 and the protocol in Fig. 11.

Functionality $\mathcal{F}_{\mathsf{jmp4}}$

$\mathcal{F}_{\mathsf{jmp4}}$ interacts with the parties in $\mathcal{P}$ and the adversaries $\mathcal{S}_{\mathcal{A}}$ and $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$.

- $\mathcal{F}_{\mathsf{jmp4}}$ receives $(\mathsf{Input}, \vec{\mathsf{v}}_s)$ from P_s for $s \in \{i, j\}$, while it receives $(\mathsf{Select}, \mathcal{D})$ from $\mathcal{S}_{\mathcal{A}}$. Here, $\mathcal{D}$ denotes the pair of parties that $\mathcal{S}_{\mathcal{A}}$ wants to choose as dispute pair. Let P^* be the party corrupted by $\mathcal{S}_{\mathcal{A}}$.
- If $\vec{\mathsf{v}}_i = \vec{\mathsf{v}}_j$ and $\mathcal{D} = \perp$, then $\mathsf{msg}_i = \mathsf{msg}_j = \perp$, $\mathsf{msg}_k = \mathsf{msg}_m = \vec{\mathsf{v}}_i$.
- Else if, $P^* \in \mathcal{D}$, then set $\mathtt{DP} = \mathcal{D}$ and $\mathsf{msg}_s = \mathtt{DP}$ for each $P_s \in \mathcal{P}$.
- Else set $\mathtt{DP} = \mathcal{P} \setminus \mathcal{D}$ and $\mathsf{msg}_s = \mathtt{DP}$ for each $P_s \in \mathcal{P}$.

Output: Send $(\mathsf{Output}, \mathsf{msg}_s)$ to $P_s \in \mathcal{P}$.

Fig. 10: Ideal functionality for jmp4 primitive

Protocol jmp4$(P_i, P_j, \vec{\mathsf{v}}, P_k, P_m)$

- P_i and P_j jmp3-send $\vec{\mathsf{v}}$ to P_k; P_i and P_j jmp3-send $\vec{\mathsf{v}}$ to P_m in parallel.
- Let $\mathtt{DP}_k$ and $\mathtt{DP}_m$ be the dispute pairs identified in the invocations with P_k and P_m respectively.
- If $\mathtt{DP}_k = \mathtt{DP}_m = \phi$, then parties terminate.
- Else, if $\mathtt{DP}_k \neq \phi$, parties output $\mathtt{DP}_k$.
- Else, if $\mathtt{DP}_m \neq \phi$, parties output $\mathtt{DP}_m$.

Fig. 11: P_i, P_j send a common list of values to P_k, P_m

Lemma 2. *Protocol* jmp4 *(Fig. 11) requires* 1 *round and an amortized communication of* 2 *elements. If any malicious behaviour is identified, it takes 3 rounds of communication.*

Proof. The protocol jmp4 is composed of two parallel execution of jmp3, which together requires communication of 2 elements and 1 round.

Oblivious Product Evaluation (OPE)*:* The ideal functionality for OPE appears in Fig. 12.

Functionality $\mathcal{F}_{\mathsf{OPE}}$

$\mathcal{F}_{\mathsf{OPE}}$ interacts with two parties in P_s, P_r and the adversaries $\mathcal{S}_{\mathcal{A}}$, $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$.

- $\mathcal{F}_{\mathsf{OPE}}$ receives a value m_r from the receiver P_r and two values $m_s, -\alpha_s$ from the sender P_s.
- Let P^* be the party controlled by $\mathcal{S}_{\mathcal{A}}$, $\mathcal{S}_{\mathcal{A}}$ fixes the input(s) on behalf of P^*, and P^*_H be the party controlled by $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$.
- $\mathcal{S}_{\mathcal{A}}$ sends it's view to $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$.
- $\mathcal{F}_{\mathsf{OPE}}$ sets $\mathsf{msg}_s = \perp$ and $\mathsf{msg}_r = \alpha_r = m_s \cdot m_r - \alpha_s$.
- $\mathcal{F}_{\mathsf{OPE}}$ sends msg of P^* to $\mathcal{S}_{\mathcal{A}}$, and $\mathcal{S}_{\mathcal{A}}$ sends `accept` or `abort`.
- If $\mathcal{F}_{\mathsf{OPE}}$ receives `abort` from $\mathcal{S}_{\mathcal{A}}$, it sets $\mathsf{msg}_s = \perp, \mathsf{msg}_r = \perp$, else `continue`.

Output: Send $(\mathsf{Output}, \mathsf{msg}_i)$, for $i \in \{r, s\}$.

Fig. 12: Ideal functionality for OPE

Technique of [43,55] for reducing OPE *to* OT. For each $i \in \lambda$, the sender (say the party holding y) gives as input to the i^{th} OT_i, $(z_i, y + z_i)$, where $z \in \mathbb{Z}_{2^\lambda}$ is a random value and z_i is the i^{th} bit of z. Correspondingly, the receiver provides the bit decomposition of its input x to the OT. That is, it gives x_i as the input to OT_i and receives as output $x_i y + z_i$. Parties further set their arithmetic share of xy as follows: (i) The sender sets its share to be $-\sum_{i=1}^{\lambda} z_i \cdot 2^{i-1}$ and (ii) The receiver sets its share as $\sum_{i=1}^{\lambda}(x_i y + z_i) \cdot 2^{i-1}$.

Depending on the domain, we will use different OT(s) to obtain OPE. cOT_λ implies 1-out-of-2 correlated OT, where the sender's messages are λ bit strings. cOT_1 implies 1-out-of-2 correlated OT, where the sender's messages are bits, Note that both of the OTs described above are input independent. Whereas, OT_λ is 1-out-of-2 input dependent OT, where the length of the sender's messages are λ, and OT_1 is an input dependent bit OT. OT costs from Ferret [76] are given below, in Table 8, which incurs the same (amortized) costs for semi-honest OT and malicious OT. The technique for obtaining combined instance of OPE using jmp4 is described in §5.2. The cost of this combined instance of OPE in our disMult protocol (Fig. 5) is 2× the cost of individual OPE instance, that is, 259 elements.

OT	Message Length	Communication Cost
cOT_1	1	0.44 bits
cOT_λ	λ	0.44 elements
OT_1	1	3.44 bits
OT_λ	λ	$\sim$ 129.5 elements

Table 8: Ferret OT costs

Distributed Zero-Knowledge: In [20], the authors provide a distributed zero-knowledge protocol with sub-linear proof size, which is adapted for verification of messages sent in a 3PC protocol with one corruption. To achieve robustness, their zero-knowledge protocol follows the standard template of identifying a TTP in case any malicious behaviour is detected during protocol execution. For this, the protocol relies on the property of *recomputable verification*, which implies that during verification, each verifier sends a message computed as a deterministic function of the messages from the prover and public values. This means that the prover itself can recompute the messages of each verifier. This property lends itself well to identifying a dispute pair, and hence in the 3PC case, a TTP. We extend their zero-knowledge protocol to the 4 party case with one malicious corruption, and similar to [20], using the property of *recomputable verification*, we identify a dispute pair DP in case the verification fails. This perfectly models the template we require in our constructions to achieve GOD.

As described in §2, in our multiplication protocol, given P_i's input shares $a_1, b_1, a_2, b_2, a_3, b_3$ and its randomness e_1, e_2 where $e = e_1 + e_2$, parties need to ensure that $c(a_1, b_1, a_2, b_2, a_3, b_3, e)$ evaluates to 0, where

$$\begin{aligned} &c(a_1, b_1, a_2, b_2, a_3, b_3, e) \\ &= a_1b_2 + a_2b_1 + a_1b_3 + a_3b_1 + a_2b_3 + a_3b_2 - e \end{aligned} \quad (5)$$

As required for the proof, input to c is distributed among the parties such that P_j holds $(a_1, b_1, 0, 0, 0, 0, e_2)$, P_m holds $(0, 0, a_2, b_2, 0, 0, 0)$ and $(0, 0, 0, 0, a_3, b_3, e_1)$ is known to P_k.

The functionality $\mathcal{F}_{\mathsf{disZK}}$ appears in Fig. 13 and the protocol in Fig. 14.

Simulator $\mathcal{F}_{\mathsf{disZK}}$

$\mathcal{F}_{\mathsf{disZK}}$ interacts with the parties in $\mathcal{P}$ and the adversaries $\mathcal{S}_{\mathcal{A}}$, $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$. $\mathcal{F}_{\mathsf{disZK}}$ receives an index i and a parameter $m \in \mathbb{N}$ from the honest parties.

– If $P^* = P_i$, then $\mathcal{F}_{\mathsf{disZK}}$ receives, for each $u \in [m]$,

– $\{a_1^u, b_1^u, e_2^u\}$ from P_j,
$\{a_2^u, b_2^u\}$ from P_m and
$\{a_3^u, b_3^u, e_1^u\}$ from P_k
– $\mathcal{F}_{\mathsf{disZK}}$ sends $a_1^u, b_1^u, a_2^u, b_2^u, a_3^u, b_3^u, e_1^u, e_2^u$ to $\mathcal{S}_{\mathcal{A}}$, and sends inputs of P_H^* to $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$, where P_H^* is controlled by $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$.
– $\mathcal{S}_{\mathcal{A}}$ sends $\mathcal{F}_{\mathsf{disZK}}$ the command `accept` or `abort` with $(\mathsf{Select}, \mathcal{D})$ from $\mathcal{S}_{\mathcal{A}}$. Here, $\mathcal{D}$ denotes the pair of parties that $\mathcal{S}_{\mathcal{A}}$ wants to choose as dispute pair.
– If $\mathcal{F}_{\mathsf{disZK}}$ receives $\mathtt{abort}, \mathcal{D}$, command from $\mathcal{S}_{\mathcal{A}}$, then for each s $\mathcal{F}_{\mathsf{disZK}}$ sets $\mathsf{msg}_s = \mathtt{DP}$, where $\mathtt{DP} = \mathcal{D}$ if $P^* \in \mathcal{D}$, else $\mathtt{DP} = \mathcal{P} \setminus \mathcal{D}$.
– If $\mathcal{F}_{\mathsf{disZK}}$ receives `accept` from $\mathcal{S}_{\mathcal{A}}$ and if for some $u \in [m]$, $\sum_{j \neq k} a_j^u b_k^u - e_1^u - e_2^u \neq 0$, where $j, k \in [3]$, then $\mathcal{F}_{\mathsf{disZK}}$ sets $\mathsf{msg}_s = \mathtt{DP}$ to all the parties, where $\mathtt{DP} = \{P^*, P_l\}$, where $P_l \neq P^*$.
– If $\mathcal{F}_{\mathsf{disZK}}$ receives `accept` from $\mathcal{S}_{\mathcal{A}}$ and if for all $u \in [m]$, $\sum_{j \neq k} a_j^u b_k^u - e_1^u - e_2^u \neq 0$, where $j, k \in [3]$, holds, then $\mathcal{F}_{\mathsf{disZK}}$ sets $\mathsf{msg}_s = \mathtt{accept}$ for all s.

– If P_i is an honest party. Then for each $u \in [m]$,
  – Then $a_1^u, b_1^u, a_2^u, b_2^u, a_3^u, b_3^u, e_1^u, e_2^u$ to $\mathcal{F}_{\mathsf{disZK}}$.
  – If P^* is the corrupted party controlled by $\mathcal{S}_{\mathcal{A}}$ and P_H^* is the semi-honest party controlled by $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$, then $\mathcal{F}_{\mathsf{disZK}}$ sends P^*'s input to $\mathcal{S}_{\mathcal{A}}$ and P_H^*'s input to $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$.
  – $\mathcal{S}_{\mathcal{A}}$ sends `accept` or `abort` with $(\mathsf{Select}, \mathcal{D})$ from $\mathcal{S}_{\mathcal{A}}$. Here, $\mathcal{D}$ denotes the pair of parties that $\mathcal{S}_{\mathcal{A}}$ wants to choose as dispute pair.
  – If $\mathcal{S}_{\mathcal{A}}$ sends $\mathtt{abort}, \mathcal{D}$, $\mathcal{F}_{\mathsf{disZK}}$ sets $\mathtt{DP} = \mathcal{D}$, if $P^* \in \mathcal{D}$, else $\mathtt{DP} = \mathcal{P} \setminus \mathcal{D}$. $\mathcal{F}_{\mathsf{disZK}}$ sets $\mathsf{msg}_s = \mathtt{DP}$ for all s.
  – If $\mathcal{S}_{\mathcal{A}}$ sends `accept`, $\mathcal{F}_{\mathsf{disZK}}$ sets $\mathsf{msg}_s = \mathtt{accept}$ for all s.
  – $\mathcal{S}_{\mathcal{A}}$ sends it's view to $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$.

Output: Send $(\mathsf{Output}, \mathsf{msg}_s)$ for all $P_s \in \mathcal{P}$.

Fig. 13: Ideal functionality for Zero Knowledge Verification

In the triple generation protocol from Fig. 6, as mentioned, there are terms of the type $\langle \alpha_{\mathsf{x}} \rangle_{ij} \cdot \langle \alpha_{\mathsf{y}} \rangle_{ik}$ which can be computed and shared by a single party P_i locally. However, to verify the correctness of P_i's computation, the protocol relies on ZK verification. Specifically for this, we extend the distributed ZK protocol by Boyle *et al.* for our setting, where P_i acts as the prover and the remaining three parties participate as the verifiers.

Recall that in $\mathsf{tripGen}$, P_i along with P_j, P_m locally samples α^2. Following this, it computes $\delta_i^1 = \sum_{\substack{j,k \\ j \neq k}} \langle \alpha_{\mathsf{x}} \rangle_{ij} \langle \alpha_{\mathsf{y}} \rangle_{ik} - \delta_i^2$ and sends it to P_k, $k = i + 3$. Note that we require each party to act as the prover in exactly one instance of the ZK protocol, for proving correctness of this aggregated computation. To verify the correctness of P_i's computation, the remaining parties have to verify that the circuit $c(\langle \alpha_{\mathsf{x}} \rangle_{ij}, \langle \alpha_{\mathsf{y}} \rangle_{ij}, \langle \alpha_{\mathsf{x}} \rangle_{ik}, \langle \alpha_{\mathsf{y}} \rangle_{ik}, \langle \alpha_{\mathsf{x}} \rangle_{im}, \langle \alpha_{\mathsf{y}} \rangle_{im}, \delta_i)$ evaluates to 0.

Observe that the input to c is additively distributed among P_j, P_k, P_m. That is,

- P_j has $(a_1, b_1, e_2) = (\langle \alpha_{\mathsf{x}} \rangle_{ij}, \langle \alpha_{\mathsf{y}} \rangle_{ij}, \delta_i^2)$.
- P_k has $(a_3, b_3, e_1) = (\langle \alpha_{\mathsf{x}} \rangle_{ik}, \langle \alpha_{\mathsf{y}} \rangle_{ik}, \delta_i^1)$.
- P_m has $(a_2, b_2) = (\langle \alpha_{\mathsf{x}} \rangle_{im}, \langle \alpha_{\mathsf{y}} \rangle_{im})$.

where $\delta = \delta_i^1 + \delta_i^2$.

Note that, this follows the semantics of the circuit c described in (5), and hence a zero-knowledge proof on this circuit can be used to verify the correctness of P_i's computation. Instead of the naive verification, we use the same amortization used in [20] for verification of m number of c circuits.

For $\ell \in [m]$, let

$$(x^i_{7(\ell-1)+1}, \ldots, x^i_{7(\ell-1)+7}) = (a_1^\ell, b_1^\ell, a_2^\ell, b_2^\ell, a_3^\ell, b_3^\ell, e^\ell)$$
$$(x^j_{7(\ell-1)+1}, \ldots, x^j_{7(\ell-1)+7}) = (a_1^\ell, b_1^\ell, 0, 0, 0, 0, e_2^\ell)$$
$$(x^k_{7(\ell-1)+1}, \ldots, x^k_{7(\ell-1)+7}) = (0, 0, 0, 0, a_3^\ell, b_3^\ell, e_1^\ell)$$
$$(x^m_{7(\ell-1)+1}, \ldots, x^m_{7(\ell-1)+7}) = (0, 0, a_2^\ell, b_2^\ell, 0, 0, 0)$$

We construct a sub-circuit g that contains L of the smaller c circuits. That is, it takes $7L$ inputs and outputs the random linear combination of L outputs of the corresponding c circuits. Specifically,

$$g(x_1, \ldots, x_{7L}) = \sum_{k=1}^{L} \theta_k \cdot c(x_{7(k-1)+1}, \ldots, x_{7(k-1)+7L})$$

Finally, we set $M = m/L$ and define the verification circuit G which outputs a random linear combination of the g circuits outputs, that is,

$$G(x_1, \ldots, x_{7m}) = \sum_{k=1}^{M} \beta_k \cdot g(x_{7L(k-1)+1}, \ldots, x_{7L(k-1)+7L})$$

where θ_k and β_k are uniformly distributed over $\mathbb{F}$ and obtained by parties using $\mathcal{F}_{\mathsf{Coin}}$. The protocol outline is similar to the protocol from [20] and the complete description is given in Fig. 14.

In our implementation, we use the recursive variant of the ZK protocol described in [20] which achieves computation and communication efficiency while trading off the number of rounds of communication. In particular, it incurs a communication cost of $\eta(1 + 4\log m)$ elements for verification of m multiplication gates, for $\eta > \log(2 + \frac{5\log m+1}{\epsilon})$, where each element in $\mathbb{Z}_{2^\lambda}$ is lifted to a η-degree polynomial in $\mathbb{Z}_{2^\lambda}[x]/f(x)$. The computational cost the recursive variant of distributed ZK protocol is similar to [20]. For the verification of m multiplication, the computational costs are $32m$ and $7m$ multiplications over $\mathbb{Z}_{2^\lambda}[x]/f(x)$ for a prover and a verifier respectively. In our protocol, a party acts as a prover once and thrice as a verifier. Therefore, per party computational cost is $(32m + 3 \times 7m) = 53m$.

Protocol $\mathsf{disZK}(\mathcal{P}, [\![\mathsf{x}]\!], [\![\mathsf{y}]\!])$

- Step 1
- Parties invoke $\mathcal{F}_{\mathsf{Coin}}$ and receive random $\theta_1, \theta_2, \ldots, \theta_L \in \mathbb{F}$.
- P_i chooses random $\omega_1, \omega_2, \ldots, \omega_{7L} \in \mathbb{F}$.
- P_i defines $7L$ polynomials $f_1, f_2, \ldots, f_{7L} \in \mathbb{F}[x]$ of degree M such that for each $j \in [7L]$, $f_j(0) = \omega_j$ and $f_j(\ell) = x^i_{7L(\ell-1)+j}$, for all $\ell \in [M]$.
- P_i computes the coefficients of the $2M$-degree polynomial $p(x) \in \mathbb{F}[x]$ defined by $p = g(f_1, f_2, \ldots, f_{7L})$ where $g(x_1, x_2, \ldots, x_{7L}) = \sum_{k=1}^{L} \theta_k \cdot c(x_{7(k-1)+1}, \ldots, x_{7(k-1)+7})$ and $G(x_1, x_2, \ldots, x_{7m}) = \sum_{k=1}^{M} \beta_k \cdot g(x_{7L(k-1)+1}, \ldots, x_{7L(k-1)+7L})$ and $M = \frac{m}{L}$.
- Let $a_0, a_1, \ldots, a_{2M}$ be the coefficients obtained. P_i defines $\pi = (\omega_1, \omega_2, \ldots, \omega_{7L}, a_0, a_1, \ldots, a_{2M})$.
- P_i and P_{i+1} randomly pick $\pi^{i+1} \in \mathbb{F}^{7L+2M+1}$. Similarly, P_i and P_{i+2} randomly pick $\pi^{i+2} \in \mathbb{F}^{7L+2M+1}$.
- P_i sends $\pi^{i+3} = \pi - \pi^{i+1} - \pi^{i+2}$ to P_{i+3}.
- Step 2
- Parties invoke $\mathcal{F}_{\mathsf{Coin}}$ and receive random $\beta_1, \beta_2, \ldots, \beta_M \in \mathbb{F}$ and $r \in \mathbb{F} \setminus \{0, \ldots, M\}$.
- Each party P_t, where $t \in \{i+1, i+2, i+3\}$ does the following:
  - Parse the message π^t as $(\omega^t_1, \omega^t_2, \ldots, \omega^t_{7L}, a^t_0, a^t_1, \ldots, a^t_{2M})$.
  - Define $7L$ polynomials $f^t_1, f^t_2, \ldots, f^t_{7L} \in \mathbb{F}[x]$ of degree M such that for each $j \in [7L]$, $f^t_j(0) = \omega^t_j$ and $f^t_j(\ell) = x_{7L(\ell-1)+j}$ for all $\ell \in [M]$.
  - Compute $f^t_j(r)$ for each $j \in [7L]$ and $p^t_r = \sum_{j=0}^{2M} a^t_j \cdot r^j$.
  - Compute $b^t = \sum_{j=1}^{M} \beta_j \cdot \sum_{k=0}^{2M} a^t_k \cdot j^k$.
- P_t sends $f^t_1(r), f^t_2(r), \ldots, f^t_{7L}(r), p^t_r, b^t_r$ to P_{i+2}, where $t \in \{i+1, i+3\}$.
- Step 3
- Upon receiving the message from P_{i+1}, P_{i+3} in round 2, P_{i+2} computes $f'_j(r) = f^{i+1}_j(r) + f^{i+2}_j(r) + f^{i+3}_j(r)$, $p_r = p^{i+1}_r + p^{i+2}_r + p^{i+3}_r$ and $b = b^{i+1} + b^{i+2} + b^{i+3}$.
- P_{i+2} checks if $p_r = g(f'_1(r), \ldots, f'_{7L}(r))$ and $b = 0$. If either of the equalities does not hold, then it outputs `abort`. Otherwise, it outputs accept.
- If P_{i+2} outputs `abort`, then $P_t, t \in \{i+1, i+2, i+3\}$, broadcasts $H(f^t_1(r), \ldots, f^t_{7L}(r), a^t_0, \ldots, a^t_{2M})$ where H is a CRH.
- P_i, P_{i+2} compute $H(f^t_1(r), \ldots, f^t_{7L}(r), a^t_0, \ldots, a^t_{2M})$. If for some P_t, P_i, P_{i+2} get different value, then:
- If P_i accuses P_t, parties output $\mathtt{DP} = \{P_i, P_t\}$.
- If P_{i+2} accuses P_t, parties output $\mathtt{DP} = \{P_{i+2}, P_t\}$.
- If P_i, P_{i+2} accuse P_t, parties output $\mathtt{DP} = \{P_i, P_t\}$.
- If neither P_i nor P_{i+2} accuses, parties output $\mathtt{DP} = \{P_i, P_{i+2}\}$.

