PentaGOD: Stepping beyond Traditional GOD with Five Parties

Nishat Koti, Varsha Bhat Kukkala, Arpita Patra,
Bhavish Raj Gopal
kotis,varshak,arpita@iisc.ac.in
gbhavish@gmail.com
Indian Institute of Science
Bangalore, India

ABSTRACT

Secure multiparty computation (MPC) is increasingly being used to address privacy issues in various applications. The recent work of Alon et al. (CRYPTO'20) identified the shortcomings of traditional MPC and defined a Friends-and-Foes (FaF) security notion to address the same. We showcase the need for FaF security in real-world applications such as dark pools. This subsequently necessitates designing concretely efficient FaF-secure protocols. Towards this, keeping efficiency at the center stage, we design ring-based FaF-secure MPC protocols in the small-party honest-majority setting. Specifically, we provide (1,1)-FaF secure 5 party computation protocols (5PC) that consider one malicious and one semi-honest corruption and constitutes the optimal setting for attaining honest-majority. At the heart of it lies the multiplication protocol that requires a single round of communication with 8 ring elements (amortized). To facilitate having FaF-secure variants for several applications, we design a variety of building blocks optimized for our FaF setting. The practicality of the designed (1,1)-FaF secure 5PC framework is showcased by benchmarking dark pools. In the process, we also improve the efficiency and security of the dark pool protocols over the existing traditionally secure ones. This improvement is witnessed as a gain of up to 62× in throughput compared to the existing ones. Finally, to demonstrate the versatility of our framework, we also benchmark popular deep neural networks.

KEYWORDS

multi-party computation; friends-and-foes security (FaF); honest majority; dark pools; PPML

1 INTRODUCTION

With the steady incline in the awareness of data privacy, we are witnessing a paradigm shift in healthcare, finance, and various other sectors involved in processing a large amount of sensitive client data. Various privacy-preserving practices are being adopted to reassure clients and provide them with the highest level of security guarantees. Given the ease of accommodating multiple data owners and its computational efficiency, many real-world applications prefer the use of secure multiparty computation (MPC) to perform privacy-preserving computations [7–9, 55]. Informally, MPC enables n mutually distrusting parties to compute a function over their private inputs, while ensuring the privacy of the same against an adversary controlling up to t parties. Various application scenarios where MPC based solutions have been proposed include– secure auctions [9], privacy-preserving machine learning [13, 18, 19, 24, 36, 37, 43, 45, 56, 63], secure recommendation systems [35, 59] and real-world deployments such as the Estonian study on correlation between tax data and educational records [8] and the study of salary inequities across various employees in the city of Boston [7]. Although MPC has been the apt solution for addressing privacy issues in various real-world problems, we expose the inadequacy of traditional security offered by MPC for applications with highly sensitive data. We use the example of financial trading forums such as dark pools to demonstrate the same.

The need for privacy naturally extends to processing financial transaction data. This has been one of the primary reasons for the emergence of *dark pools* that allow investors to trade (buy and/or sell) financial instruments such as securities (stocks, bonds etc.) outside of the prying eyes of the public and ensure the trade remains unexposed until it is completed. This allows investors to trade *large* blocks of securities privately and ensure the market is not impacted by the knowledge of such potential large-scale trade. For example, public knowledge of an institution trying to sell a large portion of its shares would cause a sudden depreciation of its share value even before the transaction is completed. On the other hand, the market impact is known to be much smaller when the trade is reported after it is executed. This is the working principle underlying dark pools, which makes them a popular choice for trading. Dark pools are traditionally operated by trusted brokers who are made aware of the trade interests of the clients. They are then expected to find matching counter-parties within their network of private clients. The clients, in the process, place complete trust on the broker to not misuse the trade interests disclosed on clear. However, several instances have showcased misuse of insider information where dark pool operators have been fined for the same [47–53].

To guarantee complete privacy, the interest to trade must never be disclosed in the open, not even to the broker operating the dark pool. Ideally, matches between sellers and buyers must be found without disclosing this sensitive information. Thus, the problem can be modeled as an instance of MPC, where the private input is the data related to the trade, and the clients are interested in securely matching the possible trades. In this setting, rather than the dark pool being operated by a central trusted broker, it is emulated by an MPC protocol run among a set of parties. Clients *secret share* their trade data to these parties in such a way that no subset, of at most t of these parties, learns any information. These parties are responsible for running the MPC protocol designed to identify matching trades securely. The applicability of MPC for securely operating dark pools has been shown previously [5, 16, 17, 23]. Although MPC is befitting to the addressed problem, the current solutions are far from complete. All the proposed protocols only offer malicious security with abort. That is, the protocols are designed to abort if the malicious adversary misbehaves and it is possible that

This is the full version of the paper to appear at ACM CCS 2022.

the adversary alone obtains the output. This could cause denial of service attacks and result in the protocol terminating even before the matched trades are disclosed. Further, such a setting allows an adversary to cause repeated failures. Since time is of essence in applications such as dark pools, this not only results in wastage of valuable compute resources, but may also hamper the functionality of the system. Hence, any security notion that empowers the adversary to abort does not fit the bill. Instead, a security notion that guarantees delivery of output regardless of the adversary's misbehaviour is desirable. This is achieved by *guaranteed output delivery* (GOD) or *robustness*, which is the strongest security notion that an MPC protocol offers. Hence it is imperative to realize robust, secure dark pools. The presence of GOD uplifts the trust and encourages client participation in the system.

It is well known that an honest majority among the parties is necessary to achieve GOD [21]. Moreover, honest majority enables designing efficient protocols in comparison to dishonest majority. Further, honest-majority MPC for a small set of parties has witnessed huge interest lately [2, 3, 10, 11, 13, 15, 18, 19, 29, 44, 46, 56]. This is due to the various customizations it allows resulting in huge efficiency gains. Hence, to realize applications such as secure dark pools, we focus on designing honest-majority MPC protocols with small number of parties that provide the strongest security of GOD.

Traditional GOD does not suffice. Most GOD protocols in the literature [11, 12, 15, 36] rely on an *honest* party identified as the trusted third party (TTP) to carry out the computation if misbehavior is detected. Elaborately, the parties entrust the TTP with their inputs, which carries out the computation and delivers the output to all. According to the standard security definition, this leakage of inputs towards a TTP is not considered a privacy breach. This is because the TTP is deemed to be honest and the goal is to protect against information leakage towards an adversary. However, entrusting a TTP with all the inputs may not be acceptable in real-world applications. Specifically, in the case of a dark pool, this is equivalent to having a central broker who learns all the inputs and is trusted to perform the matching. This defeats the purpose of employing an MPC protocol, as one of the goals of a secure dark pool system is to hide the trade from *every single party* since it contains highly sensitive information of the client.

Another drawback of traditional MPC protocols is the view leakage attack. While executing an MPC protocol, nothing prevents an adversary from sending its view, which consists of the view of t corrupt parties, to an honest party. This is not treated as an attack in the traditional security definition, since an honest party is expected to discard non-protocol messages, unlike a semi-honest one[1]. However, if this honest party turns rogue in the future, the party can obtain all the information about the submitted trade requests in the system. This holds because it would now possess information with respect to $t+1$ parties (t views received from the adversary and its own view), which suffices to obtain the underlying secret information. This, too, goes against the goal of providing trade secrecy expected in the system.

To address these drawbacks of the traditional MPC security definition, Alon et. al. [1] proposes a new definition, called MPC with Friends and Foes (FaF). This definition requires honest parties' inputs to be protected against not only the adversary (foes), but also from quorums of other honest parties (friends). This is modeled by a decentralized adversary which comprises two different *non-colluding