Fig. 14: Distributed Zero-Knowledge Verification Protocol

Lemma 3 (Communication). *Protocol* disZK *requires a communication of* $4(16\sqrt{m}+5)$ *elements in the preprocessing phase for the verification of* m *multiplication gates.*

Proof. For one instance of disZK, the prover picks $\omega_1^j, \ldots, \omega_{7L}^j$ for all verifiers P_j, and completes the rest of the proof generation computation locally. Finally, the prover only needs to send the co-efficients to the verifiers. For this, the prover and two of the verifiers pick random values using their common keys, and then prover sends the remaining share of the co-efficients to the third verifier. Therefore to send the proof, the prover communicates $(2M+1)$ elements. Further, two of the verifiers communicate $(7L+2)$ bits each, thus requiring an overall communication of $((2M+1)+2(7L+2))$ elements for a proof. Since, each party runs the protocol as a prover, the total communication for m gates will be $4(14L+2M+5)$ elements. We set the parameters $M = L = \sqrt{m}$, then total communication cost will be $4(16\sqrt{m}+5)$ elements.

Therefore, for per multiplication gate communication cost due to disZK is $\frac{64}{\sqrt{m}} + \frac{20}{m}$ elements, which does not contribute to additional communication cost for evaluating a multiplication gate in the preprocessing phase for large enough m.

Lemma 4. *The protocol* disZK FaF*-securely computes* $\mathcal{F}_{\mathsf{disZK}}$ *with identifying a dispute set if aborts in the presence of one malicious party and one semi-honest party, with statistical error* $\frac{2M+1}{|\mathbb{F}|-M}$.

Proof. We construct an ideal world simulator $\mathcal{S}_{\mathcal{A}}$ for two different cases.

Case-I The prover P_i is corrupted. In this case $\mathcal{S}_{\mathcal{A}}$ receives the inputs of P_i from $\mathcal{F}_{\mathsf{disZK}}$ and so $\mathcal{S}_{\mathcal{A}}$ knows the inputs of the honest parties. Thus, $\mathcal{S}_{\mathcal{A}}$ can simulate exactly the role of the honest parties in the protocol. $\mathcal{S}_{\mathcal{A}}$ invoke the real world adversary to receive the proof sent by P_i to the three other parties. $\mathcal{S}_{\mathcal{A}}$ simulates $\mathcal{F}_{\mathsf{Coin}}$ handing r and random co-efficients to the parties and follows the instruction of three verifiers.

If P_i is acting honestly in the execution of the main protocol and the output of the circuit is $c = 0$ for every multiplication, then $\mathcal{S}_{\mathcal{A}}$ sends $\mathcal{F}_{\mathsf{disZK}}$ the output of the verifiers. The simulation is perfect.

If the output of the circuit $c \neq 0$ for some multiplication gate, then $\mathcal{F}_{\mathsf{disZK}}$ outputs aborts to the parties along with a dispute pair DP. Thus, if the honest parties simulated by $\mathcal{S}_{\mathcal{A}}$ output accept, then $\mathcal{S}_{\mathcal{A}}$ outputs fail and halts. The only difference between the simulation and the real execution is the event that $\mathcal{S}_{\mathcal{A}}$ outputs fail. Observe that this happens iff the following events happen:

1. The random co-efficients $\theta_1, \ldots, \theta_L$ were chosen such that the output of the g gate is 0. OR
2. $p(r) = g(f_1(r), \ldots, f_{7L}(r))$ but $p(x) \neq g(f_1(x), \ldots, f_{7L}(x))$. OR
3. If the random linear combination using $\beta_1, \ldots, \beta_M$ yields that the output of G is 0.

(1) and (3) can happen with probability $\frac{1}{|\mathbb{F}|}$. (2) can happen if r is a root of $p(x) - g(f_1(x), \ldots, f_{7L}(x))$. Since, $p(x) - g(f_1(x), \ldots, f_{7L}(x))$ is a non-zero polynomial of degree at most $2M$, the probability that a randomly picked r from $\mathbb{F} \setminus \{0, 1, \ldots, M\}$ is a root of the above polynomial is $\frac{2M}{|\mathbb{F}|-M-1}$. Therefore, the probability that $\mathcal{S}_{\mathcal{A}}$ fails $< \frac{2M+1}{|\mathbb{F}|-M}$.

If the proof given by P_i is such that the verifiers output reject, then simulation also does the same, and follow the instructions of the verifiers to generate the correct messages and broadcast the hash of those messages. Finally, $\mathcal{S}_{\mathcal{A}}$ outputs a dispute pair, $\texttt{DP} = \{P_i, P_v\}$, if P_i accuses P_v, where $v \in \{i+1, i+2, i+3\}$, otherwise $v = i+2$.

Subcase 1: Let P_{i+1} be the semi-honest party. Let $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ be the simulator. Since, P_{i+1} is not receiving any messages from other verifiers, nothing to simulate till round 2.

If P_i's proof is such that P_{i+2} outputs reject, $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ broadcasts the hash of the correct verifiers' messages on behalf of P_{i+2}, P_{i+3} and outputs $\texttt{DP}$.

If P_i's proof is such that P_{i+2} outputs accept, then $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ outputs accept. In both the cases simulation is perfect.

Subcase 2: Let P_{i+2} be the semi-honest party. Since, P_i is the malicious party, $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ has inputs of P_i from $\mathcal{S}_{\mathcal{A}}$, and the messages sent to the honest verifiers. So, the simulator $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ follows the protocol instructions correctly and sends correct messages on behalf of the P_{i+1}, P_{i+3}.

Subcase 3: Let P_{i+3} be the semi-honest party. This simulation is the same as the Subcase 1.

Case-II The prover P_i is honest. In this case the simulator $\mathcal{S}_{\mathcal{A}}$ receives the inputs known to the corrupted verifier.

$\mathcal{S}_{\mathcal{A}}$ simulates $\mathcal{F}_{\mathsf{Coin}}$ handing $\theta_1, \ldots, \theta_L \in \mathbb{F}$ to the parties and then it chooses a random $\pi^j \in \mathbb{F}^{7L+2M+1}$, corresponding to the corrupt verifier P_j.

Then it simulates the ideal functionality $\mathcal{F}_{\mathsf{Coin}}$ handing a random $r \in \mathbb{F} \setminus \{0, 1, \ldots, M\}$ and co-efficients $\beta_1, \ldots, \beta_M \in \mathbb{F}$ to the parties. Since, $\mathcal{S}_{\mathcal{A}}$ knows the corrupted party's inputs, it can compute the message that should be sent by the corrupted party $f_1^j(r), \ldots, f_{7L}^j(r), p_r^j, b^j$.

If $j = i+1$ or $i+3$ and it sends message at the end of round 2 to the honest P_{i+2}, who decides whether to accept or not, then upon receiving the message, $\mathcal{S}_{\mathcal{A}}$ can conclude whether P_{i+2} will reject or accept by comparing it to the message that should have sent.

- If the received message on behalf of P_{i+2} is the same as the message that should have sent, the $\mathcal{S}_{\mathcal{A}}$ outputs accept and the simulation is perfect.
- If not, then P_{i+2} would have output reject. So, $\mathcal{S}_{\mathcal{A}}$ outputs reject and does the following:
  1. $\mathcal{S}_{\mathcal{A}}$ samples $f'_1(r), \ldots, f'_{7L}(r)$ uniformly at random.
  2. $\mathcal{S}_{\mathcal{A}}$ computes $g_r = g((f_1^j(r) + f'_1(r)), \ldots, (f_{7L}^j(r) + f'_{7L}(r)))$ and sets $p'_r = g_r - p_r^j$ and $b' = -b^j$.
  3. $\mathcal{S}_{\mathcal{A}}$ picks $f''_1(r), \ldots, f''_{7L}(r), p''_r, b''$ uniformly at random and sets as P_{i+2}'s message. It computes $(f'_1(r) - f''_1(r)), \ldots, (f'_{7L}(r) - f''_{7L}(r)), (p'_r - p''_r), (b' -$

b'') and sets as P_{i+3}'s message. Then $\mathcal{S}_\mathcal{A}$ broadcasts hash both the messages on behalf of the honest verifiers.

4. $\mathcal{S}_\mathcal{A}$ accuses P_j and outputs $\texttt{DP} = \{P_i, P_j\}$ or $\{P_{i+2}, P_j\}$ according to the protocol instructions.

Without loss of generality, Let us assume that $j = i + 1$.

Subcase 1: Let P_i be the semi-honest party. $\mathcal{S}_{\mathcal{A}_\mathcal{H}}$ has all the inputs of P_i, correct simulation is trivial.

Subcase 2: Let P_{i+2} be the semi-honest party.

– $\mathcal{S}_{\mathcal{A}_\mathcal{H}}$ picks $\pi^{i+2} \in \mathbb{F}^{7L+2M+1}$ uniformly and sends to P_{i+2} on behalf of P_i.
– $\mathcal{S}_{\mathcal{A}_\mathcal{H}}$ picks $f_1^{i+3}(r), \dots, f_{7L}^{i+3}(r)$ uniformly.
– It computes $p_r^{i+3} = g((\sum_{k=1}^{3} f_1^{i+k}(r)), \dots, (\sum_{k=1}^{3} f_{7L}^{i+k}(r))) - p_r^{i+2} - p_r^{i+1}$, $b^{i+3} = -b^{i+1} - b^{i+2}$.
– $\mathcal{S}_{\mathcal{A}_\mathcal{H}}$ sends $f_1^{i+3}(r), \dots, f_{7L}^{i+3}(r), p_r^{i+3}, b^{i+3}$ to P_{i+2}.

If P_{i+2} outputs reject, $\mathcal{S}_{\mathcal{A}_\mathcal{H}}$ outputs $\texttt{DP}$ according to the protocol instruction.

Subcase 3: Let P_{i+3} be the semi-honest party. $\mathcal{S}_{\mathcal{A}_\mathcal{H}}$ picks $\pi^{i+2} \in \mathbb{F}^{7L+2M+1}$ uniformly and sends to P_{i+2} on behalf of P_i. If $\mathcal{S}_\mathcal{A}$ output reject at round 3, then $\mathcal{S}_{\mathcal{A}_\mathcal{H}}$ picks $f_1^{i+3}(r), \dots, f_{7L}^{i+3}(r), p_r^{i+3}, b^{i+3}$ similar to Subcase 2 and broadcasts the hash values. Finally, $\mathcal{S}_{\mathcal{A}_\mathcal{H}}$ outputs $\texttt{DP}$ and the simulation is perfect.

If $j = i + 2$ and thus $\mathcal{S}_\mathcal{A}$ needs to simulate the message sent by the honest verifiers P_{i+1}, P_{i+3}. Thus to compute the message sent by the $P_t, t \in \{i{+}1, i{+}3\}$, $\mathcal{S}_\mathcal{A}$ does the following:

1. It chooses random $f_1^t(r), \dots, f_{7L}^t(r) \in \mathbb{F}$, and computes $g_r = g((\sum_{k=1}^{3} f_1^{i+k}(r)), \dots, (\sum_{k=1}^{3} f_{7L}^{i+k}(r)))$
2. $\mathcal{S}_\mathcal{A}$ picks p_r^{i+1} uniformly and sets $p_r^{i+3} = g_r - p_r^{i+1} - p_r^{i+2}$.
3. $\mathcal{S}_\mathcal{A}$ sets b^{i+1}, b^{i+3} such that $b^{i+1} + b^{i+2} + b^{i+3} = 0$

$\mathcal{S}_\mathcal{A}$ sends $f_1^t(r), \dots, f_{7L}^t(r), p_r^t, b^t$ to P_{i+2} for $t \in \{i + 1, i + 3\}$.

If P_{i+2} outputs accept, then the simulation is perfect.

If P_{i+2} outputs reject, $\mathcal{S}_\mathcal{A}$ broadcasts $\texttt{H}(f_1^t(r), \dots, f_{7L}^t(r), p_r^t, b^t)$, for $t \in \{i + 1, i + 3\}$ and outputs $\texttt{DP} = \{P_i, P_{i+2}\}$.

Subcase 1: Let P_i be the semi-honest party. $\mathcal{S}_{\mathcal{A}_\mathcal{H}}$ has all the inputs of the prover P_i, therefore the simulation is trivial.

Subcase 2: Let P_{i+1} be the semi-honest party.

$\mathcal{S}_{\mathcal{A}_\mathcal{H}}$ sends $\pi^{i+1} \in_R \mathbb{F}^{7L+2M+1}$ to P_{i+1}.

$\mathcal{S}_{\mathcal{A}_\mathcal{H}}$ computes the message of P_{i+3} similar to the case when P_{i+1} was malicious and P_{i+2} was semi-honest.

If P_{i+2} outputs reject, $\mathcal{S}_{\mathcal{A}_\mathcal{H}}$ broadcasts the hash of the same message and follows the protocol instructions and outputs $\texttt{DP}$.

As described in Section 2, this distributed zero-knowledge can be extended to the Ring $\mathbb{Z}_{2^\lambda}$, this follows from the [20, Theorem 4.7]. The aforementioned theorem can be adapted our construction in the following way:

Lemma 5. *Let η be such that $2^\eta > 2M + 1$. Then, the protocol* disZK *securely computes $\mathcal{F}_{\mathsf{disZK}}$ with identifying a dispute pair if aborts in the presence of one malicious party with statistical error* $\frac{2^{(\lambda-1)\eta} \cdot 2M+1}{2^{\lambda\eta}-M}$

Due to this, the communication blows up by a factor of η.

B Proof of Necessity of Oblivious Transfer

In this section, we provide details of the construction of OT protocol π_{OT} (Fig. 15) from the (t, h^*)-`FaF` secure protocol π_f as described in §3. We also describe the simulators $\mathcal{S}_S$ and $\mathcal{S}_R$ for corrupt sender and corrupt receiver respectively in Fig. 16 and Fig. 17. We conclude with an indistinguishability argument to complete the proof of Theorem 3.

Protocol π_{OT}

- **Input, Output:** P_S has input (m_0, m_1) and P_R has input b. P_R outputs m_b.
- **Primitives:** n-party (t, h^*)-`FaF` secure protocol π_f for computing $f((m_0, m_1), \perp, \dots, \perp, b) = (\perp, \perp, \dots, \perp, m_b)$.

– P_S emulates the role of $Q_S = \{P_1, P_2, \dots P_{t+h^*}\}$ in π_f.
– P_R emulates the role of $Q_R = \{P_{t+h^*+1}, \dots, P_n\}$ in π_f.
– P_S and P_R execute π_f emulating the roles of parties in Q_S and Q_R respectively.
– P_R's output m_b, is the same the output of P_n in π_f.

Fig. 15: 1-out-of-2 OT Protocol

Simulator $\mathcal{S}_S$

Let $H = \{P_1, \dots, P_{h^*}\}$ and $I = Q_S \setminus H$.
Let $\mathcal{S}_{\mathcal{A}}$ be the malicious simulator for $I = \{P_{h^*+1}, P_{h^*+2}, \dots, P_{t+h^*}\}$ in π_f and $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ be the semi-honest simulator for $H = \{P_1, \dots, P_{h^*}\}$ in π_f.
– $\mathcal{S}_S$ runs $\mathcal{S}_{\mathcal{A}}$, which in turn sends its view to $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$.
– $\mathcal{S}_S$ runs $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ with (m_0, m_1) as the input of P_1.
– $\mathcal{S}_S$ outputs the view of $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$.

Fig. 16: Simulator for corrupt sender P_S

Simulator $\mathcal{S}_R$

Let $H = \{P_{2t+h^*+1}, \ldots, P_n\}$ and $I = Q_R \setminus H$.
Let $\mathcal{S}_\mathcal{A}$ be the malicious simulator for $I = \{P_{t+h^*+1}, \ldots, P_{2t+h^*}\}$ in π_f and $\mathcal{S}_{\mathcal{A}_\mathcal{H}}$ be the semi-honest simulator for $H = \{P_{2t+h^*+1}, \ldots, P_n\}$ in π_f.

- $\mathcal{S}_R$ runs $\mathcal{S}_\mathcal{A}$, which in turn sends its view to $\mathcal{S}_{\mathcal{A}_\mathcal{H}}$.
- $\mathcal{S}_R$ runs $\mathcal{S}_{\mathcal{A}_\mathcal{H}}$ with b as the input and m_b as the output of P_n.
- $\mathcal{S}_R$ outputs the view of $\mathcal{S}_{\mathcal{A}_\mathcal{H}}$.

Fig. 17: Simulator for corrupt receiver P_R

Indistinguishability Proof. The security of π_f guarantees that

$$\{\mathcal{S}_{\mathcal{A}_\mathcal{H}}(1^\kappa, (m_0, m_1)), \mathsf{Output}_{P_n}((m_0, m_1), \bot, \ldots, \bot, b)\} \stackrel{c}{\equiv} \{\mathsf{View}^{\pi_f}_{\mathcal{A},\mathcal{A}_\mathcal{H}}(1^\kappa, (m_0, m_1)), \mathsf{Output}^{\pi_f}_{P_n}((m_0, m_1), \bot, \ldots, \bot b)\} \tag{6}$$

where $\mathsf{View}^{\pi_f}_{\mathcal{A},\mathcal{A}_\mathcal{H}}(1^\kappa, (m_0, m_1))$ denotes the view of $\mathcal{A}_\mathcal{H}$ with the view of $\mathcal{A}$ in π_f as input, $\mathsf{Output}^{\pi_f}_{P_n}((m_0, m_1), \bot, \ldots, \bot, b)$ denotes output of the honest party P_n in π_f and $\mathsf{Output}_{P_n}((m_0, m_1), \bot, \ldots, \bot, b)$ denotes output of P_n in the functionality f. From construction of P_S in Fig. 15 it can be seen that

$$\{\mathsf{View}^{\pi_f}_{\mathcal{A},\mathcal{A}_\mathcal{H}}(1^\kappa, (m_0, m_1)), \mathsf{Output}^{\pi_f}_{P_n}((m_0, m_1), \bot, \ldots, \bot, b)\} \equiv \{\mathsf{View}^{\pi_\mathsf{OT}}_{\mathcal{A}_\mathsf{OT}}(1^\kappa, (m_0, m_1)), \mathsf{Output}^{\pi_\mathsf{OT}}_{P_R}((m_0, m_1), b)\} \tag{7}$$

where $\mathcal{A}_\mathsf{OT}$ is the semi-honest adversary corrupting the sender P_S in π_OT, $\mathsf{View}^{\pi_\mathsf{OT}}_{\mathcal{A}_\mathsf{OT}}(1^\kappa, (m_0, m_1))$ is the view of $\mathcal{A}_\mathsf{OT}$ in π_OT and $\mathsf{Output}^{\pi_\mathsf{OT}}_{P_R}((m_0, m_1), b)$ is the output of the receiver P_R in π_OT. Similarly, from the construction of $\mathcal{S}_S$ in Fig. 16, we have

$$\{\mathcal{S}_{\mathcal{A}_\mathcal{H}}(1^\kappa, (m_0, m_1)), \mathsf{Output}_{P_n}((m_0, m_1), \bot, \ldots, \bot, b)\} \equiv \{\mathcal{S}_S(1^\kappa, (m_0, m_1)), \mathsf{Output}_{P_R}((m_0, m_1), b)\} \tag{8}$$

where $\mathsf{Output}_{P_R}((m_0, m_1), b)$ is the output of P_R in the 1-out-of-2 OT functionality. From equations (6), (7) and (8), we have

$$\{\mathcal{S}_S(1^\kappa, (m_0, m_1)), \mathsf{Output}_{P_R}((m_0, m_1), b)\} \stackrel{c}{\equiv} \{\mathsf{View}^{\pi_\mathsf{OT}}_{\mathcal{A}_\mathsf{OT}}(1^\kappa, (m_0, m_1)), \mathsf{Output}^{\pi_\mathsf{OT}}_{P_R}((m_0, m_1), b)\}$$

This proves that the view generated by $\mathcal{S}_S$ is computationally indistinguishable from the view of P_S in π_OT. This also proves the correctness of P_R's output in π_OT since π_f does not abort when the corrupt parties are semi-honest, since the output of P_n in π_f is indistinguishable from P_n's output in f as shown.

The simulator $\mathcal{S}_R$ for the case when P_R is corrupted can be constructed similar to $\mathcal{S}_S$ with $I = \{P_{t+h^*+1}, \ldots, P_{2t+h^*}\}$ and $H = \{P_{2t+h^*+1}, \ldots, P_n\}$ and is provided in Fig. 17. In the case when $n < 2t + 2h^*$, we populate the H set first with up to h^* parties and then set $I = Q_R \setminus H$. Specifically, since the existence of a (t, h^*)-`FaF` secure (abort) protocol requires $n \geq t + h^* + 1$ [3], we have

$0 \leq |I| \leq t$ and $1 \leq |H| \leq h^*$ in this case and the construction of $\mathcal{S}_R$ proceeds similar to that of $\mathcal{S}_S$.

This proves the necessity of semi-honest-OT for (t, h^*)-FaF secure protocol where $t + h^* < n \leq 2t + 2h^*$. Moreover, the sufficiency of OT for the same is given in [3, Theorem 4.1].

C Functionalities and Security Proofs

In this section, we present simulation-based security proofs of the subprotocols used as building blocks for this work. For a protocol Π, we have two simulators, $\mathcal{S}_{\mathcal{A}}$, which represents the ideal-world malicious party, and $\mathcal{S}_{\mathcal{A}_{\varkappa}}$ that represents the ideal-world semi-honest party. Corresponding to the protocol Π, we will call the simulators as $\mathcal{S}_{\mathcal{A}\,\Pi}$ and $\mathcal{S}_{\mathcal{A}_{\varkappa}\,\Pi}$. To relax from the heavy notational burden while describing the simulators $\mathcal{S}_{\mathcal{A}\,\Pi}$ $\mathcal{S}_{\mathcal{A}_{\varkappa}\,\Pi}$, we will refer to them as $\mathcal{S}_{\mathcal{A}}$ and $\mathcal{S}_{\mathcal{A}_{\varkappa}}$. On the other hand, if a simulator for a protocol, say Π_1 executes another simulator for a protocol, say Π_2, then we refer to the latter simulator as $\mathcal{S}_{\mathcal{A}\,\Pi_2}$ and $\mathcal{S}_{\mathcal{A}_{\varkappa}\,\Pi_2}$. Furthermore, we indicate by ${\mathcal{S}_{\mathcal{A}}}^{P_i}$ that the simulator corresponding to the scenario when P_i is maliciously corrupted, and similarly, ${\mathcal{S}_{\mathcal{A}_{\varkappa}}}^{P_i}$ indicates P_i is semi-honest. In the protocols in which parties are either sender or receiver, we use ${\mathcal{S}_{\mathcal{A}}}^{s}$ and ${\mathcal{S}_{\mathcal{A}}}^{r}$ for indicating the corrupt sender, P_s or the corrupt receiver, P_r, respectively. Similar for the $\mathcal{S}_{\mathcal{A}_{\varkappa}}$ as well.

As mentioned, our protocol is secure against a mixed adversary corrupting one malicious and one semi-honest party. This is due to the fact that the malicious adversary even with the view of the semi-honest party cannot get any additional advantage. Simulation for mixed security is similar to the simulation of the FaF-security, we demonstrate this by providing the simulator for the sharing protocol $[\![\cdot]\!]$-Sh. Since the simulation is similar we omit the details for the other protocols.

C.1 Sharing and Reconstruction Protocols

In this section we provide the ideal functionalities and the corresponding simulators for the sharing and reconstruction protocol from §4.

$[\![\cdot]\!]$-Sh *Functionality:* The ideal functionality for $[\![\cdot]\!]$-Sh (Fig. 1) appears in Fig. 18.

Functionality $\mathcal{F}_{\mathsf{Sh}}$

$\mathcal{F}_{\mathsf{Sh}}$ interacts with the parties in $\mathcal{P}$ and the adversaries $\mathcal{S}_{\mathcal{A}}$ and $\mathcal{S}_{\mathcal{A}_{\varkappa}}$.

- $\mathcal{F}_{\mathsf{Sh}}$ receives $(\mathsf{Input}, \mathsf{v})$ from P_c(client). Let P^* be the party corrupted by $\mathcal{S}_{\mathcal{A}}$.
- $\mathcal{F}_{\mathsf{Sh}}$ receives `continue` or `abort` with $(\mathsf{Select}, \mathcal{D})$ from $\mathcal{S}_{\mathcal{A}}$. Here, $\mathcal{D}$ denotes the pair of parties that $\mathcal{S}_{\mathcal{A}}$ wants to choose as dispute pair.
- If $\mathcal{F}_{\mathsf{Sh}}$ receives `continue`, randomly picks $\langle \alpha_{\mathsf{v}} \rangle_{ij} \in \mathbb{Z}_{2^\ell}$, for $1 \leq i < j \leq 4$ and compute $\beta_{\mathsf{v}} = \mathsf{v} + \sum_{(i,j)} \langle \alpha_{\mathsf{v}} \rangle_{ij}$. Set $\mathsf{msg}_s = (\beta_{\mathsf{v}}, \{\langle \alpha_{\mathsf{v}} \rangle_{st}\}_{t \in [4], s \neq t})$, for each $P_s \in \mathcal{P}$.
- Else if $\mathcal{F}_{\mathsf{Sh}}$ receives `abort`, then:

– If $P^* \in \mathcal{D}$, then set $\texttt{DP} = \mathcal{D}$ and $\mathsf{msg}_s = \texttt{DP}$ for each $P_s \in \mathcal{P}$.
– Else set $\texttt{DP} = \mathcal{P} \backslash \mathcal{D}$ and $\mathsf{msg}_s = \texttt{DP}$ for each $P_s \in \mathcal{P}$.
– $\mathcal{S}_{\mathcal{A}}$ sends it's view to $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$.

Output: Send $(\mathsf{Output}, \mathsf{msg}_s)$ to $P_s \in \mathcal{P}$.

Fig. 18: Ideal functionality for $[\![\cdot]\!]$-Sh

$[\![\cdot]\!]$-Sh *Simulator:* Simulator for $[\![\cdot]\!]$-Sh (Fig. 1) is provided in Fig. 19.

Simulator $\mathcal{S}_{\mathcal{A}}{}^{P_i}$, $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}{}^{P_j}$

Malicious Simulation:

Preprocessing:
– Let P_i be the corrupt party. Then $\mathcal{S}_{\mathcal{A}}$ emulates $\mathcal{F}_{\mathsf{setup}}$ so that $\mathcal{S}_{\mathcal{A}}$ and P_i get $\{\langle \alpha_{\mathsf{v}} \rangle_{ij}\}$, where $j \neq i$.
– If P_i is the client, then P_i and $\mathcal{S}_{\mathcal{A}}$ pick $\langle \alpha_{\mathsf{v}} \rangle_{jk}$ $1 \leq j < k \leq 4$.

Online:
– If P_i is the client, then $\mathcal{S}_{\mathcal{A}}$ either receives β_{v} from P_i outputs $\texttt{DP} = \{P_i, P_k\}$, for some $k \neq i$.
– If P_i is not the client. $\mathcal{S}_{\mathcal{A}}$ invokes $\mathcal{F}_{\mathsf{Sh}}$ and receives β_{v}. It sends β_{v} to P_i.

Semi-Honest Simulation:

Preprocessing:
– Let P_j be the semi-honest party. Then $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ has $\{\langle \alpha_{\mathsf{v}} \rangle_{ik}\}$ from $\mathcal{S}_{\mathcal{A}}$. $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ and P_j pick $\{\langle \alpha_{\mathsf{v}} \rangle_{jl}\}$, where $l \neq \{i, j\}$.
– If P_j is the client, then $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ and P_j pick $\langle \alpha_{\mathsf{v}} \rangle_{kl}$, where $k, l \notin \{i, j\}$.

Online:
– If P_j be the client, then $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ either receives β_{v} from P_j or outputs $\texttt{DP} = \{P_i, P_k\}$, for some $k \neq i$.
– If P_j is not the client. $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ sends β_{v} to P_j, where $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ received β_{v} from $\mathcal{S}_{\mathcal{A}}$.

Fig. 19: Simulator $\mathcal{S}_{\mathcal{A}}$ for $[\![\cdot]\!]$-Sh

Lemma 6 (Security). *The protocol $[\![\cdot]\!]$-Sh, described in Fig. 1, realizes $\mathcal{F}_{\mathsf{Sh}}$ (Fig. 18) with computational security in the $(\mathcal{F}_{\mathsf{setup}}, \mathcal{F}_{\mathsf{jmp4}})$-hybrid model against $(1,1)$-FaF adversaries $\mathcal{A}$, $\mathcal{A}_{\mathcal{H}}$, controlling P_i, P_j respectively.*

Proof. Claim 1: The simulator $\mathcal{S}_{\mathcal{A}}{}^{P_i}$, described in Fig. 19, generates a transcript that is indistinguishable from $\mathcal{A}$'s view.

Proof of Claim 1: Case I: If P_i is the client, then for all pairs (j,k), $\langle \alpha_{\mathsf{v}} \rangle_{jk}$ are obtained using the corresponding common key, which is indistinguishable from randomly picked values and P_i computes β_{v}. Therefore, the transcript $\tau = \{\{\langle \alpha_{\mathsf{v}} \rangle_{jk}\}_{j,k}, \beta_{\mathsf{v}}\}$, generated by $\mathcal{S}_{\mathcal{A}}{}^{P_i}$, is indistinguishable from a real transcript.

Case II: If P_i is not the client, then P_i's view consists of $\{\beta_{\mathsf{v}}, \{\langle \alpha_{\mathsf{v}} \rangle_{ij}\}_{j \neq i}\}$, where $\beta_{\mathsf{v}} = \mathsf{v} - \sum_{(j,k)} \langle \alpha_{\mathsf{v}} \rangle_{jk}$. According to P_i, β_{v} remains random, since $\mathsf{v}, \langle \alpha_{\mathsf{v}} \rangle_{jk}$, for $j, k \neq i$, are unknown to P_i. Therefore the transcript generated by ${\mathcal{S}_{\mathcal{A}}}^{P_i}$ is indistinguishable from P_i's view.

Claim 2: The simulator ${\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}}^{P_j}$, that has P_j's input, output, and the view generated by ${\mathcal{S}_{\mathcal{A}}}^{P_i}$, generates a transcript which is indistinguishable from P_j's view (view of P_j along with P_i's view).

Proof of Claim 2: Case I: If P_j is the client, then it is easy to see that the transcript generated by ${\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}}^{P_j}$ is computationally indistinguishable from $\mathcal{A}_{\mathcal{H}}$'s view.

Case II: If P_j is not the client, $\mathcal{A}_{\mathcal{H}}$'s view consists of $\{\beta_{\mathsf{v}}, \langle\alpha_{\mathsf{v}}\rangle_{ij}, \{\langle\alpha_{\mathsf{v}}\rangle_{ik}\}_{k\neq i,j}, \{\langle\alpha_{\mathsf{v}}\rangle_{jk}\}_{k\neq i,j}\}$. Since P_j misses the share $\langle\alpha_{\mathsf{v}}\rangle_{kl}$, for $(k,l) \neq (i,j)$, β_{v} is random to P_j. Therefore the transcript generated by ${\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}}^{P_j}$ is computationally indistinguishable from $\mathcal{A}_{\mathcal{H}}$'s view.

$[\![\cdot]\!]$-Sh *Functionality for mixed-security:* The ideal functionality for $[\![\cdot]\!]$-Sh (Fig. 1) appears in Fig. 18.

Functionality $\mathcal{F}_{\mathsf{Sh}}^{\mathsf{mixed}}$

$\mathcal{F}_{\mathsf{Sh}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}$.

– $\mathcal{F}_{\mathsf{Sh}}^{\mathsf{mixed}}$ receives $(\mathsf{Input}, \mathsf{v})$ from $P_c(\mathsf{client})$. Let P_m^* be the malicious and P_h^* be the semi-honest party corrupted by $\mathcal{S}$.
– $\mathcal{F}_{\mathsf{Sh}}^{\mathsf{mixed}}$ receives `continue` or `abort` with $(\mathsf{Select}, \mathcal{D})$ from $\mathcal{S}$. Here, $\mathcal{D}$ denotes the pair of parties that $\mathcal{S}$ wants to choose as dispute pair.
– If $\mathcal{F}_{\mathsf{Sh}}^{\mathsf{mixed}}$ receives `continue`, randomly picks $\langle\alpha_{\mathsf{v}}\rangle_{ij} \in \mathbb{Z}_{2^\ell}$, for $1 \leq i < j \leq 4$ and compute $\beta_{\mathsf{v}} = \mathsf{v} + \sum_{(i,j)} \langle\alpha_{\mathsf{v}}\rangle_{ij}$. Set $\mathsf{msg}_s = (\beta_{\mathsf{v}}, \{\langle\alpha_{\mathsf{v}}\rangle_{st}\}_{t\in[4],s\neq t})$, for each $P_s \in \mathcal{P}$.
– Else if $\mathcal{F}_{\mathsf{Sh}}^{\mathsf{mixed}}$ receives `abort`, then:
  – If $P_m^* \in \mathcal{D}$, then set $\mathtt{DP} = \mathcal{D}$ and $\mathsf{msg}_s = \mathtt{DP}$ for each $P_s \in \mathcal{P}$.
  – Else set $\mathtt{DP} = \mathcal{P}\backslash\mathcal{D}$ and $\mathsf{msg}_s = \mathtt{DP}$ for each $P_s \in \mathcal{P}$.

Output: Send $(\mathsf{Output}, \mathsf{msg}_s)$ to $P_s \in \mathcal{P}$.

Fig. 20: Mixed-Secure Ideal functionality for $[\![\cdot]\!]$-Sh

Simulator $\mathcal{S}$

Preprocessing:
– Let P_i^* be the malicious party and P_j^* be the semi-honest party.
– $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{setup}}$ so that $\mathcal{S}$ and P_i^* get $\{\langle\alpha_{\mathsf{v}}\rangle_{ik}\}_{k\neq i}$ and $\mathcal{S}$ and P_j^* get $\{\langle\alpha_{\mathsf{v}}\rangle_{jk}\}_{k\neq i,j}$.
– If P_i^* or P_j^* is the client, then the adversary $(\{P_i^*, P_j^*\})$ and $\mathcal{S}$ pick $\langle\alpha_{\mathsf{v}}\rangle_{jk}$ $1 \leq j < k \leq 4$.

Online:
– If P_i^* is the client, $\mathcal{S}$ receives either β_{v} from P_i^* or outputs $\mathtt{DP} = \{P_i^*, P_k\}$, for some $k \neq i$.

– If P_j^* is the client, $\mathcal{S}$ receives either β_{v} from P_j^* or outputs $\mathtt{DP} = \{P_i^*, P_k\}$, for some $k \neq i$.
– If P_i^* or P_j^* is not the client. $\mathcal{S}$ invokes $\mathcal{F}_{\mathsf{Sh}}^{\mathsf{mixed}}$ and receives β_{v}. It sends β_{v} to P_i^*, P_j^*.

Fig. 21: Simulator $\mathcal{S}$ for $[\![\cdot]\!]$-Sh

The view generated by the simulator in Fig. 21 is indistinguishable from the real view and the argument is similar to Lemma 6.

$\langle\cdot\rangle$-Rec *Functionality:* The ideal functionality for $\langle\cdot\rangle$-Rec (Fig. 3) appears in Fig. 22.

Functionality $\mathcal{F}_{\langle\cdot\rangle\text{-Rec}}$

$\mathcal{F}_{\langle\cdot\rangle\text{-Rec}}$ interacts with the parties in $\mathcal{P}$ and the adversaries $\mathcal{S}_{\mathcal{A}}$ and $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$.
– $\mathcal{F}_{\langle\cdot\rangle\text{-Rec}}$ receives $(\mathsf{Input}, \langle \mathsf{v}\rangle_s)$ from each $P_s \in \mathcal{P}$. Let P^* be the party corrupted by $\mathcal{S}_{\mathcal{A}}$.
– $\mathcal{F}_{\langle\cdot\rangle\text{-Rec}}$ sends $\mathsf{v} = \sum_{(i,j)} \langle \mathsf{v}\rangle_{ij}$ to $\mathcal{S}_{\mathcal{A}}$, $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ and receives `continue` or `abort` with $(\mathsf{Select}, \mathcal{D})$ from $\mathcal{S}_{\mathcal{A}}$. Here, $\mathcal{D}$ denotes the pair of parties that $\mathcal{S}_{\mathcal{A}}$ wants to choose as dispute pair.
– If $\mathcal{F}_{\langle\cdot\rangle\text{-Rec}}$ receives `continue`, set $\mathsf{msg}_s = \mathsf{v}$, for each $P_s \in \mathcal{P}$.
– Else if $\mathcal{F}_{\langle\cdot\rangle\text{-Rec}}$ receives `abort`, then:
– If $P^* \in \mathcal{D}$, then set $\mathtt{DP} = \mathcal{D}$ and $\mathsf{msg}_s = \mathtt{DP}$ for each $P_s \in \mathcal{P}$.
– Else set $\mathtt{DP} = \mathcal{P}\backslash\mathcal{D}$ and $\mathsf{msg}_s = \mathtt{DP}$ for each $P_s \in \mathcal{P}$.
– $\mathcal{S}_{\mathcal{A}}$ sends it's view to $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$.

Output: Send $(\mathsf{Output}, \mathsf{msg}_s)$ to $P_s \in \mathcal{P}$.

Fig. 22: Ideal functionality for $\langle\cdot\rangle$-Rec

$\langle\cdot\rangle$-Rec *Simulator:* Simulators for $\langle\cdot\rangle$-Rec (Fig. 3) are provided in Fig. 23, Fig. 24, Fig. 25, Fig. 26.

Simulator $\mathcal{S}_{\mathcal{A}}{}^{P_1}$, $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$

Let P_1 be the maliciously corrupted party.
– $\mathcal{S}_{\mathcal{A}}$ executes $\mathcal{S}_{\mathcal{A}\,\mathsf{jmp3}}^{s}$ for $\langle \mathsf{v}\rangle_{12}$, where $s = 1$.
– $\mathcal{S}_{\mathcal{A}}$ executes $\mathcal{S}_{\mathcal{A}\,\mathsf{jmp3}}^{s}$ for $\langle \mathsf{v}\rangle_{14}$, where $s = 1$.
– $\mathcal{S}_{\mathcal{A}}$ executes $\mathcal{S}_{\mathcal{A}\,\mathsf{jmp3}}^{s}$ for $\langle \mathsf{v}\rangle_{13}$, where $s = 1$.
– $\mathcal{S}_{\mathcal{A}}$ calls $\mathcal{F}_{\langle\cdot\rangle\text{-Rec}}$ with inputs $\langle \mathsf{v}\rangle_{12}, \langle \mathsf{v}\rangle_{13}, \langle \mathsf{v}\rangle_{14}$, and gets the reconstructed value v.
– $\mathcal{S}_{\mathcal{A}}$ executes $\mathcal{S}_{\mathcal{A}\,\mathsf{jmp4}}^{r}$ for v, where $r = 1$.
  – Subcase 1: Let P_2 be semi-honest.
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}(\langle \mathsf{v}\rangle_{12}, \langle \mathsf{v}\rangle_{13}, \langle \mathsf{v}\rangle_{14}, \langle \mathsf{v}\rangle_{23}, \langle \mathsf{v}\rangle_{24}, \mathsf{v})$
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ executes $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}\,\mathsf{jmp4}}^{r,r'}$ for v, where $r = 1, r' = 2$.
  – Subcase 2: Let P_3 be semi-honest.

$\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}(\langle \mathsf{v} \rangle_{12}, \langle \mathsf{v} \rangle_{13}, \langle \mathsf{v} \rangle_{14}, \langle \mathsf{v} \rangle_{23}, \langle \mathsf{v} \rangle_{34}, \mathsf{v})$
$\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ executes $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}{}^{s,r}_{\mathsf{jmp3}}$ for $\langle \mathsf{v} \rangle_{12}$, where $s = 1, r = 3$.
$\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ executes $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}{}^{s,r}_{\mathsf{jmp3}}$ for $\langle \mathsf{v} \rangle_{14}$, where $s = 1, r = 3$.
$\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ computes $\langle \mathsf{v} \rangle_{24} = \mathsf{v} - \sum_{(i,j) \neq (2,4)} \langle \mathsf{v} \rangle_{ij}$.
$\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ executes $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}{}^{r}_{\mathsf{jmp3}}$ for $\langle \mathsf{v} \rangle_{24}$, where $r = 3$.

- Subcase 3: Let P_4 be semi-honest.
  $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}(\langle \mathsf{v} \rangle_{12}, \langle \mathsf{v} \rangle_{13}, \langle \mathsf{v} \rangle_{14}, \langle \mathsf{v} \rangle_{24}, \langle \mathsf{v} \rangle_{34}, \mathsf{v})$
  $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ computes $\langle \mathsf{v} \rangle_{23} = \mathsf{v} - \sum_{(i,j) \neq (2,3)} \langle \mathsf{v} \rangle_{ij}$.
  $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ executes $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}{}^{r}_{\mathsf{jmp3}}$ for $\langle \mathsf{v} \rangle_{12} + \langle \mathsf{v} \rangle_{23}$, where $r = 4$.
  $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ executes $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}{}^{s,r}_{\mathsf{jmp3}}$ for $\langle \mathsf{v} \rangle_{13}$, where $s = 1, r = 4$.

Fig. 23: Simulator ${\mathcal{S}_{\mathcal{A}}}^{P_1}$ for $\langle \cdot \rangle$-Rec

Simulator ${\mathcal{S}_{\mathcal{A}}}^{P_2}$, $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$

Let P_2 be the maliciously corrupted party.
- $\mathcal{S}_{\mathcal{A}}$ executes $\mathcal{S}_{\mathcal{A}}{}^{s}_{\mathsf{jmp3}}$ for $\langle \mathsf{v} \rangle_{12}$, where $s = 2$.
- $\mathcal{S}_{\mathcal{A}}$ executes $\mathcal{S}_{\mathcal{A}}{}^{s}_{\mathsf{jmp3}}$ for $\langle \mathsf{v} \rangle_{12} + \langle \mathsf{v} \rangle_{23}$, where $s = 2$.
- $\mathcal{S}_{\mathcal{A}}$ executes $\mathcal{S}_{\mathcal{A}}{}^{s}_{\mathsf{jmp3}}$ for $\langle \mathsf{v} \rangle_{24}$, where $s = 2$.
- $\mathcal{S}_{\mathcal{A}}$ calls $\mathcal{F}_{\langle \cdot \rangle\text{-Rec}}$ with inputs $\langle \mathsf{v} \rangle_{12}, \langle \mathsf{v} \rangle_{23}, \langle \mathsf{v} \rangle_{24}$, and gets the reconstructed value v.
- $\mathcal{S}_{\mathcal{A}}$ executes $\mathcal{S}_{\mathcal{A}}{}^{r}_{\mathsf{jmp4}}$ for v, where $r = 2$.
  - Subcase 1: Let P_1 be semi-honest.
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}(\langle \mathsf{v} \rangle_{12}, \langle \mathsf{v} \rangle_{13}, \langle \mathsf{v} \rangle_{14}, \langle \mathsf{v} \rangle_{23}, \langle \mathsf{v} \rangle_{24}, \mathsf{v})$
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ executes $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}{}^{r,r'}_{\mathsf{jmp4}}$ for v, where $r = 2, r' = 1$.
  - Subcase 2: Let P_3 be semi-honest.
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}(\langle \mathsf{v} \rangle_{12}, \langle \mathsf{v} \rangle_{13}, \langle \mathsf{v} \rangle_{23}, \langle \mathsf{v} \rangle_{24}, \langle \mathsf{v} \rangle_{34}, \mathsf{v})$
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ executes $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}{}^{s,r}_{\mathsf{jmp3}}$ for $\langle \mathsf{v} \rangle_{12}$, where $s = 2, r = 3$.
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ executes $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}{}^{s,r}_{\mathsf{jmp3}}$ for $\langle \mathsf{v} \rangle_{24}$, where $s = 2, r = 3$.
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ computes $\langle \mathsf{v} \rangle_{14} = \mathsf{v} - \sum_{(i,j) \neq (1,4)} \langle \mathsf{v} \rangle_{ij}$.
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ executes $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}{}^{r}_{\mathsf{jmp3}}$ for $\langle \mathsf{v} \rangle_{14}$, where $r = 3$.
  - Subcase 3: Let P_4 be semi-honest.
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}(\langle \mathsf{v} \rangle_{12}, \langle \mathsf{v} \rangle_{14}, \langle \mathsf{v} \rangle_{23}, \langle \mathsf{v} \rangle_{24}, \langle \mathsf{v} \rangle_{34}, \mathsf{v})$
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ computes $\langle \mathsf{v} \rangle_{13} = \mathsf{v} - \sum_{(i,j) \neq (1,3)} \langle \mathsf{v} \rangle_{ij}$.
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ executes $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}{}^{r}_{\mathsf{jmp3}}$ for $\langle \mathsf{v} \rangle_{13}$, where $r = 4$.
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ executes $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}{}^{s,r}_{\mathsf{jmp3}}$ for $\langle \mathsf{v} \rangle_{12} + \langle \mathsf{v} \rangle_{23}$, where $s = 2, r = 4$.

Fig. 24: Simulator ${\mathcal{S}_{\mathcal{A}}}^{P_2}$ for $\langle \cdot \rangle$-Rec

Simulator ${\mathcal{S}_{\mathcal{A}}}^{P_3}$, $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$

Let P_3 be the maliciously corrupted party.
- $\mathcal{S}_{\mathcal{A}}$ picks $\langle \mathsf{v} \rangle_{12}$ and executes $\mathcal{S}_{\mathcal{A}}{}^{r}_{\mathsf{jmp3}}$ for $\langle \mathsf{v} \rangle_{12}$, where $r = 3$.
- $\mathcal{S}_{\mathcal{A}}$ picks $\langle \mathsf{v} \rangle_{14}$ and executes $\mathcal{S}_{\mathcal{A}}{}^{r}_{\mathsf{jmp3}}$ for $\langle \mathsf{v} \rangle_{14}$, where $r = 3$.
- $\mathcal{S}_{\mathcal{A}}$ calls $\mathcal{F}_{\langle \cdot \rangle\text{-Rec}}$ with inputs $\langle \mathsf{v} \rangle_{13}, \langle \mathsf{v} \rangle_{23}, \langle \mathsf{v} \rangle_{34}$ and gets the reconstructed value v.
- $\mathcal{S}_{\mathcal{A}}$ sets $\langle \mathsf{v} \rangle_{24} = \mathsf{v} - \sum_{(i,j) \neq (2,4)} \langle \mathsf{v} \rangle_{ij}$ and executes $\mathcal{S}_{\mathcal{A}}{}^{r}_{\mathsf{jmp3}}$ for $\langle \mathsf{v} \rangle_{24}$, where $r = 3$.

– $\mathcal{S}_{\mathcal{A}}$ executes $\mathcal{S}^{s}_{\mathcal{A}\mathsf{jmp3}}$ for $\langle \mathsf{v} \rangle_{13}$, where $s = 3$.
– $\mathcal{S}_{\mathcal{A}}$ executes $\mathcal{S}^{s}_{\mathcal{A}\mathsf{jmp3}}$ for $\langle \mathsf{v} \rangle_{12} + \langle \mathsf{v} \rangle_{23}$, where $s = 3$.
– $\mathcal{S}_{\mathcal{A}}$ executes $\mathcal{S}^{s}_{\mathcal{A}\mathsf{jmp4}}$ for v, where $s = 3$.
  – Subcase 1: Let P_1 be semi-honest party.
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}(\langle \mathsf{v} \rangle_{12}, \langle \mathsf{v} \rangle_{13}, \langle \mathsf{v} \rangle_{14}, \langle \mathsf{v} \rangle_{23}, \langle \mathsf{v} \rangle_{34}, \mathsf{v})$
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ executes $\mathcal{S}^{s,r}_{\mathcal{A}_{\mathcal{H}}\mathsf{jmp4}}$ for v, where $s = 3, r = 1$.
  – Subcase 2: Let P_2 be semi-honest party.
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}(\langle \mathsf{v} \rangle_{12}, \langle \mathsf{v} \rangle_{13}, \langle \mathsf{v} \rangle_{23}, \langle \mathsf{v} \rangle_{24}, \langle \mathsf{v} \rangle_{34}, \mathsf{v})$
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ executes $\mathcal{S}^{s,r}_{\mathcal{A}_{\mathcal{H}}\mathsf{jmp4}}$ for v, where $s = 3, r = 2$.
  – Subcase 3: Let P_4 be semi-honest party.
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}(\langle \mathsf{v} \rangle_{13}, \langle \mathsf{v} \rangle_{14}, \langle \mathsf{v} \rangle_{23}, \langle \mathsf{v} \rangle_{24}, \langle \mathsf{v} \rangle_{34}, \mathsf{v})$
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ computes $\langle \mathsf{v} \rangle_{12} = \mathsf{v} - \sum_{(i,j) \neq (1,2)} \langle \mathsf{v} \rangle_{ij}$.
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ executes $\mathcal{S}^{r}_{\mathcal{A}_{\mathcal{H}}\mathsf{jmp3}}$ for $\langle \mathsf{v} \rangle_{13}$, where $s = 3, r = 4$.
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ executes $\mathcal{S}^{s,r}_{\mathcal{A}_{\mathcal{H}}\mathsf{jmp3}}$ for $\langle \mathsf{v} \rangle_{12} + \langle \mathsf{v} \rangle_{23}$, where $s = 3, r = 4$.

Fig. 25: Simulator $\mathcal{S}_{\mathcal{A}}{}^{P_3}$ for $\langle \cdot \rangle$-Rec

Simulator $\mathcal{S}_{\mathcal{A}}{}^{P_4}$, $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$

Let P_4 be the maliciously corrupt party.
– $\mathcal{S}_{\mathcal{A}}$ picks $\langle \mathsf{v} \rangle_{12}, \langle \mathsf{v} \rangle_{23}$ and executes $\mathcal{S}^{r}_{\mathcal{A}\mathsf{jmp3}}$ for $\langle \mathsf{v} \rangle_{12} + \langle \mathsf{v} \rangle_{23}$, where $r = 4$.
– $\mathcal{S}_{\mathcal{A}}$ calls $\mathcal{F}_{\langle \cdot \rangle\text{-}\mathsf{Rec}}$ with inputs $\langle \mathsf{v} \rangle_{14}, \langle \mathsf{v} \rangle_{24}, \langle \mathsf{v} \rangle_{34}$ and gets the reconstructed value v.
– $\mathcal{S}_{\mathcal{A}}$ sets $\langle \mathsf{v} \rangle_{13} = \mathsf{v} - \sum_{(i,j) \neq (1,3)} \langle \mathsf{v} \rangle_{ij}$ and executes $\mathcal{S}^{r}_{\mathcal{A}\mathsf{jmp3}}$ for $\langle \mathsf{v} \rangle_{13}$, where $r = 4$.
– $\mathcal{S}_{\mathcal{A}}$ executes $\mathcal{S}^{s}_{\mathcal{A}\mathsf{jmp3}}$ for $\langle \mathsf{v} \rangle_{14}$, where $s = 4$.
– $\mathcal{S}_{\mathcal{A}}$ executes $\mathcal{S}^{s}_{\mathcal{A}\mathsf{jmp3}}$ for $\langle \mathsf{v} \rangle_{24}$, where $s = 4$.
– $\mathcal{S}_{\mathcal{A}}$ executes $\mathcal{S}^{s}_{\mathcal{A}\mathsf{jmp4}}$ for v, where $s = 4$.
  – Subcase 1: Let P_1 be semi-honest party.
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}(\langle \mathsf{v} \rangle_{12}, \langle \mathsf{v} \rangle_{13}, \langle \mathsf{v} \rangle_{14}, \langle \mathsf{v} \rangle_{24}, \langle \mathsf{v} \rangle_{34}, \mathsf{v})$
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ executes $\mathcal{S}^{s,r}_{\mathcal{A}_{\mathcal{H}}\mathsf{jmp4}}$ for v, where $s = 4, r = 1$.
  – Subcase 2: Let P_2 be semi-honest party.
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}(\langle \mathsf{v} \rangle_{12}, \langle \mathsf{v} \rangle_{23}, \langle \mathsf{v} \rangle_{24}, \langle \mathsf{v} \rangle_{14}, \langle \mathsf{v} \rangle_{34}, \mathsf{v})$
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ executes $\mathcal{S}^{s,r}_{\mathcal{A}_{\mathcal{H}}\mathsf{jmp4}}$ for v, where $s = 4, r = 2$.
  – Subcase 3: Let P_3 be semi-honest party.
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}(\langle \mathsf{v} \rangle_{13}, \langle \mathsf{v} \rangle_{14}, \langle \mathsf{v} \rangle_{23}, \langle \mathsf{v} \rangle_{24}, \langle \mathsf{v} \rangle_{34}, \mathsf{v})$
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ computes $\langle \mathsf{v} \rangle_{12} = \mathsf{v} - \sum_{(i,j) \neq (1,2)} \langle \mathsf{v} \rangle_{ij}$.
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ executes $\mathcal{S}^{r}_{\mathcal{A}_{\mathcal{H}}\mathsf{jmp3}}$ for $\langle \mathsf{v} \rangle_{12}$, where $r = 3$.
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ executes $\mathcal{S}^{s,r}_{\mathcal{A}_{\mathcal{H}}\mathsf{jmp3}}$ for $\langle \mathsf{v} \rangle_{14}$, where $s = 4, r = 3$.
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ executes $\mathcal{S}^{s,r}_{\mathcal{A}_{\mathcal{H}}\mathsf{jmp3}}$ for $\langle \mathsf{v} \rangle_{24}$, where $s = 4, r = 3$.

Fig. 26: Simulator $\mathcal{S}_{\mathcal{A}}{}^{P_4}$ for $\langle \cdot \rangle$-Rec

Lemma 7 (Security). *The protocol $\langle \cdot \rangle$-Rec, described in Fig. 3, realizes $\mathcal{F}_{\langle \cdot \rangle\text{-}\mathsf{Rec}}$ (Fig. 22) with computational security in the $(\mathcal{F}_{\mathsf{setup}}, \mathcal{F}_{\mathsf{jmp3}}, \mathcal{F}_{\mathsf{jmp4}})$-hybrid model against $(1,1)$-FaF adversaries $\mathcal{A}$, $\mathcal{A}_{\mathcal{H}}$, controlling P_i, P_j respectively.*

Proof. Case I: If $i = 1$, ${\mathcal{S}_{\mathcal{A}}}^{P_1}$ generates a transcript $\tau = \{\langle \mathsf{v} \rangle_{12}, \langle \mathsf{v} \rangle_{14}, \langle \mathsf{v} \rangle_{13}, \mathsf{v}\}$, which is computationally indistinguishable from P_1's view.

Case II: If $i = 2$, ${\mathcal{S}_{\mathcal{A}}}^{P_2}$ generates a transcript $\tau = \{\langle \mathsf{v} \rangle_{12}, \langle \mathsf{v} \rangle_{23}, \langle \mathsf{v} \rangle_{24}, \mathsf{v}\}$, which is computationally indistinguishable from P_2's view.

Case III: If $i = 3$, ${\mathcal{S}_{\mathcal{A}}}^{P_3}$ generates a transcript $\tau = \{\langle \mathsf{v} \rangle_{12}, \langle \mathsf{v} \rangle_{13}, \langle \mathsf{v} \rangle_{14}, \langle \mathsf{v} \rangle_{23}, \langle \mathsf{v} \rangle_{24}, \langle \mathsf{v} \rangle_{34}, \mathsf{v}\}$, where $\langle \mathsf{v} \rangle_{12}, \langle \mathsf{v} \rangle_{14}$ are randomly picked. Therefore, the transcript is indistinguishable from P_3's view in the protocol.

Case IV: If $i = 4$, ${\mathcal{S}_{\mathcal{A}}}^{P_4}$ generates a transcript $\tau = \{\langle \mathsf{v} \rangle_{13}, \langle \mathsf{v} \rangle_{14}, \langle \mathsf{v} \rangle_{12} + \langle \mathsf{v} \rangle_{23}, \langle \mathsf{v} \rangle_{24}, \langle \mathsf{v} \rangle_{34}, \mathsf{v}\}$, where $\langle \mathsf{v} \rangle_{12}, \langle \mathsf{v} \rangle_{23}$ are randomly picked. Therefore, the transcript is indistinguishable from P_4's view in the protocol.

Suppose P_i is the malicious party and P_j is the semi-honest party. The simulator, $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$, starts with P_j's inputs and output. Additionally, it receives P_i's inputs and v from ${\mathcal{S}_{\mathcal{A}}}^{P_i}$. The simulated view is thus indistinguishable from the view of P_j which consists of $\{\{\langle \mathsf{v} \rangle_{st}\}_{s,t}, \mathsf{v}\}$, where $(s, t) \neq (k, m)$ and P_k, P_m are the honest parties.

C.2 Multiplication Protocols

In this section, we provide the ideal functionalities and the simulation proofs corresponding to the multiplication protocols from §5.

Distributed Multiplication Functionality: The ideal functionality for disMult (Fig. 5) appears in Fig. 27.

Functionality $\mathcal{F}_{\mathsf{disMult}}$

$\mathcal{F}_{\mathsf{disMult}}$ interacts with the parties in $\mathcal{P}$ and the adversaries $\mathcal{S}_{\mathcal{A}}$, $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$.

- $\mathcal{F}_{\mathsf{disMult}}$ receives a from P_i, P_j and b, c^2 from P_k, P_m. Let P^* be the party controlled by $\mathcal{S}_{\mathcal{A}}$.
- If $P^* \in \{P_i, P_j\}$ $\mathcal{F}_{\mathsf{disMult}}$ gives $\mathcal{S}_{\mathcal{A}}$ a, otherwise b, c^2.
- $\mathcal{S}_{\mathcal{A}}$ gives input to $\mathcal{F}_{\mathsf{disMult}}$ on behalf of P^*.
- If $\mathcal{F}_{\mathsf{disMult}}$ receives `abort` then set $\mathtt{DP} = \mathcal{D}$ if $P^* \in \mathcal{D}$, else $\mathtt{DP} = \mathcal{P} \backslash \mathcal{D}$. And set $\mathsf{msg}_s = \mathtt{DP}$ $\forall P_s \in \mathcal{P}$.
- Else, $\mathcal{F}_{\mathsf{disMult}}$ sets $\mathsf{msg}_i = \mathsf{msg}_j = c^1$ and $\mathsf{msg}_k = \mathsf{msg}_m = c^2$ such that $c^1 + c^2 = ab$.
- $\mathcal{F}_{\mathsf{disMult}}$ sends msg of P^* to $\mathcal{S}_{\mathcal{A}}$, receives the command `continue` or `abort` with $(\mathsf{Select}, \mathcal{D})$. Here, $\mathcal{D}$ denotes the pair of parties that $\mathcal{S}_{\mathcal{A}}$ wants to choose as dispute pair.
- If $\mathcal{F}_{\mathsf{disMult}}$ receives `abort` command with $\mathcal{D}$ from $\mathcal{S}_{\mathcal{A}}$, then $\mathcal{F}_{\mathsf{disMult}}$ outputs a dispute pair DP to all the parties, where $\mathtt{DP} = \mathcal{D}$ if $P^* \in \mathcal{D}$, else $\mathtt{DP} = \mathcal{P} \backslash \mathcal{D}$. And sets $\mathsf{msg}_s = \mathtt{DP}$ $\forall P_s \in \mathcal{P}$. Else continue.
- $\mathcal{S}_{\mathcal{A}}$ sends it's view to $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$.

Output: Send $(\mathsf{Output}, \mathsf{msg}_s)$ $\forall P_s \in \mathcal{P}$.

Fig. 27: Ideal functionality for disMult

Distributed Multiplication Simulator: Simulators for the distributed multiplication protocol disMult (Fig. 5) are provided in Fig. 28 and Fig. 29. Here, we provide the simulation for the case of malicious P_i and P_k. The simulations for malicious P_j and P_m are analogous.

Simulator ${\mathcal{S}_{\mathcal{A}}}^{P_i}$, $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$

Let P_i be the maliciously corrupted party.

- $\mathcal{S}_{\mathcal{A}}$ with input a invokes $\mathcal{F}_{\mathsf{disMult}}$ and receives c^1. It randomly picks b and computes $c^2 = ab - c^1$.
- $\mathcal{S}_{\mathcal{A}}$ executes $\mathcal{S}_{\mathsf{OPE}}$ with P_j's input a and P_j, P_k's input b, c^2.
- For every communication from the receiver to the sender, $\mathcal{S}_{\mathcal{A}}$ invokes $\mathcal{S}_{\mathcal{A}\mathsf{jmp4}}^{s}$ with $s = i$.
- For every communication from the sender to the receiver, $\mathcal{S}_{\mathcal{A}}$ invokes $\mathcal{S}_{\mathcal{A}\mathsf{jmp4}}^{r}$ with $r = i$.
  - Subcase 1: Let P_j be semi-honest.
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}(a, b, c^2)$
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ executes $\mathcal{S}_{\mathsf{OPE}}$ and sends c^1 to P_j.
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ executes $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}\mathsf{jmp4}}^{r,r'}$ and $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}\mathsf{jmp4}}^{s,s'}$ for every communication accordingly.
  - Subcase 2: Let P_k be semi-honest.
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}(a, b, c^2)$
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ discards b, c^2.
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ computes $c^1 = ab - c^2$.
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ executes $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}\mathsf{jmp4}}^{r,s'}$ and $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}\mathsf{jmp4}}^{s,r'}$ for every communication accordingly.

Fig. 28: Simulator ${\mathcal{S}_{\mathcal{A}}}^{P_i}$ for disMult

Simulator ${\mathcal{S}_{\mathcal{A}}}^{P_k}$, $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$

Let P_k be the maliciously corrupted party.

- $\mathcal{S}_{\mathcal{A}}$ invokes $\mathcal{F}_{\mathsf{disMult}}$ with b, c^2 and randomly picks a.
- $\mathcal{S}_{\mathcal{A}}$ executes $\mathcal{S}_{\mathsf{OPE}}$ with P_i, P_j's input a and P_m's input b, c^2.
- For every communication from the receiver to the sender, $\mathcal{S}_{\mathcal{A}}$ invokes $\mathcal{S}_{\mathcal{A}\mathsf{jmp4}}^{r}$ with $r = k$.
- For every communication from the sender to the receiver, $\mathcal{S}_{\mathcal{A}}$ invokes $\mathcal{S}_{\mathcal{A}\mathsf{jmp4}}^{s}$ with $s = k$.
  - Subcase 1: Let P_i be semi-honest.
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}(a, b, c^2)$
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ discards a.
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ computes $c^1 = ab - c^2$.
    $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ executes $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}\mathsf{jmp4}}^{r,s'}$ and $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}\mathsf{jmp4}}^{s,r'}$ for every communication accordingly.
  - Subcase 2: Let P_m be semi-honest.

$\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}(a, b, c^2)$
$\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ computes $c^1 = ab - c^2$.
$\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ executes ${\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}}_{\mathsf{jmp4}}^{r,r'}$ and ${\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}}_{\mathsf{jmp4}}^{s,s'}$ for every communication accordingly.

Fig. 29: Simulator ${\mathcal{S}_{\mathcal{A}}}^{P_k}$ for disMult

Lemma 8 (Security). *The protocol* disMult*, described in Fig. 5, realizes* $\mathcal{F}_{\mathsf{disMult}}$ *(Fig. 27) with computational security in the* $(\mathcal{F}_{\mathsf{setup}}, \mathcal{F}_{\mathsf{jmp4}})$*-hybrid model against* $(1,1)$-FaF *adversaries* $\mathcal{A}$*,* $\mathcal{A}_{\mathcal{H}}$*, controlling one one party each when* disMult *is instantiated with a secure* OPE *protocol.*

Proof. This follows directly from the security of OPE and jmp4 invocations.

Case I: If one of the receivers P_i or P_j is malicious. By the security of semi-honest OPE, $\exists$ a simulator $\mathcal{S}^r_{\mathsf{OPE}}$, for a corrupt receiver such that with input a, c^1, it generates a view that is indistinguishable from the real view of the corrupt receiver. Let $\mathcal{S}^r$ be the simulator for disMult for a corrupt receiver. $\mathcal{S}^r$ invokes $\mathcal{F}_{\mathsf{disMult}}$ with input a. If it receives DP, it outputs DP and terminates. Else, if it receives c^1, then it runs $\mathcal{S}^r_{\mathsf{OPE}}$ and replaces the communications by $\mathcal{F}_{\mathsf{jmp4}}$ invocations. For this $\mathcal{S}^r_{\mathsf{OPE}}$ executes $\mathcal{S}_{\mathcal{A}\mathsf{jmp4}}$. The view generated by $\mathcal{S}^r$ is indistinguishable from a corrupt receiver's view. If not, then it implies that the view generated by $\mathcal{S}^r_{\mathsf{OPE}}$ is not indistinguishable, which contradicts the security of OPE.
Subcase I: If the other receiver is semi-honest, the simulation proceeds similar to the simulation of a malicious receiver.
Subcase II: If one of the senders is semi-honest, since the simulator knows all the inputs to the $\mathcal{F}_{\mathsf{disMult}}$, the simulation is trivial.

Case II: If one of the senders P_k or P_m is malicious. By the security of semi-honest OPE, $\exists$ a simulator $\mathcal{S}^s_{\mathsf{OPE}}$, for a corrupt sender such that with input b, c^2, it generates a view that is indistinguishable from the real view of the corrupt sender. Let $\mathcal{S}^s$ be the simulator for disMult for a corrupt sender. $\mathcal{S}^r$ invokes $\mathcal{F}_{\mathsf{disMult}}$ with input b, c^2. If it receives DP, it outputs DP and terminates. Else, it runs $\mathcal{S}^s_{\mathsf{OPE}}$ and replaces the communications by $\mathcal{F}_{\mathsf{jmp4}}$ invocations. For this $\mathcal{S}^s_{\mathsf{OPE}}$ executes $\mathcal{S}_{\mathcal{A}\mathsf{jmp4}}$. The view generated by $\mathcal{S}^s$ is indistinguishable from a corrupt sender's view. If not, then it implies that the view generated by $\mathcal{S}^s_{\mathsf{OPE}}$ is not indistinguishable, which contradicts the security of OPE.
Subcase I: If one of the receivers is semi-honest, since the simulator knows all the inputs to the $\mathcal{F}_{\mathsf{disMult}}$, the simulation is trivial.
Subcase II: If the other sender is semi-honest, the simulation proceeds similar to the simulation of a malicious sender.

Multiplication Functionality: The ideal functionality for mult (Fig. 4) appears in Fig. 30.

Functionality $\mathcal{F}_{\mathsf{mult}}$

$\mathcal{F}_{\mathsf{mult}}$ interacts with the parties in $\mathcal{P}$ and the adversaries $\mathcal{S}_{\mathcal{A}}$ and $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$.
- $\mathcal{F}_{\mathsf{mult}}$ receives $(\mathsf{Input}, [\![\mathsf{x}]\!]_s, [\![\mathsf{y}]\!]_s)$ from $P_s \in \mathcal{P}$.
- Let P^* be the malicious party controlled by $\mathcal{S}_{\mathcal{A}}$.
- $\mathcal{F}_{\mathsf{mult}}$ randomly picks $\langle \alpha_{\mathsf{z}} \rangle_{ij} \in \mathbb{Z}_{2^\ell}$, for $1 \le i < j \le 4$ and computes $\alpha_{\mathsf{z}} = \sum_{(i,j)} \langle \alpha_{\mathsf{z}} \rangle_{ij}$.
- $\mathcal{F}_{\mathsf{mult}}$ computes $[\![\mathsf{z}]\!]_s = (\beta_{\mathsf{z}}, \langle \alpha_{\mathsf{z}} \rangle_s)$ where $\beta_{\mathsf{z}} = \mathsf{x} \cdot \mathsf{y} + \alpha_{\mathsf{z}}$, for each $P_s \in \mathcal{P}$.
- $\mathcal{F}_{\mathsf{mult}}$ sends $[\![\mathsf{z}]\!]_s$ to $\mathcal{S}_{\mathcal{A}}$, $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$, and receives `continue` or `abort` with $(\mathsf{Select}, \mathcal{D})$ from $\mathcal{S}_{\mathcal{A}}$. Here, $\mathcal{D}$ denotes the pair of parties that $\mathcal{S}_{\mathcal{A}}$ wants to choose as dispute pair.
- If $\mathcal{F}_{\mathsf{mult}}$ receives `continue`, set $\mathsf{msg}_s = [\![\mathsf{z}]\!]_s$, for each $P_s \in \mathcal{P}$.
- Else if $\mathcal{F}_{\mathsf{mult}}$ receives `abort`, then:
- If $P^* \in \mathcal{D}$, then set $\mathtt{DP} = \mathcal{D}$ and $\mathsf{msg}_s = \mathtt{DP}$ for each $P_s \in \mathcal{P}$.
- Else set $\mathtt{DP} = \mathcal{P} \setminus \mathcal{D}$ and $\mathsf{msg}_s = \mathtt{DP}$ for each $P_s \in \mathcal{P}$.

Output: Send $(\mathsf{Output}, \mathsf{msg}_s)$ to $P_s \in \mathcal{P}$.

Fig. 30: Ideal functionality for evaluating a multiplication gate

TripGen Simulator: The simulator for tripGen(Fig. 6) appears in Fig. 31.

Simulator $\mathcal{S}_{\mathcal{A}\mathsf{tripGen}}, \mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$

<u>**Malicious Simulation**</u> Let P_i be the maliciously corrupt party, and let $P_j = P_{i+1},\ P_m = P_{i+2},\ P_k = P_{i+3}$.
- $\mathcal{S}_{\mathcal{A}}$ has $\langle \alpha_{\mathsf{x}} \rangle_{is}, \langle \alpha_{\mathsf{y}} \rangle_{is}\ \forall s \in \{j, k, m\}$.
- Simulation for S_2 terms: $\mathcal{S}_{\mathcal{A}}$ initializes $\langle \tau_{u,v} \rangle_{is} = \langle \alpha_{\mathsf{x}} \rangle_{is} \langle \alpha_{\mathsf{y}} \rangle_{is}$ if $(u, v) = (i, s)$ and $\langle \tau_{u,v} \rangle_{is} = 0$, otherwise.
- Simulation for S_1 terms:
  1. $\mathcal{S}_{\mathcal{A}}$ receives δ_i^1 from P_i on behalf of P_k. $\mathcal{S}_{\mathcal{A}}$ executes the simulator $\mathcal{S}_{\mathcal{A}\mathsf{disZK}}$ for the malicious prover case. $\mathcal{S}_{\mathcal{A}}$ sets $\langle \delta_i \rangle_{ik} = \delta_i^1$ and $\langle \delta_i \rangle_{is} = 0$, for all $s \in \{j, m\}$.
  2. $\mathcal{S}_{\mathcal{A}}$ sends on behalf of P_j δ_j^1 to P_i, $\mathcal{S}_{\mathcal{A}}$ executes the simulator $\mathcal{S}_{\mathcal{A}\mathsf{disZK}}$ where P_j is the prover and P_i as the corrupted verifier. $\mathcal{S}_{\mathcal{A}}$ sets $\langle \delta_j \rangle_{ij} = \delta_j^1$ and $\langle \delta_j \rangle_{is} = 0$, for all $s \in \{k, m\}$.
  3. $\mathcal{S}_{\mathcal{A}}$ and P_i pick random δ_m^2, $\mathcal{S}_{\mathcal{A}}$ executes the simulator $\mathcal{S}_{\mathcal{A}\mathsf{disZK}}$ where P_m is the prover and P_i is the corrupted verifier. $\mathcal{S}_{\mathcal{A}}$ sets $\langle \delta_m \rangle_{im} = \delta_m^2$ and $\langle \delta_m \rangle_{is} = 0$, for all $s \in \{j, k\}$.
  4. $\mathcal{S}_{\mathcal{A}}$ and P_i pick random δ_k^2, $\mathcal{S}_{\mathcal{A}}$ executes the simulator $\mathcal{S}_{\mathcal{A}\mathsf{disZK}}$ where P_k is the prover and P_i is the corrupted verifier. $\mathcal{S}_{\mathcal{A}}$ sets $\langle \delta_k \rangle_{ik} = \delta_k^2$ and $\langle \delta_k \rangle_{is} = 0$, for all $s \in \{j, m\}$.
  5. $\mathcal{S}_{\mathcal{A}}$ continues if `DP` is not the output
- Simulation for S_0 terms:
  1. $\mathcal{S}_{\mathcal{A}}$ executes $\mathcal{S}_{\mathcal{A}\mathsf{disMult}}$ for all the six terms of the summands S_0 for computing the terms of the $\langle \alpha_x \rangle_{uv} \cdot \langle \alpha_y \rangle_{pq}$, with P_i as malicious party.
  2. $\mathcal{S}_{\mathcal{A}}$ sets $\langle \gamma_{uv,pq} \rangle_{is}$ according to the protocol, if $\mathcal{S}_{\mathcal{A}\mathsf{disMult}}$ does not output `DP`.
- P_i outputs $\langle \alpha_{\mathsf{x}} \alpha_{\mathsf{y}} \rangle$.

Semi-Honest Simulation Let P_j be the semi-honest party. $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ be the simulator.
- $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ has $(\langle\alpha_x\rangle_{is}, \langle\alpha_x\rangle_{tl})$, for all s, t and view generated by $\mathcal{S}_{\mathcal{A}}$ and the output.
- Simulation for S_2 terms: $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ initializes $\langle\alpha_x\alpha_y\rangle_{kl} = \langle\alpha_x\rangle_{kl} \cdot \langle\alpha_y\rangle_{kl}$, where $l \neq i, j, k$.
- Simulation for S_1 terms: $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ executes $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}\,\mathsf{disZK}}$ for maliciously corrupted P_i and semi-honest P_j. $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ sets the shares according to the protocol execution, for all the four terms $\delta_i, \delta_j, \delta_m, \delta_k$.
- Simulation for S_0 terms: $\mathcal{S}_{\mathcal{A}}$ executes $\mathcal{S}_{\mathcal{A}\,\mathsf{disMult}}$ for all the six terms of the summands S_0 for computing the terms of the $\langle\alpha_x\rangle_{uv} \cdot \langle\alpha_y\rangle_{pq}$, with P_i as malicious party.

Fig. 31: Simulator $\mathcal{S}_{\mathsf{tripGen}}$ for $\mathsf{tripGen}$

Multiplication Simulator: The simulator for mult(Fig. 4) appears in Fig. 32.

Simulator $\mathcal{S}_{\mathsf{mult}}$

Malicious Simulation Let P_i be the maliciously corrupt party. $\mathcal{S}_{\mathcal{A}}$ invokes $\mathcal{F}_{\mathsf{mult}}$ with input $[\![x]\!]_i$, $[\![y]\!]_i$ and receives $[\![z]\!]_i$ where $[\![\mathsf{v}]\!]_i = (\beta_{\mathsf{v}}, \langle\alpha_{\mathsf{v}}\rangle_i)$.

Preprocessing:
- $\mathcal{S}_{\mathcal{A}}$ has $\langle\alpha_{\mathsf{x}}\rangle_{ij}, \langle\alpha_{\mathsf{y}}\rangle_{ij}$ $\forall j \in \{i+1, i+2, i+3\}$.
- $\mathcal{S}_{\mathcal{A}}$ executes $\mathcal{S}_{\mathcal{A}\,\mathsf{tripGen}}$ on input $\langle\alpha_{\mathsf{x}}\rangle_i$, $\langle\alpha_{\mathsf{y}}\rangle_i$ and $\langle\alpha_{\mathsf{x}}\alpha_{\mathsf{y}}\rangle_i$, with P_i as the malicious party.
- $\mathcal{S}_{\mathcal{A}}$ outputs $\mathtt{DP}$ if $\mathcal{S}_{\mathcal{A}\,\mathsf{tripGen}}$ does, else continue.

Online:
- $\mathcal{S}_{\mathcal{A}}$ has $\beta_x, \langle\alpha_{\mathsf{x}}\rangle_{ij}, \beta_y, \langle\alpha_{\mathsf{y}}\rangle_{ij}, \langle\alpha_{\mathsf{x}}\alpha_{\mathsf{y}}\rangle_{ij}, \langle\alpha_{\mathsf{z}}\rangle_{ij}$ $\forall j \in \{i+1, i+2, i+3\}$.
- $\mathcal{S}_{\mathcal{A}}$ computes $\langle\beta_{\mathsf{z}}\rangle_{ij}$ correctly, for all $j \in \{i+1, i+2, i+3\}$.
- $\mathcal{S}_{\mathcal{A}}$ executes $\mathcal{S}_{\mathcal{A}\,\langle\cdot\rangle\text{-}\mathsf{Rec}}$ on input $\langle\beta_{\mathsf{z}}\rangle$ and output β_{z}, with P_i as the corrupted party.

Semi-Honest Simulation Let P_j be the semi-honest party. $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ be the simulator.

Preprocessing
- $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ has $(\langle\alpha_{\mathsf{x}}\rangle_{ik}, \langle\alpha_{\mathsf{x}}\rangle_{jm}, \langle\alpha_{\mathsf{y}}\rangle_{ik}, \langle\alpha_{\mathsf{y}}\rangle_{jm}$ $\forall j \neq i \& m \neq i, j, k)$ and view generated by $\mathcal{S}_{\mathcal{A}}$.
- $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ executes $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}\,\mathsf{tripGen}}$ with P_i as the malicious party and P_j as the semi-honest party.

Online
- $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ executes $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}\,[\![\cdot]\!]\text{-}\mathsf{Rec}}$ on $\langle\beta_{\mathsf{z}}\rangle$, with P_i as the malicious party and P_j as the semi-honest party.

Fig. 32: Simulator $\mathcal{S}_{\mathsf{mult}}$ for mult

Lemma 9 (Security). *The protocol* mult *(Fig. 4), realizes* $\mathcal{F}_{\mathsf{mult}}$ *(Fig. 30) with computational security in the* $(\mathcal{F}_{\mathsf{setup}}, \mathcal{F}_{\langle\cdot\rangle\text{-}\mathsf{Rec}}, \mathcal{F}_{\mathsf{disZK}}, \mathcal{F}_{\mathsf{disMult}})$*-hybrid model against* $(1,1)$-$\mathtt{FaF}$ *adversaries* $\mathcal{A}$, $\mathcal{A}_{\mathcal{H}}$*, controlling one one party each.*

Proof. Let P_i be the malicious party, controlled by $\mathcal{A}$.

Claim: The simulator $\mathcal{S}_{\mathcal{A}}$, described in Fig. 32, generates a transcript indistinguishable from P_i's view.

The transcript generated by $\mathcal{S}_{\mathcal{A}}$ in the preprocessing phase is the same as the transcript generated by $\mathcal{S}_{\mathcal{A}}$ of tripGen, which is indistinguishable from P_i's view of the pre-processing phase.

$\mathcal{S}_{\mathcal{A}}$ generates the transcript of the online phase by executing the simulator of $\langle\cdot\rangle$-Rec. Therefore, the transcript is indistinguishable.

Let P_j be the semi-honest party, controlled by $\mathcal{A}_{\mathcal{H}}$.

It is obvious due to the same reason mentioned above that the simulator $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$, described in Fig. 32, generates a transcript indistinguishable from P_j's view.

C.3 4PC FaF Protocol

4PC FaF Functionality: The ideal functionality for 4PC (Fig. 34) appears in Fig. 33.

Functionality $\mathcal{F}_{\text{4PC-FaF}}$

$\mathcal{F}_{\text{4PC-FaF}}$ interacts with the parties in $\mathcal{P}$ and the adversaries $\mathcal{S}_{\mathcal{A}}$ and $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$. Let x_s, y_s be the input and output corresponding to a party P_s respectively, i.e. $(y_0, y_1, y_2, y_3) = f(x_0, x_1, x_2, x_3)$.

– $\mathcal{F}_{\text{4PC-FaF}}$ receives $(\mathsf{Input}, \mathsf{x}_s)$ from $P_s \in \mathcal{P}$ and computes $(y_0, y_1, y_2, y_3) = f(x_0, x_1, x_2, x_3)$.

Output: Send (Output, y_s) to $P_s \in \mathcal{P}$.

Fig. 33: Ideal functionality for evaluating f in 4PC FaF Model

4PC FaF Protocol: The 4PC FaF protocol is as shown in Fig. 34.

Protocol 4PC

At each stage, the verification of all parallel instances of jmp3, jmp4 for every pair of parties is performed simultaneously with the send, in the same round.

Preprocessing Phase:

– For each input gate u, parties execute preprocessing phase of $[\![\cdot]\!]$-Sh to obtain $\langle\alpha_{\mathsf{u}}\rangle$.

– For each addition gate with input wires u, v and output wire w, parties locally compute $\langle\alpha_{\mathsf{w}}\rangle = \langle\alpha_{\mathsf{u}}\rangle + \langle\alpha_{\mathsf{v}}\rangle$.

– For each multiplication gate with input wires u, v and output wire w, parties execute preprocessing phase of mult to obtain $\langle\alpha_{\mathsf{w}}\rangle$ and $\langle\alpha_{\mathsf{u}}\alpha_{\mathsf{v}}\rangle$.

– For each output gate w, parties execute the preprocessing phase of $[\![\cdot]\!]$-Rec.

Online Phase:

– For each input v held by a client, parties invoke the online phase of $[\![\cdot]\!]$-Sh to obtain $[\![\mathsf{u}]\!]$.

– For each addition gate with input wires u, v and output wire w, parties locally compute $[\![\mathsf{w}]\!] = [\![\mathsf{u}]\!] + [\![\mathsf{v}]\!]$.

– For each multiplication gate with input wires u, v and output wire w, parties holding $([\![\mathsf{u}]\!], [\![\mathsf{v}]\!], \langle \alpha_{\mathsf{u}}\alpha_{\mathsf{v}} \rangle, \langle \mathsf{w} \rangle)$ execute the online phase of mult to obtain $[\![\mathsf{w}]\!]$.

– For each output gate, parties holding $[\![\mathsf{w}]\!]$ execute the online phase of $[\![\cdot]\!]$-Rec to reconstruct w towards the designated client.

2PC Phase:

If any dispute pair DP is identified either in preprocessing or online phase, execute ABY2.0 with TP, where in the latter case (dispute is identified in the online), all the parties perform share conversion as described in §6 to ensure 2PC sharing among parties in TP.

Fig. 34: 4PC FaF Protocol

Simulator $\mathcal{S}_{\mathcal{A}}{}^{P_i}$, $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$

Malicious Let P_i be the maliciously corrupted party.

– $\mathcal{S}_{\mathcal{A}}$ executes $\mathcal{S}_{\mathcal{A}[\![\cdot]\!]}$ (simulator for the input sharing) for the input gates for corrupt P_i.

– $\mathcal{S}_{\mathcal{A}}$ calls the functionality $\mathcal{F}_{\mathsf{4PC\text{-}FaF}}$ with P_i's input, $\mathcal{S}_{\mathcal{A}}$ holds the inputs of P_i due to the replication of the sharing semantic. and gets output z (say).

– For each output wire z, $\mathcal{S}_{\mathcal{A}}$ emulates $\mathcal{F}_{\mathsf{setup}}$ to generate $\langle \alpha_{\mathsf{z}} \rangle$. It computes the commitment of $\langle \alpha_{\mathsf{z}} \rangle$ on behalf of the honest parties and emulates $\mathcal{F}_{\mathsf{jmp4}}$ with respective inputs. If jmp4 fails, $\mathcal{S}_{\mathcal{A}}$ emulates 2PC.

– Since addition is local, nothing to simulate.

– $\mathcal{S}_{\mathcal{A}}$ executes $\mathcal{S}_{\mathcal{A}\mathsf{mult}}$ for corrupt P_i.

– Corresponding to the consistent openings, $\mathcal{S}_{\mathcal{A}}$ executes $\mathcal{S}_{\mathcal{A}\langle\cdot\rangle\text{-}\mathsf{Rec}}$ for β_{z} which is made consistent with the output z for the output wire obtained from $\mathcal{F}_{\mathsf{4PC\text{-}FaF}}$[a].

– $\mathcal{S}_{\mathcal{A}}$ opens the commitments of $\langle \alpha_{\mathsf{z}} \rangle$ corresponding to the honest parties shares of the output wire.

Semi-Honest Let P_j be the semi-honest party.

– $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ has inputs of P_i, P_j and the view of $\mathcal{S}_{\mathcal{A}}$, and the outputs of P_i, P_j.

– $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ executes $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}[\![\cdot]\!]}$ for the input gates for malicious P_i and semi-honest P_j.

– $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ executes $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}\mathsf{mult}}$ for malicious P_i, and semi-honest P_j.

[a] W.l.o.g we assume that the output is a multiplication gate. If not, then we execute this step for the multiplication gates which act as input to the output gate(s).

Fig. 35: Simulator $\mathcal{S}_{\mathcal{A}}{}^{P_i}$ for 4PC

The proof sketch of Theorem 4: We provide the proof in $(\mathcal{F}_{\mathsf{setup}}, \mathcal{F}_{\mathsf{OPE}})$-hybrid model. In particular, the proof follows a sequence of hybrids corresponding to each subprotocol as shown in Fig. 35. We provide the intuition for FaF security first followed by the proof. During the preprocessing phase of sharing, each pair of parties (P_i, P_j) pick a random value $\langle \alpha_{\mathsf{v}} \rangle_{ij}$ non-interactively. This ensures that each party misses 3 shares, and every pair of parties misses 1 share respectively. Since in the online phase β_{v} received by all the parties (to complete $[\![\mathsf{v}]\!]$-sharing) is computed by masking v with α_{v}, the masked value β_{v} is indistinguishable from a random value, even for a pair of parties, due to the missing share of α_{v}. This is precisely the scenario to be tackled for $(1,1)$-`FaF` security, where the malicious party can send its view to the semi-honest party to breach privacy. The same argument holds for each addition gate evaluation, since it is performed locally by parties on their own $[\![\cdot]\!]$ shares.

We will give an intuition of the multiplication protocol's security based on its component subprotocols and using the argument above for privacy of $[\![\cdot]\!]$-sharing. During the preprocessing, tripGen is invoked, which in turn invokes disMult to compute terms of S_0 of the type $\langle\alpha_{\mathsf{x}}\rangle_{ij}\langle\alpha_{\mathsf{y}}\rangle_{km}$. Here, the security holds in the $\mathcal{F}_{\mathsf{OPE}}$-hybrid model. Following this, the verification phase of disMult requires broadcasting hash values of the output received by P_i, P_j to check consistency, whose security is ensured by the underlying hash function. In case any inconsistency is identified at this stage, the parties reveal their inputs to the others. Since the inputs of the parties to this protocol are $\langle\alpha_{\mathsf{x}}\rangle_{ij}$ and $\langle\alpha_{\mathsf{y}}\rangle_{km}$, which are randomly picked even before the inputs to the actual circuit are available, such a revelation of inputs is secure. Moreover, after this stage a dispute set is identified which is ensured to include the malicious party, and the protocol either terminates (fairness) or moves to semi-honest 2PC protocol (GOD). In the latter case, security of the 2PC protocol ensures the security of our GOD protocol. Further, computation of summands of S_1 of the type $\langle\alpha_{\mathsf{x}}\rangle_{ij}\langle\alpha_{\mathsf{y}}\rangle_{ik}$ is secure from the perspective of the malicious party, since δ_i^1 (received by P_k) and δ_i^2 (held by P_j, P_m) are random values, they do not leak P_i's shares. To understand its security in the `FaF` model informally, we consider the following cases. First, if one of the corrupted parties is P_i, then security trivially holds. If the corrupted parties are P_j, P_m, together they hold δ_i^2, which is a random value. Finally, if the corrupted parties are P_k, P_m, the semi-honest party can learn $\delta_i^1 + \delta_i^2$ which is equivalent to learning $\sum_{(j,k)} \langle\alpha_{\mathsf{x}}\rangle_{ij} \cdot \langle\alpha_{\mathsf{y}}\rangle_{ik}$. However, due to the term $\langle\alpha_{\mathsf{x}}\rangle_{ij} \cdot (\langle\alpha_{\mathsf{y}}\rangle_{ik} + \langle\alpha_{\mathsf{y}}\rangle_{im}) + (\langle\alpha_{\mathsf{x}}\rangle_{ik} + \langle\alpha_{\mathsf{x}}\rangle_{im}) \cdot \langle\alpha_{\mathsf{y}}\rangle_{ij}$, of which 2 terms, specifically $\langle\alpha_{\mathsf{x}}\rangle_{ij}$ and $\langle\alpha_{\mathsf{y}}\rangle_{ij}$ are unknown to the semi-honest party, thus ensuring privacy of P_i's shares. Finally, the summands of S_2 require local computation, this privacy holds trivially. During the online phase of multiplication, parties perform local computation of $\langle\beta_{\mathsf{z}}\rangle$, which is z masked with α_{z} and open it towards all the parties. Here, the privacy of z is ensured by the randomness of α_{z}, which follows from the same argument as described for sharing.

Finally, the last step in protocol 4PC requires reconstruction of the output towards all the parties. During the preprocessing phase of $[\![\cdot]\!]$-Rec to reconstruct the output v towards all, parties commit to their shares of $\langle\alpha_{\mathsf{v}}\rangle$. The hiding property of the underlying commitment scheme ensures that the commitment values don't leak the shares of individual parties and thus ensures privacy of α_{v}. As described in §4.1, if the computation during the preprocessing phase fails, parties do not open the commitments to their shares, thus ensuring v is not obtained by any party. To extend the security to GOD, parties resort to 2PC semi-honest protocol, whose security ensures the security of our protocol. On the other hand, if the preprocessing phase of $[\![\cdot]\!]$-Rec succeeds then the protocol ensures fairness as follows. In the online phase, each party receives opening of every share of α_{v} from 2 parties, at least one of which is ensured to be (semi) honest, and hence provides an opening which is consistent with the commitment agreed upon in the preprocessing phase. This ensures that every party receives the output v, thus achieving fairness.

Proof of Theorem 4.

Proof. The simulator for 4PC appears in Fig. 35. The real world view of the protocol is indistinguishable from the simulated view. We prove it using a sequence of hybrids. Note that, adversaries' views in $[\![\cdot]\!]$-Sh are indistinguishable from the simulated views in $\mathcal{F}_{\mathsf{Sh}}$ (Lemma 6) and adversaries' views in mult are indistinguishable from the simulated views in $\mathcal{F}_{\mathsf{mult}}$ (Lemma 9). Finally, the simulators receive the output from the functionality $\mathcal{F}_{\mathsf{4PC\text{-}FaF}}$, and either simulate $[\![\cdot]\!]$-Rec or execute the simulator for semi-honest 2PC ABY2.0. In both cases, the simulated views are indistinguishable from the real views. Consider the following hybrids.

$\mathbf{Hybrid_0}$: 4PC: Execution of the 4 party protocol in the real world.

$\mathbf{Hybrid_1}$: $\mathcal{F}_{\mathsf{Sh}}$-Computation-Output: In this hybrid, sharing the inputs is performed using $\mathcal{F}_{\mathsf{Sh}}$ functionality followed by the real execution of the computation and output reconstruction as per 4PC. The distributions of $\mathbf{Hybrid_0}$ and $\mathbf{Hybrid_1}$ are indistinguishable due to Lemma 6.

$\mathbf{Hybrid_2}$: $\mathcal{F}_{\mathsf{Sh}}$-$\mathcal{F}_{\mathsf{mult}}$-Output: In this hybrid, sharing the inputs is done using $\mathcal{F}_{\mathsf{Sh}}$ functionality and the multiplication is performed using $\mathcal{F}_{\mathsf{mult}}$ functionality, followed by the real execution of the output reconstruction as per as per 4PC. The distributions of $\mathbf{Hybrid_1}$ and $\mathbf{Hybrid_2}$ are indistinguishable due to Lemma 9.

$\mathbf{Hybrid_3}$: $\mathcal{F}_{\mathsf{4PC\text{-}FaF}}$ - Execution of the 4 party protocol in the ideal world. The distributions of $\mathbf{Hybrid_2}$ and $\mathbf{Hybrid_3}$ are indistinguishable, since the simulator for 4PC internally invokes the simulator for $\langle\cdot\rangle$-Rec (as described in Fig. 35) which is indistinguishable from $\mathcal{F}_{\langle\cdot\rangle\text{-}\mathsf{Rec}}$ as shown in Lemma 7.

Thus, we conclude that distribution of $\mathbf{Hybrid_0}$ which is the protocol execution in the real world is computationally indistinguishable from the distribution of $\mathbf{Hybrid_3}$ corresponding to the execution in the ideal world.

D Challenges in Extension to nPC

We note that extending our 4PC protocol in FaF-model for arbitrary number of parties to handle more than one malicious corruption is non-trivial. We list the challenges involved below:

- Depending on the values of t and h^*, a share will be commonly held by a larger subset of parties (compared to our protocol where every pair holds a common share). This will have two immediate implications as below.
  - The joint message passing primitives (jmp3, jmp4) in our protocol operate under the assumption of a pair of parties holding a common value. In the protocol for n parties, a new joint message passing primitive would be required for dispute identification.

- The categorisation of summands in our triple generation protocol (tripGen) into types S_0, S_1, S_2 depends on the threshold of sharing. In the n party case, depending on the threshold, the categorisation of summands will vary and may require additional techniques for tackling each category.

– If the number of malicious parties is more than one, then the disZK protocol (used for handling summands of S_1) must tackle a malicious prover and malicious verifier(s) simultaneously. This may need additional primitives such as verifiable secret sharing (VSS) as used in [21], which is unknown in the FaF setting.

– The technique used to tackle summands of type S_0, when extended to n parties may require parallel executions of multiple OPEs in disMult. Moreover, the consistency may now require to hold among a larger subset of parties (holding a common share). Thus, the identification of a dispute pair is much more challenging.

– Even after the identification of a dispute pair, an execution of a semi-honest protocol need not be sufficient in the n party setting if the number of the malicious parties is more than one. A potential approach may require iterative runs of FaF secure protocols with reduced threshold. For instance, running a $n-2$ party $(t-1, h^*)$-FaF secure protocol after eliminating the parties in dispute pair.

We leave designing an efficient generic protocol in FaF setting as a potential future work. In fact, it is interesting to even design a $(t, 1)$-FaF secure protocol, where the number of semi-honest corruptions h^* is restricted to 1, as the (semi) honest parties may not collude.

Secondly, in this work, we design the PPML building blocks necessary for PPML inference. For training however, additional building blocks such as garbled circuit based protocols and efficient protocols for conversions between **A**rithmetic-**B**oolean-**Y**ao [37] domains are required in FaF setting. It is interesting to design the entire protocol suit to handle PPML training which we leave as a potential future work.

E Communication Complexity Analysis

In this section, we analyse the communication complexity of our protocols. Note that the lemmas are described in terms of the OPE instance relying on jmp4 and corresponding to two senders and two receivers described in disMult (§5.2). Unless stated otherwise, OPE refers to the instance relying on jmp4.

E.1 Sharing and Reconstruction Protocols

Lemma 10 (Communication). *Protocol* $[\![\cdot]\!]$-Sh *requires 2 rounds and communication of* 3 *elements in the online phase.*

Proof. The preprocessing phase is non-interactive. In the online phase, P_i sends β_{v} to P_j requiring one round and a communication of 1 element. This is followed by jmp4-send by P_i, P_j which requires one round and a communication of 2 elements.

Lemma 11 (Communication). *Protocol* $[\![\cdot]\!]$-Rec *requires* 1 *round and a communication of* 12κ *bits in the preprocessing phase, whereas* 1 *round and a communication of* 24 *elements in the online phase, for reconstructing a value towards all the parties.*

Proof. To robustly reconstruct a $[\![\cdot]\!]$-shared value v towards all the parties, in the preprocessing phase, each pair of parties P_i, P_j execute jmp4 to send $\mathsf{Com}(\langle \alpha_{\mathsf{v}} \rangle_{ij})$ to the other two parties, in parallel. This together requires one round and a communication of 12κ bits.

In the online phase, each pair of parties P_i, P_j, where $1 \le i < j \le 4$, open the commitment to the other two parties in parallel, which requires a communication of 4 elements from each pair P_i, P_j. Since six distinct P_i, P_j pairs open the commitments in parallel, the online phase requires one round and a communication of 24 elements.

Lemma 12 (Communication). *Protocol* $\langle \cdot \rangle$-Rec *requires three rounds and communication of* 7 *elements to reconstruct a value towards all parties.*

Proof. To reconstruct a value v towards all the parties, all the invocations of jmp3 towards P_3 are done in parallel, which requires one round and a communication of 3 elements. Following this, all the invocations of jmp3 towards P_4 are done in parallel, which requires one round and a communication of 2 elements. Now, P_3 and P_4 can reconstruct the value v. Further, P_3, P_4 execute jmp4 to send v to P_1, P_2, which requires one round and communication of 2 elements.

E.2 Multiplication Protocols

Lemma 13 (Communication). *Protocol* disMult *requires* 1 *instance of* OPE.

Proof. The protocol disMult described in Fig. 5 requires 1 instance of OPE relying on jmp4. This essentially incurs a cost equivalent to 2 instances of standard OPE. However, for all the subsequent protocols, we provide the costs in terms of the instance of OPE relying on jmp4.

Lemma 14 (Communication). *Protocol* tripGen *requires* 6 OPE *invocations and a communication of* 4 *elements.*

Proof. Terms in the summand of S_2 are computed locally, and parties generate the $\langle \cdot \rangle$-share of each of the term non-interactively. The summand of S_1 are computed in 4 parts, each of which is computed by a dedicated party. Each party adds the 6 terms of the summand and additively shares this computed value with only 1 element communication, followed by a distributed zero-knowledge proof, which is amortized across multiple executions of tripGen. Due to this amortization, disZK does not incur any additional overhead. This incurs a total cost of 4 elements for the summands of S_1. Terms in the summand of S_0 are computed using disMult, the cost for which is given in Lemma 13. There are 6 such terms, that aggregates to 6 invocations of OPEs.

Lemma 15 (Communication). *Protocol* mult *requires* 6 *instances of* OPE *and a communication of* 4 *elements in the preprocessing phase, whereas* 3 *rounds and a communication of* 7 *elements in the online phase.*

Proof. In the preprocessing phase, parties locally sample $\langle\cdot\rangle$-sharing of α_{z}, which is non-interactive. Further, parties invoke $\mathsf{tripGen}$, which requires 6 instances of OPEs for 6 instances of $\mathsf{disMult}$ and a communication of 4 elements for the summands in S_1. In the online phase, parties compute the $\langle\cdot\rangle$-sharing of β_{z}, which is non-interactive. This is followed by an invocation of $\langle\cdot\rangle$-Rec Fig. 3, which requires three rounds and a communication cost of 7 elements (Lemma 12).

Lemma 16 (Communication). $\mathsf{4PC}$ *achieves GOD from our fair protocol § 6; Fig. 34 without additional overhead in the online phase, and with additional* 12 *instances of standard* OPE*s in the preprocessing phase.*

Proof. To GOD variant of our protocol evaluates the circuit in segments, where each segment is executed similar to our fair protocol. Hence, either each segment succeeds, or a dispute pair is identified. In the former case, the cost of evaluating the segment is equivalent to that incurred in the fair variant. In the latter case, to complete the computation, the failed segment is rerun and all the following segments are executed using the state-of-the-art semi-honest 2PC of [70] with the parties outside the dispute pair. ABY2.0 [70] operates in the preprocessing paradigm and has an online cost of 2 elements. Although this may increase the cost per multiplication gate of the failed segment to 9 elements (7 elements for our fair protocol, and an additional 2 elements for rerunning 2PC of ABY2.0), note that this is a one-time cost incurred for a single segment. Every segment following this incurs a cost of only 2 elements per multiplication gate, thus resulting in a GOD protocol with the same online cost as that of the fair variant. Further, since [70] operates in the preprocessing paradigm, to complete the computation, parties need the preprocessing data for the 2PC protocol. Parties cannot do the preprocessing after identifying the dispute pair, as this may happen in the online phase. To circumvent this, every pair of parties executes the preprocessing of ABY2.0 [70] along with the preprocessing phase of our protocols. The preprocessing cost of a multiplication gate in ABY2.0 is 2 instances of OPE. Since every pair of parties compute the preprocessing data for ABY2.0, our protocol's GOD version incurs an extra cost of 12 instances of standard OPEs (without $\mathsf{jmp4}$) in the preprocessing phase.

F Building Blocks for Applications

We design the following tools for the applications considered, that is, liquidity matching and PPML inference– (i) input sharing and output reconstruction in SOC setting, ii) bit extraction, iii) bit to arithmetic conversion, iv) bit injection, v) ReLU and vi) dot product with truncation. Since we consider the applications in the SOC setting, we refer the parties who execute the computation as servers.

The $[\![\cdot]\!]$-sharing over Boolean ring is referred as Boolean sharing and denoted as $[\![\cdot]\!]^{\mathbf{B}}$. Additionally, our protocols use primitives referred to as *joint sharing* (jsh and jshRSS), which allow a pair of servers to generate a $[\![\cdot]\!]$ and $\langle\cdot\rangle$-sharing respectively, of a commonly known value. Below we provide the details of the building blocks.

F.1 Sharing and Reconstruction Protocol

$[\![\cdot]\!]$-$\mathsf{Sh}^{\mathsf{SOC}}$ (Fig. 36) enables a user U to $[\![\cdot]\!]$-share its input v. Similarly, protocol $[\![\cdot]\!]$-$\mathsf{Rec}^{\mathsf{SOC}}$ (Fig. 36) allows the servers to reconstruct a value v towards U. These are similar in spirit to $[\![\cdot]\!]$-Sh (Fig. 1) and $[\![\cdot]\!]$-Rec (Fig. 2) respectively.

To facilitate U to share v, servers generate a $\langle\cdot\rangle$-sharing of a random value α_{v} non-interactively via shared key setup and reconstruct α_{v} towards U. Then U can generate and send β_{v} to all the servers to complete $[\![\mathsf{v}]\!]$. Elaborately, each pair of servers commit to their common share of α_{v} and $\mathsf{jmp4\text{-}send}$ it to the other two servers. For a common share, say between P_i, P_j, the shared key setup is used to generate a common commitment. At this point either a $\mathtt{DP}$ is identified or every server holds the commitments for all shares of α_{v}. To ensure that the reconstruction of α_{v} towards U is always successful, each server sends all the commitments to U (for optimization, two servers can send the commitments and the remaining two the hash of the commitments). Since at most one server can be malicious, U accepts the value that forms a majority. Following this, every pair of servers open their common share of α_{v} to U. U accepts the consistent opening and reconstructs α_{v}. Finally, U sends $\beta_{\mathsf{v}} = \mathsf{v} + \alpha_{\mathsf{v}}$ to all the servers. Servers ensure that the sharing is correct by broadcasting the received value. If there exists a majority, accept that value, else set a default value.

To reconstruct a value v towards U, servers execute the preprocessing of $[\![\cdot]\!]$-Rec to agree upon the committed values of the shares of α_{v}. Each server sends β_{v} and the commitment of all the shares of α_{v}. Each pair of servers open their common share of α_{v} to U. U accepts the consistent opening and reconstructs α_{v}. Finally, it computes $\mathsf{v} = \beta_{\mathsf{v}} - \alpha_{\mathsf{v}}$.

If at any point, a $\mathtt{DP}$ is identified in either of the protocols, then servers signal the $\mathtt{DP}$s' identity to U. U selects $\mathtt{TP} = \mathcal{P} \backslash \mathtt{DP}$ as the one forming a majority and shares its input using additive sharing to the servers in $\mathtt{TP}$, who compute the function output and send it back to U.

Protocol $[\![\cdot]\!]$-$\mathsf{Sh}^{\mathsf{SOC}}(\mathsf{U}, \mathsf{v})$ and $[\![\cdot]\!]$-$\mathsf{Rec}^{\mathsf{SOC}}(\mathsf{U}, [\![\mathsf{v}]\!])$

- **Input, Output:** U has v. The servers output $[\![\mathsf{v}]\!]$.
- **Primitives:** $\mathsf{jmp4\text{-}send}$ and Com (§2).

Input Sharing:

– Every pair of severs, (P_i, P_j) sample $\langle\alpha_{\mathsf{v}}\rangle_{ij} \in \mathbb{Z}_{2^\lambda}$, using their common key.

– Every pair of servers, (P_i, P_j) $\mathsf{jmp4\text{-}send}$ $\mathsf{Com}(\langle\alpha_{\mathsf{v}}\rangle_{ij})$ to the remaining two servers.

– Each server sends $\mathsf{Com}(\langle \alpha_{\mathsf{v}} \rangle_{ij})$, where $1 \leq i < j \leq 4$, to U who accepts the values that form majority.
– Every pair of severs, (P_i, P_j) open $\langle \alpha_{\mathsf{v}} \rangle_{ij}$ towards U.
– U accepts the consistent openings, recovers $\langle \alpha_{\mathsf{v}} \rangle_{ij}$ for all i, j. It computes $\beta_{\mathsf{v}} = \mathsf{v} + \sum_{(i,j)} \langle \alpha_{\mathsf{v}} \rangle_{ij}$, and sends β_{v} to all four servers.
– Servers broadcast the received value and accept the majority value if it exists, and a default value, otherwise.

- **Input, Output:** The servers input $[\![\mathsf{v}]\!]$. U outputs v.
- **Primitives:** jmp4-send and Com (§2).

Output Reconstruction:

– Servers execute the preprocessing of $[\![\cdot]\!]$-$\mathsf{Rec}(\mathcal{P}, [\![\mathsf{v}]\!])$ to agree upon commitments of $\langle \alpha_{\mathsf{v}} \rangle_{ij}$, for all i, j.
– Each server sends β_{v} and commitments on $\langle \alpha_{\mathsf{v}} \rangle_{ij}$, for every i, j, to U, who accepts the values forming majority.
– P_i, P_j for all i, j open $\langle \alpha_{\mathsf{v}} \rangle_{ij}$ to U.
– U accepts the consistent opening and computes $\mathsf{v} = \beta_{\mathsf{v}} - \sum_{(i,j)} \langle \alpha_{\mathsf{v}} \rangle_{ij}$.

Fig. 36: 4PC Input Sharing and Output Reconstruction in SOC setting

Lemma 17 (Communication). *Protocol* $[\![\cdot]\!]$-$\mathsf{Sh}^{\mathsf{SOC}}$ *for a value* v *requires a communication of* 4 *rounds and* $36\kappa + 16\lambda$ *bits and* 4 *element broadcasts.*

Proof. Every pair of servers, (P_i, P_j) non-interactively sample $\langle \alpha_{\mathsf{v}} \rangle_{ij}$. This is followed by a jmp4 execution by each pair of servers, that costs 1 round and $6 \times 2\kappa$ bits of communication. Each server sends $\mathsf{Com}(\langle \alpha_{\mathsf{v}} \rangle_{ij})$ for all i, j to U, simultaneously each pair of servers open their common share to U, that adds 1 round and $24\kappa + 12\lambda$ bits of communication. U computes β_{v} and sends β_{v} to all the servers in the 3rd round that cost 4λ bits of communication. At the end, all the servers broadcast their received value to agree on a common value, that adds 4 element broadcast to the communication.

Lemma 18 (Communication). *Protocol* $[\![\cdot]\!]$-$\mathsf{Rec}^{\mathsf{SOC}}$ *for a value* v *requires a communication of* 2 *rounds and* $36\kappa + 16\lambda$ *bits.*

Proof. Each pair of servers (P_i, P_j) run jmp4 on $\mathsf{Com}(\langle \alpha_{\mathsf{v}} \rangle_{ij})$, that incurs a cost of 1 round and 12κ bits. Then every server sends β_{v} and $\mathsf{Com}(\langle \alpha_{\mathsf{v}} \rangle_{ij})$, for all i, j to U, that adds 1 round and $4\lambda + 24\kappa$ bits of communication, simultaneously, each pair of servers, (P_i, P_j) open $\langle \alpha_{\mathsf{v}} \rangle_{ij}$ to U, that costs 12λ bits of communication.

F.2 Dot Product Protocol

Given the $[\![\cdot]\!]$-sharing of vectors $\vec{\mathbf{x}}$ and $\vec{\mathbf{y}}$, protocol DotP (Fig. 37) allows servers to generate $[\![\cdot]\!]$-sharing of $z = \vec{\mathbf{x}} \odot \vec{\mathbf{y}}$ robustly, where $\odot$ represents the dot product operation. Here, $[\![\cdot]\!]$-sharing of a vector $\vec{\mathbf{x}}$ of size n indicates that each element

$\mathbf{x}_i \in \mathbb{Z}_{2^\lambda}$ of $\vec{\mathbf{x}}$, for $i \in [\mathsf{n}]$, is $[\![\cdot]\!]$-shared. We borrow ideas from BLAZE [71] for obtaining an online communication cost *independent* of n and use $\mathsf{jmp3}$ and $\mathsf{jmp4}$ primitives to ensure either success or TP selection. At a high-level, n-independent online phase is achieved by reconstructing the aggregated β_{z} (masked values), thanks to the linearity of $[\![\cdot]\!]$-sharing. Hence, the cost of a dot product is the same as a robust reconstruction in the online phase. Our dot product protocol essentially offloads the call to n parallel instances of $\mathsf{tripGen}$ to the preprocessing phase. Contrary to SWIFT, we cannot combine the n instances of $\mathsf{tripGen}$ since it requires OPE executions that are not aggregatable. The protocol appears in Fig. 37.

Protocol $\mathsf{DotP}(\mathcal{P}, ([\![\mathsf{x}^s]\!], [\![\mathsf{y}^s]\!])_{s \in [\mathsf{n}]})$

- **Input and Output:** The input is $[\![\vec{\mathbf{x}}]\!] = \{[\![\mathsf{x}^s]\!]\}_{s\in[\mathsf{n}]}, [\![\vec{\mathbf{y}}]\!] = \{[\![\mathsf{y}^s]\!]\}_{s\in[\mathsf{n}]}$. The output is $[\![\vec{\mathbf{x}} \odot \vec{\mathbf{y}}]\!]$.
- **Primitives:** Protocol $\mathsf{tripGen}$ (§5.2; Fig. 6) and $\langle\cdot\rangle$-Rec (§4.2; Fig. 3).

Preprocessing:

– Each P_i, P_j where $1 \le i < j \le 4$ sample random $\langle\alpha_{\mathsf{z}}\rangle_{ij} \in \mathbb{Z}_{2^\lambda}$.

– Servers invoke $\mathsf{tripGen}(\mathcal{P}, \langle\alpha_{\mathsf{x}}^s\rangle, \langle\alpha_{\mathsf{y}}^s\rangle)$ and obtain $\langle\alpha_{\mathsf{x}}^s\alpha_{\mathsf{y}}^s\rangle$, for $s \in [\mathsf{n}]$.

Online:

– Each P_i, P_j for $1 \le i < j \le 4$ and $(i,j) \neq (1,2)$ compute $\langle\beta_{\mathsf{z}}\rangle_{ij}$ such that $\langle\beta_{\mathsf{z}}\rangle_{ij} = \sum_{s=1}^{\mathsf{n}}(-\beta_{\mathsf{x}}^s\langle\alpha_{\mathsf{y}}^s\rangle_{ij} - \beta_{\mathsf{y}}^s\langle\alpha_{\mathsf{x}}^s\rangle_{ij} + \langle\alpha_{\mathsf{x}}^s\alpha_{\mathsf{y}}^s\rangle_{ij}) + \langle\alpha_{\mathsf{z}}\rangle_{ij}$.

– P_1, P_2 compute $\langle\beta_{\mathsf{z}}\rangle_{12} = \sum_{s=1}^{\mathsf{n}}(\beta_{\mathsf{x}}^s\beta_{\mathsf{y}}^s - \beta_{\mathsf{x}}^s\langle\alpha_{\mathsf{y}}^s\rangle_{12} - \beta_{\mathsf{y}}^s\langle\alpha_{\mathsf{x}}^s\rangle_{12} + \langle\alpha_{\mathsf{x}}^s\alpha_{\mathsf{y}}^s\rangle_{12}) + \langle\alpha_{\mathsf{z}}\rangle_{12}$.

– Servers invoke $\langle\cdot\rangle$-Rec and obtain β_{z}.

Fig. 37: Dot Product Protocol

Lemma 19 (Communication). *Protocol* DotP *with feature size* n *requires a communication of* $4\mathsf{n}$ *elements and* $6\mathsf{n}$ *instances of* OPE*s in the preprocessing phase, whereas* 3 *rounds and a communication of* 7 *elements in the online phase.*

Proof. In the preprocessing phase, servers locally sample $\langle\cdot\rangle$-sharing of α_{z}^s, which is non-interactive. Further, servers invoke $\mathsf{tripGen}$ for each $s \in [\mathsf{n}]$, which requires a communication of $4\mathsf{n}$ elements and $6\mathsf{n}$ instances of OPEs. In the online phase, servers compute the $\langle\cdot\rangle$-sharing of β_{z}, which is non-interactive. This is followed by an invocation of the $\langle\cdot\rangle$-Rec (Fig. 3) for the aggregated β_{z}, which requires three rounds and a communication cost of 7 elements (Lemma 12).

F.3 Joint RSS Sharing Protocol

Protocol jshRSS enables a pair of (unordered) servers (P_i, P_j) to jointly generate a $\langle\cdot\rangle$-sharing of value $\mathsf{v} \in \mathbb{Z}_{2^\lambda}$ known to both of them. Servers execute the protocol non-interactively. This makes the protocol robust. The protocol is described in Fig. 38.

Protocol jshRSS

- **Input and Output:** The input is v to P_i, P_j. The output is $\langle \mathsf{v} \rangle$ to all the servers.

– P_i, P_j compute $\langle \mathsf{v} \rangle_{ij} = \mathsf{v}$.
– All other pair of servers P_k, P_m set $\langle \mathsf{v} \rangle_{km} = 0$, for all $(k, m) \neq (i, j)$.

Fig. 38: $\langle \cdot \rangle$-sharing a value $\mathsf{v} \in \mathbb{Z}_{2^\lambda}$ jointly by P_i, P_j

Lemma 20 (Communication). *Protocol* jshRSS *of a value* v *is non-interactive.*

Proof. All the operations in protocol Fig. 38 are local and do not require any communication.

F.4 Joint Sharing Protocol

Protocol jsh enables a pair of (unordered) servers (P_i, P_j) to jointly generate a $[\![\cdot]\!]$-sharing of value $\mathsf{v} \in \mathbb{Z}_{2^\lambda}$ known to both of them. Servers execute the protocol non-interactively. This makes the protocol robust. The protocol is described in Fig. 39.

Protocol jsh

- **Input and Output:** The input is v to P_i, P_j. The output is $[\![\mathsf{v}]\!]$ to all the servers.

– All the servers sample β_{v} using their common key.
– P_i, P_j compute $\langle \alpha_{\mathsf{v}} \rangle_{ij} = \beta_{\mathsf{v}} - \mathsf{v}$.
– All other pair of servers P_k, P_m set $\langle \alpha_{\mathsf{v}} \rangle_{km} = 0$, for all $(k, m) \neq (i, j)$.

Fig. 39: $[\![\cdot]\!]$-sharing a value $\mathsf{v} \in \mathbb{Z}_{2^\lambda}$ jointly by P_i, P_j

Lemma 21 (Communication). *Protocol* jsh *of a value* v *is non-interactive.*

Proof. All the operations in protocol Fig. 39 are local and do not require any communication.

F.5 Bit Extraction Protocol

The bit extraction protocol, BitExt allows servers to compute Boolean sharing of the most significant bit (msb) of a value v from its arithmetic sharing $[\![\mathsf{v}]\!]$. Our bit extraction uses the optimized 2-input Parallel Prefix Adder (PPA) circuit proposed in [64] which works over bits, requiring the given arithmetic sharing to be converted to Boolean, which is challenging due to the presence of 6 component shares, each with λ bits. To tackle this challenge without blowing up the cost, we use a series of full adders (FAs) in an optimized way as described below. To compute the msb, servers use the optimized PPA circuit from ABY3 [64] consisting of $2\lambda - 2$ AND gates and having a multiplicative depth of $\log \lambda$. This circuit takes as input two Boolean values and outputs the msb of the sum of these inputs. The value v whose msb has to be computed is expressed as $\mathsf{v} = \beta_{\mathsf{v}} + (-\alpha_{\mathsf{v}})$ where $\alpha_{\mathsf{v}} = \sum_{(i,j)} \langle \alpha_{\mathsf{v}} \rangle_{ij}$. For brevity of description, let $x_1 = \langle \alpha_{\mathsf{v}} \rangle_{12}$, $x_2 = \langle \alpha_{\mathsf{v}} \rangle_{13}$, $x_3 = \langle \alpha_{\mathsf{v}} \rangle_{14}$, $x_4 = \langle \alpha_{\mathsf{v}} \rangle_{23}$, $x_5 = \langle \alpha_{\mathsf{v}} \rangle_{24}$, and $x_6 = \langle \alpha_{\mathsf{v}} \rangle_{34}$.

Note that β_{v} is held by all the servers and hence they can locally compute $[\![\beta_{\mathsf{v}}[i]]\!]^{\mathbf{B}}$. Here $\beta_{\mathsf{v}}[i]$ represents the i^{th} bit of β_{v}. Further, since each x_k is held by two servers, they execute jsh (Fig. 39) on each bit $x_k[i]$ to obtain $[\![x_k[i]]\!]^{\mathbf{B}}$. Finally, $[\![\alpha_{\mathsf{v}}[i]]\!]^{\mathbf{B}} = [\![\sum_{k \in [6]} x_k[i]]\!]^{\mathbf{B}}$ is obtained from $[\![x_k[i]]\!]^{\mathbf{B}}$ using the Full Adder (FA) given in ABY3. Here, $\mathsf{FA}(p[i], q[i], r[i]) \rightarrow (c[i], s[i])$, for all $i \in \{0, 1, \ldots, \lambda - 1\}$ is such that $p + q + r = 2c + s$. Servers compute $[\![\alpha_{\mathsf{v}}[i]]\!]^{\mathbf{B}}$ simultaneously for all $i \in [\lambda]$ using FA as follows:

– (1) $\mathsf{FA}(x_1[i], x_2[i], x_3[i]) \rightarrow (c_1[i], s_1[i])$; (2) $\mathsf{FA}(x_4[i], x_5[i], x_6[i]) \rightarrow (c_2[i], s_2[i])$;
– (3) $\mathsf{FA}(c_1[i-1], s_1[i], c_2[i-1]) \rightarrow (c_3[i], s_3[i])$; (4) $\mathsf{FA}(s_2[i], c_3[i-1], s_3[i]) \rightarrow (c_4[i], s_4[i])$;
– (5) $\mathrm{PPA}(2c_4, s_4) \rightarrow \alpha_{\mathsf{v}}$.

Here, the first two FA evaluations are run in parallel and the next two are sequentially executed to compute $[\![(2c_1 + s_1 + 2c_2 + s_2)[i]]\!]^{\mathbf{B}}$ where $2c[i] = c[i-1]$ and $c[-1] = 0$. Finally, servers compute msb(v) by running the optimized PPA circuit on $[\![\beta_{\mathsf{v}}]\!]^{\mathbf{B}}$ and $[\![\alpha_{\mathsf{v}}]\!]^{\mathbf{B}}$.

Protocol BitExt

- **Input and Output:** The input is $[\![\mathsf{v}]\!] = (\beta_{\mathsf{v}}, \langle \alpha_{\mathsf{v}} \rangle)$. The output is $[\![\mathsf{msb}(\mathsf{v})]\!]^{\mathbf{B}}$.
- **Primitives:** Protocol jsh (§F;Fig. 39), 4PC (§6; Fig. 34).

Preprocessing

– Servers compute $[\![\alpha_{\mathsf{v}}[i]]\!]^{\mathbf{B}}$, where $\alpha_{\mathsf{v}}[i]$ is the ith bit of α_{v}, in the following way: for each $i \in \{0, \ldots, \lambda - 1\}$,
  – P_s, P_t jsh $\langle \alpha_{\mathsf{v}} \rangle_{st}[i]$, for all s, t.
  – Servers execute $\mathsf{FA}(\langle \alpha_{\mathsf{v}} \rangle_{12}[i], \langle \alpha_{\mathsf{v}} \rangle_{13}[i], \langle \alpha_{\mathsf{v}} \rangle_{14}[i])$, $\mathsf{FA}(\langle \alpha_{\mathsf{v}} \rangle_{23}[i], \langle \alpha_{\mathsf{v}} \rangle_{24}[i], \langle \alpha_{\mathsf{v}} \rangle_{34}[i])$ parallelly, and obtain $(c_1[i], s_1[i]), (c_2[i], s_2[i])$ respectively.
  – Servers execute $\mathsf{FA}(c_1[i-1], s_1[i], c_2[i-1])$ and obtain $(c_3[i], s_3[i])$, where $c_1[-1] = c_2[-1] = 0$.
  – Servers execute $\mathsf{FA}(s_2[i], c_3[i-1], s_3[i])$ and obtain $(c_4[i], s_4[i])$, where $c_3[-1] = 0$.
  – Servers compute $[\![\alpha_{\mathsf{v}}[i]]\!]^{\mathbf{B}} = [\![(2c_4[i] + s_4[i])]\!]^{\mathbf{B}}$, by evaluating PPA circuit on $[\![2c_4[i]]\!]^{\mathbf{B}}, [\![s_4[i]]\!]^{\mathbf{B}}$.

– Servers execute preprocessing for optimized PPA circuit.

Online

– Servers obtain $[\![\mathsf{msb}(\mathsf{v})]\!]^{\mathbf{B}}$ by executing online phase of optimized PPA circuit on $\beta_{\mathsf{v}}[i]$ and $\alpha_{\mathsf{v}}[i]$.

Fig. 40: Maximum Bit Extraction of a value v

Lemma 22 (Communication). *Protocol* BitExt *requires a communication cost of* $(52\lambda+11\lambda\log\lambda-8)$ *bits communication and* $(72\lambda+12\lambda\log\lambda-24)$ $\mathsf{OT_1}$*s in the preprocessing phase and require* $3\log\lambda$ *rounds and an amortized communication of* $14(\lambda-1)$ *bits in the online phase.*

Proof. In the preprocessing phase, each pair of servers execute jsh, which is non-interactive. BitExt requires evaluation of 4λ FA. This comprises 4λ AND gates. Following this, the computation of a PPA circuit, that involves $\lambda\log\lambda$ AND gates evaluation. Therefore, servers invoke mult for $(4\lambda+\lambda\log\lambda)$ AND gates, where the cost of mult for a single AND gate is 11 bits and $12\mathsf{OT_1}$s. Since one evaluation of optimized PPA is required in online, the preprocessing of this is executed in the offline. Optimized PPA consists of $2(\lambda-1)$ AND gates, corresponding to which servers execute $\mathsf{tripGen}$ for $2(\lambda-1)$ AND gates. Cost of $\mathsf{tripGen}$ for a single AND gate is 4 bits and 12 $\mathsf{OT_1}$s. In total, this accounts for a communication cost of $11(4\lambda+\lambda\log\lambda)+4*2(\lambda-1)$ bits i.e., $(52\lambda+11\lambda\log\lambda-8)$ bits and $12(4\lambda+\lambda\log\lambda)+12*2(\lambda-1)=(72\lambda+12\lambda\log\lambda-24)$ $\mathsf{OT_1}$s.

In the online phase, servers execute the online phase of the optimized PPA circuit, that is, the online phase of $2(\lambda-1)$ AND gates, which incurs a communication cost of $14(\lambda-1)$ bits and it requires $3\log\lambda$ rounds.

F.6 Bit to Arithmetic Protocol

Given the Boolean sharing of a bit b, denoted as $[\![\mathsf{b}]\!]^{\mathbf{B}}$, protocol $\mathsf{Bit2A}$ allows servers to compute the arithmetic sharing $[\![\mathsf{b}^{\mathsf{R}}]\!]$ where b^{R} denotes the value b over $\mathbb{Z}_{2^\lambda}$. We present a protocol whose preprocessing cost is approximately half of our multiplication protocol. Note that, β_{b} is available to all the servers in clear, so we consider $[\![\beta_{\mathsf{b}}]\!]_i=(\beta_{\mathsf{b}},0,0,0)$ for all P_i. Servers preprocess $\langle{\alpha_{\mathsf{b}}}^{\mathsf{R}}\rangle$ and non-interactively obtain the $[\![{\alpha_{\mathsf{b}}}^{\mathsf{R}}]\!]$. Finally, servers obtain $[\![\mathsf{b}^{\mathsf{R}}]\!]$ by performing a multiplication in the online phase. Thus the main challenge in $\mathsf{Bit2A}$ is to compute $\langle{\alpha_{\mathsf{b}}}^{\mathsf{R}}\rangle$. We provide an efficient computation of $\langle{\alpha_{\mathsf{b}}}^{\mathsf{R}}\rangle$.

In the bit to arithmetic protocol, the conversion of Boolean shared value to the arithmetic domain requires evaluation of arithmetic equivalent of $\mathbf{x}\oplus\mathbf{y}$ which is $\mathbf{x}+\mathbf{y}-2\mathbf{xy}$. Extending this to our protocol involves several sequential multiplication operations, due to the 6 components in the Boolean sharing each of which requires to be shared in the arithmetic domain. For efficiency, we make non-black-box use of our sharing by leveraging the fact that each component is jointly held by two parties. Here, we give the technical details, followed by the complete bit to arithmetic protocol in Fig. Fig. 41. Recall that $\mathsf{b}^{\mathsf{R}}=(\beta_{\mathsf{b}}\oplus\alpha_{\mathsf{b}})^{\mathsf{R}}=$

$\beta_{\mathsf{b}}{}^{\mathsf{R}} + \alpha_{\mathsf{b}}{}^{\mathsf{R}} - 2\beta_{\mathsf{b}}{}^{\mathsf{R}}\alpha_{\mathsf{b}}{}^{\mathsf{R}}$, where $\alpha_{\mathsf{b}}{}^{\mathsf{R}} = (x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_5 \oplus x_6)^{\mathsf{R}}$. To proceed with this computation, we denote $\alpha_{\mathsf{b}}{}^{\mathsf{R}} = \Delta_1 + \Delta_2 - 2\Delta_1 \cdot \Delta_2$ where $\Delta_1 = (x_1 \oplus x_2 \oplus x_3)^{\mathsf{R}}$ and $\Delta_2 = (x_4 \oplus x_5 \oplus x_6)^{\mathsf{R}}$. Due to the heavy notations involved, we decompose the protocol into smaller components as described below.

Computation of Δ_1: Note that, $\Delta_1 = (x_1 + x_2 + x_3 - 2x_1x_2 - 2x_1x_3 - 2x_2x_3 + 4x_1x_2x_3)$. Since all the terms in Δ_1 are held by P_1, it can compute Δ_1 locally. The computation of Δ_1 proceeds as follows. Servers first generate $\langle\cdot\rangle$-sharing of $y_1 = x_1x_2$, $y_2 = x_1x_3$, $y_3 = x_2x_3$, and $z_1 = x_1x_2x_3 = y_1x_3$. From the above shares servers locally obtain $\langle\Delta_1\rangle$. We will discuss how servers compute $\langle y_1\rangle, \langle z_1\rangle$. For y_2 and y_3, the computation proceeds similarly.

Computation of y_1: Recall that x_1 is joint RSS shared among all the servers such that $\langle x_1\rangle_{ij} = 0\ \forall(i,j) \neq (1,2)$, similarly $\langle x_2\rangle_{ij} = 0\ \forall(i,j) \neq (1,3)$. P_1, P_2 pick $\langle y_1\rangle_{12}$ using their common key. P_1 sends $\langle y_1\rangle_{13} = x_1x_2 - \langle y_1\rangle_{12}$ to P_3. Furthermore, P_1 gives a proof of honest computation using distributed zero-knowledge protocol disZK (Fig. 14), where P_2, P_3 act as verifiers.

Computation of z_1: Note that $z_1 = y_1x_3$. $\langle\cdot\rangle$-sharing of y_1 and x_3 is such that $\langle y_1\rangle_{ij} = 0$ for all pairs excluding $(i,j) = (1,2),(1,3)$ and $\langle x_3\rangle_{ij} = 0$ for all pairs excluding $(i,j) = (1,4)$. P_1, P_2 and P_1, P_3 pick $\langle z_1\rangle_{12}$ and $\langle z_1\rangle_{13}$ using their common keys. P_1 computes $\langle z_1\rangle_{14} = y_1x_3 - (\langle z_1\rangle_{12} + \langle z_1\rangle_{13})$ and sends it to P_4. P_1 gives a proof of honest computation using distributed zero-knowledge protocol disZK (Fig. 14), where P_2, P_3, P_4 act as verifiers.

Servers locally obtain $\langle\Delta_1\rangle = \langle x_1\rangle + \langle x_2\rangle + \langle x_3\rangle - 2\langle y_1\rangle - 2\langle y_2\rangle - 2\langle y_3\rangle + 4\langle z_1\rangle$. Note that, $\langle\Delta_1\rangle_{ij} = 0$ for all $i \neq 1$. *Computation of Δ_2:* $\Delta_2 = (x_4 \oplus x_5 \oplus x_6) = z_2 + x_6 - 2z_2x_6$, where $z_2 = x_4 + x_5 - 2y_4$, and $y_4 = x_4x_5$. Let $z_3 = z_2x_6$. First we will discuss the computation of y_4.

Computation of y_4: Note that, x_4 and x_5 is joint RSS shared among all the servers such that $\langle x_4\rangle_{ij} = 0\ \forall(i,j) \neq (2,3)$, similarly $\langle x_5\rangle_{ij} = 0\ \forall(i,j) \neq (2,4)$. The computation is similar to the computation of y_1. Finally, servers obtain $\langle y_4\rangle$ such that $\langle y_4\rangle_{ij} = 0$ for all excluding $(i,j) = (2,3),(2,4)$.

Computation of z_2: Servers compute $\langle z_2\rangle = \langle x_4\rangle + \langle x_5\rangle - 2\langle y_4\rangle$.

Computation of z_3: Observe that $\langle z_2\rangle_{ij} = 0$ for all pairs $(i,j) \neq (2,3),(2,4)$. Therefore, $z_3 = x_6(\langle z_2\rangle_{23} + \langle z_2\rangle_{24})$. We set $y_5 = x_6\langle z_2\rangle_{23}$ and $y_6 = x_6\langle z_2\rangle_{24}$ so that computation of y_5 and y_6 can be done locally by P_3 and P_4 respectively. Here we obtain $\langle z_3\rangle$ by locally adding shares of y_5 and y_6. Furthermore, $\langle z_3\rangle_{ij} = 0$ for all pairs $(i,j) \neq (2,3),(2,3),(3,4)$.

Computation of $\Delta_1 \cdot \Delta_2$: Recall that servers obtain $\langle\Delta_1\rangle$ and $\langle\Delta_2\rangle$ such that $\langle\Delta_1\rangle_{ij} = 0$ if $i \neq 1$ and $\langle\Delta_2\rangle_{ij} = 0$ if $i = 1$. Therefore, $\langle\Delta_1\Delta_2\rangle$ has the following non-zero terms $\sum_{j\in\{2,3,4\}} \langle\Delta\rangle_{1j}\langle\Delta_2\rangle_{23} + \langle\Delta\rangle_{1j}\langle\Delta_2\rangle_{24} + \langle\Delta\rangle_{1j}\langle\Delta_2\rangle_{34}$. Out of these nine terms P_2 computes $\langle\Delta_1\rangle_{12}\langle\Delta_2\rangle_{23}$, $\langle\Delta_1\rangle_{12}\langle\Delta_2\rangle_{24}$ and share with P_1, P_3, P_4. Similarly, P_3, P_4 computes $\langle\Delta_1\rangle_{13}\langle\Delta_2\rangle_{23}$, $\langle\Delta_1\rangle_{13}\langle\Delta_2\rangle_{34}$ and $\langle\Delta_1\rangle_{14}\langle\Delta_2\rangle_{24}$, $\langle\Delta_1\rangle_{14}\langle\Delta_2\rangle_{34}$ and share with P_1, P_2, P_4 and P_1, P_2, P_3 respectively. For the remaining terms servers execute $\mathsf{disMult}$. For each such terms, 2 OPEs are required. So, total 6 OPEs are needed.

Finally, servers obtain $\langle \alpha_{\mathsf{b}}{}^{\mathsf{R}} \rangle$. Now in the online phase, to compute $\mathsf{b}^{\mathsf{R}} = \beta_{\mathsf{b}}{}^{\mathsf{R}} + \alpha_{\mathsf{b}}{}^{\mathsf{R}} - 2\beta_{\mathsf{b}}{}^{\mathsf{R}}\alpha_{\mathsf{b}}{}^{\mathsf{R}}$. Since, $[\![\beta_{\mathsf{b}}{}^{\mathsf{R}}]\!] = (\beta_{\mathsf{b}}, 0)$ and $[\![\alpha_{\mathsf{b}}{}^{\mathsf{R}}]\!] = (0, \alpha_{\mathsf{b}})$, servers perform a multiplication and obtain $[\![\mathsf{b}^{\mathsf{R}}]\!]$.

Protocol Bit2A

- **Input and Output:** The input is $[\![\mathsf{b}]\!]^{\mathbf{B}}$ and the output is $[\![\mathsf{b}^{\mathsf{R}}]\!]$.
- **Primitives:** Protocol mult, tripGen, disZK (Section 5;Fig. 6, Fig. 4, Section 2 ;Fig. 14).

Preprocessing:

Computation of Δ_1:

- P_1, P_2 pick $\langle y_1 \rangle_{12}, \langle z_1 \rangle_{12}$, P_1, P_3 pick $\langle y_2 \rangle_{13}, \langle z_1 \rangle_{13}$, and P_1, P_4 pick $\langle y_3 \rangle_{14}$ using their respective common keys.
- P_1 computes $\langle y_1 \rangle_{13} = x_1x_2 - \langle y_1 \rangle_{12}$, $\langle y_2 \rangle 14 = x_2x_3 - \langle y_2 \rangle_{13}$, $\langle y_3 \rangle 12 = x_1x_3 - \langle y_3 \rangle_{14}$, $\langle z_1 \rangle 14 = x_1x_2x_3 - \langle z_1 \rangle_{12} - \langle z_1 \rangle_{13}$.
- P_1 sends $\langle y_3 \rangle_{12}$ to P_2, $\langle y_1 \rangle_{13}$ to P_3, and $\langle y_2 \rangle_{14}, \langle z_1 \rangle_{14}$ to P_4. Rest of the shares of y_1, y_2, y_3, z_1 is set to 0.
- Each pair of servers set $\langle \Delta_1 \rangle_{ij} = (\sum_{i=1}^{3} \langle x_i \rangle_{ij}) - 2(\sum_{i=1}^{3} \langle y_i \rangle_{ij}) + 4\langle z_1 \rangle_{ij}$.
- Servers run disZK to verify P_1's honest behaviour.

Computation of Δ_2:

- P_2, P_3 pick $\langle y_4 \rangle_{23}, \langle y_5 \rangle_{23}$ and P_2, P_4 pick $\langle y_6 \rangle_{24}$ using their respective common keys.
- P_2 sends $\langle y_4 \rangle_{24} = x_4x_5 - \langle y_4 \rangle_{23}$ to P_4. For all other pairs (i, j) $\langle y_4 \rangle_{ij} = 0$.
- Every pair of servers locally obtain $\langle z_2 \rangle_{ij} = \langle x_4 \rangle_{ij} + \langle x_5 \rangle_{ij} - 2\langle y_4 \rangle_{ij}$.
- Servers run disZK to verify P_2's honest behaviour.
- P_3 sends $\langle y_5 \rangle_{34} = x_6 \langle z_2 \rangle_{23} - \langle y_5 \rangle_{23}$ to P_4 and P_4 sends $\langle y_6 \rangle_{34} = x_6 \langle z_2 \rangle_{24} - \langle y_6 \rangle_{24}$ to P_3. Remaining pair of servers set the shares as 0.
- Each pair of servers set $\langle \Delta_2 \rangle_{ij} = \langle z_2 \rangle_{ij} + \langle x_6 \rangle_{ij} - 2(\langle y_5 \rangle_{ij} + \langle y_6 \rangle_{ij})$.
- Servers run disZK to verify P_3, P_4's honest behaviour.

Computation of $\Delta_1\Delta_2$:

- P2 computes and shares $\langle \Delta_1 \rangle_{12} \langle \Delta_2 \rangle_{23} + \langle \Delta_1 \rangle_{12} \langle \Delta_2 \rangle_{24}$.
- P3 computes and shares $\langle \Delta_1 \rangle_{13} \langle \Delta_2 \rangle_{23} + \langle \Delta_1 \rangle_{13} \langle \Delta_2 \rangle_{34}$.
- P4 computes and shares $\langle \Delta_1 \rangle_{14} \langle \Delta_2 \rangle_{24} + \langle \Delta_1 \rangle_{14} \langle \Delta_2 \rangle_{34}$.
- Servers run disZK to verify P_2, P_3, andP_4's honest behaviour.
- Servers run disMult for $\langle \Delta_1 \rangle_{12} \langle \Delta_2 \rangle_{34}$, $\langle \Delta_1 \rangle_{13} \langle \Delta_2 \rangle_{24}$, and $\langle \Delta_1 \rangle_{14} \langle \Delta_2 \rangle_{23}$.
- From prior steps, each pair of servers locally obtain $\langle \Delta_1\Delta_2 \rangle$.

Each pair of servers set $\langle \alpha_{\mathsf{b}}{}^{\mathsf{R}} \rangle_{ij} = \langle \Delta_1 \rangle_{ij} + \langle \Delta_2 \rangle_{ij} - 2\langle \Delta_1\Delta_2 \rangle_{ij}$.

Online:

- Servers set $[\![\beta_{\mathsf{b}}{}^{\mathsf{R}}]\!] = (\beta_{\mathsf{b}}{}^{\mathsf{R}}, 0)$, where $\langle 0 \rangle = 0$ for all pairs, and $[\![\alpha_{\mathsf{b}}{}^{\mathsf{R}}]\!] = (0, \alpha_{\mathsf{b}}{}^{\mathsf{R}})$.

– Let $w = \beta_{\mathsf{b}}{}^{\mathsf{R}}\alpha_{\mathsf{b}}{}^{\mathsf{R}}$. Servers non-interactively pick $\langle \alpha_w \rangle$ using their common keys. Note that $\alpha_{\alpha_{\mathsf{b}}{}^{\mathsf{R}}}\alpha_{\beta_{\mathsf{b}}{}^{\mathsf{R}}} = 0$. Each pair of servers set $\langle \beta_w \rangle_{ij} = -\beta_{\mathsf{b}}{}^{\mathsf{R}}\langle \alpha_{\mathsf{b}}{}^{\mathsf{R}} \rangle_{ij} + \langle \alpha_w \rangle_{ij}$.
– Servers reconstruct β_w and obtain $[\![w]\!]$.
– Servers locally obtain $[\![\mathsf{b}^{\mathsf{R}}]\!] = [\![\beta_{\mathsf{b}}{}^{\mathsf{R}}]\!] + [\![\alpha_{\mathsf{b}}{}^{\mathsf{R}}]\!] - 2[\![w]\!]$.

Fig. 41: Boolean to Arithmetic Protocol

Simulation: Let P_1 be the malicious server. Then $\mathcal{S_A}^{P_1}$ has x_1, x_2, x_3. It can correctly simulate computation of Δ_1. For Δ_2, P_1 shares of P_1 are all 0s, so nothing to simulate. To compute $\Delta_1\Delta_2$, $\mathcal{S_A}^{P_1}$ executes $\mathcal{S_A}^{P_1}_{\mathsf{disZK}}$, $\mathcal{S_A}^{P_1}_{\mathsf{disMult}}$, and for the online part it executes $\mathcal{S_A}^{P_1}_{\langle\cdot\rangle\text{-}\mathsf{Rec}}$.

If p_2 is semi-honest, $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}}$ has x_1, x_2, x_3, x_4, x_5, therefore it simulates till computation of z_2 by following the protocol correctly. For computing z_3 and so Δ_2, it executes $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}\mathsf{disZK}}$ with malicious P_1 and semi-honest P_2. To simulate the computation of $\Delta_1\Delta_2$ and the online phase $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}\mathsf{disMult}}$ and $\mathcal{S}_{\mathcal{A}_{\mathcal{H}}\langle\cdot\rangle\text{-}\mathsf{Rec}}$ is executed.

Simulation for other corruption scenarios are similar, where appropriate simulators are executed.

Lemma 23 (Communication). *Protocol* Bit2A *requires* 13 *elements and* 3 OPE*s in the pre-processing phase, and* 3 *rounds and* 7 *elements in the online phase.*

Proof. In the preprocessing phase, $y_1, y_2, y_3, y_4, y_5, y_6$ requires 6 elements, z_1 requires 1 element, z_2, z_3 is obtained by local computation. Computation of $\Delta_1\Delta_2$ needs 3 disMult and 6 elements communication. That incurs a total cost of 13 elements and 3 OPEs. In the online phase, reconstruction of β_w requires 3 rounds and 7 elements communication, rest of the computation is local.

F.7 Bit Injection Protocol

Given $[\![\mathsf{b}]\!]^{\mathbf{B}}$ for a bit b, and $[\![\mathsf{v}]\!]$ for $\mathsf{v} \in \mathbb{Z}_{2^\lambda}$, BitInj computes $[\![\cdot]\!]$-sharing of bv. Towards this, servers first execute Bit2A on $[\![\mathsf{b}]\!]^{\mathbf{B}}$ to generate $[\![\mathsf{b}]\!]$. This is followed by servers computing $[\![\mathsf{bv}]\!]$ by executing mult on $[\![\mathsf{b}]\!]$ and $[\![\mathsf{v}]\!]$.

Protocol BitInj

- **Input and Output:** The input is $[\![\mathsf{b}]\!]^{\mathbf{B}}$, $[\![\mathsf{v}]\!]$. The output is $[\![\mathsf{bv}]\!]$.
- **Primitives:** Protocol Bit2A (§F.6;Fig. 41), mult (§5; Fig. 4).

– Servers execute Bit2A run on $[\![\mathsf{b}]\!]^{\mathbf{B}}$ and obtain $[\![\mathsf{b}]\!]$.
– Servers execute mult on $[\![\mathsf{b}]\!]$ and $[\![\mathsf{v}]\!]$ and get output $[\![\mathsf{bv}]\!]$.

Fig. 42: Bit Injection Protocol

Lemma 24 (Communication). *Protocol* BitInj *requires an amortized communication cost of* 17 *elements,* 18 OPE*s in the preprocessing phase and* 6 *round and an amortized cost of* 14 *elements in the online phase.*

Proof. For BitInj, given Boolean sharing of a bit b, Bit2A requires 13 elements, 3 OPEs and in the online phase 3 rounds and 7 elements (23). Bit2A is followed by an execution of mult requires 4 elements, 6 OPEs in the preprocessing and 7 elements in the online phase. That incurs a total cost of 17 elements, 9 OPEs in the preprocessing phase and in the online phase incurs a communication cost of 14 elements and 6 rounds.

F.8 ReLU Protocol

The ReLU function, $\mathsf{relu}(\mathsf{v}) = \max(0, \mathsf{v})$, can be viewed as $\mathsf{relu}(\mathsf{v}) = \overline{\mathsf{b}} \cdot \mathsf{v}$, where bit $\mathsf{b} = 1$ if $\mathsf{v} < 0$ and 0 otherwise. Here $\overline{\mathsf{b}}$ denotes the complement of b. Given $[\![\mathsf{v}]\!]$, servers execute BitExt on $[\![\mathsf{v}]\!]$ to generate $[\![\mathsf{b}]\!]^{\mathbf{B}}$. $[\![\overline{\mathsf{b}}]\!]^{\mathbf{B}}$ is locally computed as $[\![\overline{\mathsf{b}}]\!]^{\mathbf{B}} = 1 \oplus [\![\mathsf{b}]\!]^{\mathbf{B}}$. Servers execute BitInj protocol on $[\![\overline{\mathsf{b}}]\!]^{\mathbf{B}}$ and $[\![\mathsf{v}]\!]$ to obtain the desired result.

Lemma 25 (Communication). *Protocol* relu *requires an amortized communication cost of* $(69\lambda + 11\lambda \log \lambda - 8)$ *bits,* 9 OPE*s,* $(72\lambda + 12\lambda \log \lambda - 24)$ OT_1*s in the preprocessing phase and requires* $3(\log \lambda + 2)$ *rounds and an amortized communication cost of* $28\lambda - 14$ *bits with fairness guarantee.*

Proof. One instance of relu protocol comprises of execution of one instance of BitExt, followed by BitInj. The cost, therefore follows from Lemma 22, and Lemma 24.

F.9 Sigmoid Protocol

$\mathsf{sig}(\mathsf{v}) = \overline{\mathsf{b}}_1 \mathsf{b}_2(\mathsf{v} + 1/2) + \overline{\mathsf{b}}_2$, where $\mathsf{b}_1 = 1$ if $\mathsf{v} + 1/2 < 0$ and $\mathsf{b}_2 = 1$ if $\mathsf{v} - 1/2 < 0$. The computation of MPC-friendly variant of sigmoid function is similar to the ReLU function. We follow similar approach of [58].

F.10 Dot Product with Truncation Protocol

In FPA, repeated multiplication causes overflow, resulting in loss of significant bits of information, thus affecting accuracy. Truncation tackles this by re-adjusting the shares after multiplication, such that the loss of information occurs on the least significant bits, which minimizes the accuracy loss [66]. We provide a dot product protocol with truncation using techniques form [64]. If x^d denotes the truncated value of x, then given an $(\mathsf{r}, \mathsf{r}^d)$ pair, v^d can be obtained by $(\mathsf{v} - \mathsf{r})^d + \mathsf{r}^d$.

We provide the details of our dot product with truncation protocol which uses techniques from [64]. Note that, although Mazloom *et al.* [63] achieve truncation at no additional overhead, their techniques are customised for a single corruption.

At a high level, their protocol relies on partitioning the four parties into 2 pairs such that a value is additively shared among the pairs, which is insecure in the FaF model due to the presence of the additional semi-honest party. Our dot product protocol with truncation, thus proceeds as follows.

Given an $(\mathsf{r}, \mathsf{r}^d)$ pair, where r^d represents the value of r, right-shifted by d bit position, the truncated value of v can be obtained by computing $(\mathsf{v} - \mathsf{r})^d + \mathsf{r}^d$. Note that d is the number of bits allocated for the fractional part of FPA. The correctness and accuracy of this technique was given in [64]. We use the same technique to provide a dot product protocol with truncation as described below.

Specifically, given the $[\![\cdot]\!]$-sharing of vectors $\vec{\mathbf{x}}$ and $\vec{\mathbf{y}}$, protocol DotPTr (Fig. 43) allows servers to generate $[\![z^d]\!]$ robustly, where z^d is the truncated value of $z = \vec{\mathbf{x}} \odot \vec{\mathbf{y}}$ and $\odot$ represents the dot product operation. $[\![\cdot]\!]$-sharing of a vector $\vec{\mathbf{x}}$ of size n, means that each element $\mathbf{x}_i \in \mathbb{Z}_{2^\lambda}$ of $\vec{\mathbf{x}}$, for $i \in [\mathsf{n}]$, is $[\![\cdot]\!]$-shared.

During the preprocessing phase, servers compute the $(\langle \mathsf{r} \rangle, [\![\mathsf{r}^d]\!])$-sharing of a random value $\mathsf{r} \in \mathbb{Z}_{2^\ell}$, by first generating the Boolean sharing of λ random bits $\mathsf{r}_0, \ldots, \mathsf{r}_{\lambda-1} \in \mathbb{Z}_{2^1}$ and obtaining their corresponding sharing over $\mathbb{Z}_{2^\ell}$ by invoking Bit2A. Following this, servers locally compute the $[\![\cdot]\!]$-sharing of r and r^d, using the fact that $\mathsf{r} = \sum_{s=0}^{\lambda-1} 2^s \mathsf{r}_s$ and $\mathsf{r}^d = \sum_{s=d}^{\lambda-1} 2^{s-d} \mathsf{r}_s$. Note that the $[\![\mathsf{r}]\!]$ can be locally converted by the servers to $\langle \mathsf{r} \rangle$ sharing, by any one pair of servers, say P_i, P_j adding β_r to $\langle \alpha_\mathsf{r} \rangle_{ij}$.

Similar to the dot product protocol, we borrow ideas from BLAZE for obtaining an online communication cost *independent* of n and use jmp3 and jmp4 primitives to ensure either success or TP selection. The n-independent online phase is achieved by reconstructing the aggregated z value, masked with r, which is reconstructed towards all the servers. Following this, servers can locally truncate $z + \mathsf{r}$ and compute its $[\![\cdot]\!]$-sharing, to finally obtain $[\![z^d]\!] = [\![(z + \mathsf{r})^d]\!] - [\![\mathsf{r}^d]\!]$ locally using $[\![\mathsf{r}]\!]$ generated during the preprocessing phase.

Protocol DotPTr($\mathcal{P}, ([\![\mathsf{x}^s]\!], [\![\mathsf{y}^s]\!])_{s \in [\mathsf{n}]}$)

- **Input and Output:** The input is $[\![\vec{\mathbf{x}}]\!] = \{[\![\mathsf{x}^s]\!]\}_{s \in [\mathsf{n}]}, [\![\vec{\mathbf{y}}]\!] = \{[\![\mathsf{y}^s]\!]\}_{s \in [\mathsf{n}]}$. The output is $[\![z^d]\!] = [\![(\vec{\mathbf{x}} \odot \vec{\mathbf{y}})^d]\!]$.
- **Primitives:** Protocol tripGen (§5.2; Fig. 6), $\langle\cdot\rangle$-Rec (§4.2; Fig. 3) and Bit2A (§F.6;Fig. 41).

Preprocessing:

– Servers compute $[\![\mathsf{r}_0]\!]^{\mathbf{B}}, [\![\mathsf{r}_1]\!]^{\mathbf{B}}, \ldots, [\![\mathsf{r}_{\lambda-1}]\!]^{\mathbf{B}}$ for random $\mathsf{r}_0, \ldots, \mathsf{r}_{\lambda-1} \in \mathbb{Z}_{2^1}$ as follows:
  – P_i, P_j where $1 \leq i < j \leq 4$ sample random $\langle \mathsf{r}_s \rangle_{ij} \in \mathbb{Z}_{2^1}$ for $s \in \{0, \ldots, \lambda - 1\}$.
  – Servers set $\beta_{\mathsf{r}_s} = 0$, for $s \in \{0, \ldots, \lambda - 1\}$.

– Servers invoke Bit2A on each $[\![\mathsf{r}_s]\!]^{\mathbf{B}}$ and obtain $[\![\mathsf{r}_s]\!]$ for all $s \in \{0, \ldots, \lambda - 1\}$.

– Servers compute the $[\![\mathsf{r}]\!] = \sum_{s=0}^{\lambda-1} 2^s [\![\mathsf{r}_s]\!]$ and $[\![\mathsf{r}^d]\!] = \sum_{s=d}^{\lambda-1} 2^{s-d} [\![\mathsf{r}_s]\!]$.
– Servers locally compute the $\langle \mathsf{r} \rangle$ from $[\![\mathsf{r}]\!]$.
– Servers invoke $\mathsf{tripGen}(\mathcal{P}, \langle \alpha_{\mathsf{x}}^s \rangle, \langle \alpha_{\mathsf{y}}^s \rangle)$ and obtain $\langle \alpha_{\mathsf{x}}^s \alpha_{\mathsf{y}}^s \rangle$, for $s \in [\mathsf{n}]$.

Online:

– Each P_i, P_j for $1 \leq i < j \leq 4$ and $(i,j) \neq (1,2)$ compute $\langle z - \mathsf{r} \rangle_{ij}$ such that $\langle z - \mathsf{r} \rangle_{ij} = \sum_{s=1}^{\mathsf{n}} (-\beta_{\mathsf{x}}^s \langle \alpha_{\mathsf{y}}^s \rangle_{ij} - \beta_{\mathsf{y}}^s \langle \alpha_{\mathsf{x}}^s \rangle_{ij} + \langle \alpha_{\mathsf{x}}^s \alpha_{\mathsf{y}}^s \rangle_{ij}) - \langle \mathsf{r} \rangle_{ij}$.
– P_1, P_2 compute $\langle z - \mathsf{r} \rangle_{12} = \sum_{s=1}^{\mathsf{n}} (\beta_{\mathsf{x}}^s \beta_{\mathsf{y}}^s - \beta_{\mathsf{x}}^s \langle \alpha_{\mathsf{y}}^s \rangle_{12} - \beta_{\mathsf{y}}^s \langle \alpha_{\mathsf{x}}^s \rangle_{12} + \langle \alpha_{\mathsf{x}}^s \alpha_{\mathsf{y}}^s \rangle_{12}) - \langle \mathsf{r} \rangle_{12}$.
– Servers invoke $\langle \cdot \rangle$-Rec and obtain $z - \mathsf{r}$.
– Servers locally compute $(z - \mathsf{r})^d$ and further $[\![(z - \mathsf{r})^d]\!]$.
– Servers compute $[\![z^d]\!] = [\![(z - \mathsf{r})^d]\!] + [\![\mathsf{r}^d]\!]$

Fig. 43: Dot Product Protocol with Truncation

Lemma 26 (Communication). *Protocol* DotPTr *with feature size* n *requires a communication of* $4\mathsf{n} + 13\lambda$ *elements and* $6\mathsf{n} + 3\lambda$ *instances of* OPE*s in the preprocessing phase, whereas* 3 *rounds and a communication of* 7 *elements in the online phase.*

Proof. In the preprocessing phase, servers locally generate $[\![\cdot]\!]^{\mathbf{B}}$-sharing of $\mathsf{r}_0, \ldots, \mathsf{r}_{\lambda-1}$, which is non-interactive. Following this, servers invoke λ instances of Bit2A (Fig. 41), each of which requires a communication of 13 elements and 3 instances of OPEs (Lemma 23). Further, servers invoke tripGen for each $s \in [\mathsf{n}]$, which requires a communication of $4\mathsf{n}$ elements and $6\mathsf{n}$ instances of OPEs. In the online phase, servers compute the $\langle \cdot \rangle$-sharing of $z - \mathsf{r}$, which is non-interactive. This is followed by an invocation of the $\langle \cdot \rangle$-Rec (Fig. 3) for the aggregated $z - \mathsf{r}$, which requires three rounds and a communication cost of 7 elements (Lemma 12). Following this, servers truncate $z - \mathsf{r}$ and compute its $[\![\cdot]\!]$-sharing non-interactively. Similarly, the $[\![z^d]\!]$-sharing is obtained by servers without interaction.

G Liquidity Matching

Liquidity matching requires transferring funds across banks. In these systems, preserving privacy is essential even from honest third parties. Furthermore, aborting a computation is not acceptable since it fails transactions among the banks. Thus, it is crucial to have a system where the success of the computation is guaranteed even if some parties are malicious. The system should preserve privacy from the malicious party and any third party, even if it is honest. Since the FaF model captures this exact scenario, we provide a 4 server based FaF secure protocol for computation of liquidity matching. We follow [7] where the liquidity matching is performed using the gridlock algorithm. The gridlock algorithm essentially identifies a set of transactions which can be executed while ensuring that each bank has sufficient liquidity to process those transactions, that is while ensuring that each bank has a positive balance upon completion of the transactions. We adapt the algorithm given in [7] for the 4 server case. In particular, [7] views a transaction as a tuple consisting of the source bank identifier, the amount to be transferred, and the destination bank identifier. Here we consider the open source and open destination setting, where the source and destination component of the tuple of transactions is visible to all, while the transaction amount is hidden. In [7], this variant of the algorithm is referred to as sodoGR. It proceeds by selecting a set of transactions and computing the potential balance of each bank, where these transactions are to be processed. If the computed balance of all the banks is positive, then all the transactions are processed. If not, for some banks, if the potential balance is negative, then transactions cannot be processed. In the latter case, the algorithm considers a reduced set of transactions for processing by pruning the appropriate transactions as follows. For banks with negative potential balances, the last outgoing transactions are removed from the set of transactions being considered for processing. Note that this may affect the updated balance of some other banks whose incoming transactions get removed in this process. Thus the algorithm recomputes the potential balance of each bank with the updated list of transactions. It repeats the process until we reach a list of transactions for which the updated balances of all the banks are positive, or no transaction can be processed. If the latter occurs, then we reach a deadlock. Below we provide the details of the secure evaluation of sodoGR. It takes a list Q of m transactions where a transaction is tuple (s, a, d) along with a bit x for every transaction, s is the source bank, d is the destination bank, and a is the amount of the transaction. Here $[\![a]\!]$ is secret-shared among the four servers. The bit x denotes if a transaction is considered in the computation or not. The bit x is also secret shared. Initially, it is set to 1, which means all the transactions are considered in the computation. Furthermore, the protocol takes secret shares of the balance of n banks. $[\![B_i]\!]$ represents the secret shared value of the ith bank. Note that $(i, \cdot, \cdot)$ represents the list of transactions where ith bank is the sender. Similarly, $(\cdot, \cdot, i)$ represents the list of transactions where ith bank is the receiver. For a transaction t in the list $(i, \cdot, \cdot)$ or $(\cdot, \cdot, i)$, $[\![a]\!]_t$ and $[\![x]\!]_t$ denotes the secret sharing of the amount and the x bit of the transaction t.

Protocol sodoGR

– **Input, Output:** A set of m transactions $Q = \{s_j, [\![a_j]\!], d_j\}_{j\in[m]}$, a set of execute bits, one corresponding to each transaction $\{[\![x_j]\!]\}_{j\in[m]}$ and a set of balances, one corresponding to each of the n banks $B = \{[\![B_i]\!]\}_{i\in[n]}$. It outputs a subset T of Q which can be processed and a boolean value deadLock which is 1 if and only if T is empty.

– **primitives:** addition, multiplication, comparison.

REPEAT

Update Balance: For all $i \in [n]$:
- $[\![S_i]\!] = \sum_{t=(i,\cdot,\cdot)} \mathsf{mult}([\![a]\!]_t, [\![x]\!]_t)$
- $[\![R_i]\!] = \sum_{t=(\cdot,\cdot,i)} \mathsf{mult}([\![a]\!]_t, [\![x]\!]_t)$
- $[\![UB_i]\!] = [\![B_i]\!] - [\![S_i]\!] + [\![R_i]\!]$

Check Balance: For all $i \in [n]$: $[\![h_i]\!] = 1 - \mathsf{msb}([\![UB_i]\!])$

Allowed List:
- $[\![z]\!] = \mathsf{mult}_{i\in[n]}([\![h_i]\!])$
- $output(z)$
- If $z = 1$ then output $T = \{t \in Q : x_t = 1\}$ and $\mathsf{deadLock} = 0$
- Else for all $i \in [n]$: $\{t_1, \cdots, t_v\} = (i, \cdot, \cdot)$ ordered in time of receipt order.
  - For all $j \in [v-1]$: $[\![x_{t_j}]\!] = \mathsf{mult}(\mathsf{mult}([\![x_{t_j}]\!], [\![x_{t_{j+1}}]\!]), [\![h_i]\!]) + \mathsf{mult}([\![x_{t_j}]\!], (1 - [\![h_i]\!])$
  - $[\![x_{t_v}]\!] = \mathsf{mult}([\![x_{t_v}]\!], (1 - [\![h_i]\!])$
- $[\![\mathsf{deadLock}]\!] = \mathsf{mult}_{j\in[m]}(1 - [\![x_j]\!])$

UNTIL $\mathsf{deadLock} = 1$

Output $\mathsf{deadLock}$ and $T = \phi$

Fig. 44: Secure evaluation of open source open destination GridLock Resolution

It can be observed that we need basic primitives such as addition, multiplication, and comparison to evaluate the above algorithm securely. In §5, we have discussed how to perform addition and multiplication securely. For comparison between two values $\mathbf{x}, \mathbf{y}$, we compute $\mathbf{x} - \mathbf{y}$ and check its most significant bit (msb) as described in §7. If the msb is 1, then it implies that $\mathbf{x} - \mathbf{y}$ is negative and hence $\mathbf{x} < \mathbf{y}$. Otherwise, when $\mathsf{msb} = 0$, we can conclude that $\mathbf{x} \geq \mathbf{y}$.

As evident from Fig. 44, input-dependent choices are made during the run of the protocol to optimize the overall complexity of the protocol. Incorporating this optimization requires input-dependent preprocessing and hence an all online protocol. Despite this, the reported overall run time showcases the practicality of using our `FaF` secure protocols.

H Additional Benchmarks

Table 9 details the average bandwidth and rtt between each pair of machines used in our experiments as measured by the `iperf` and `irtt` programs respectively.

As discussed in Section 7, computing summands of S_0 which involves running six instances of disMult is the communication bottleneck in the preprocessing phase of QuadSquad while computing summands of S_1 which involves running four instances of disZK is the computational bottleneck. Our implementation uses 6 threads to run each instance of disMult in a separate thread to parallelize communication while most of the remaining threads (23 out of a total of 32 threads) are used for computation in the instances of disZK. We microbenchmark

	$M_0 - M_1$	$M_0 - M_2$	$M_0 - M_3$	$M_1 - M_2$	$M_1 - M_3$	$M_2 - M_3$
Bandwidth (Mbps)	140	242	68	390	144	98
rtt (ms)	152.1	91.7	301	59.07	144.5	201.5

Table 9: Average bandwidth and round-trip time (rtt) between each pair of machines.

Number of triples	Computing summands of S_0 (s)	Computing summands of S_1 (s)
2^{17}	20.94	17.89
2^{18}	37.75	32.94
2^{19}	70.2	57.39
2^{20}	129.65	113.43

Table 10: Comparison of latency for computing summands of S_0 and S_1 when preprocessing different number of triples.

the performance of computing summands of S_0 and S_1 for preprocessing triples and summarize the results in Table 10. We observe that computing summands of S_0 tends to have higher latency than computing summands of S_1 and the difference in latency increases with the number of triples. This strongly suggests that the instances of disMult are the bottleneck in our implementation.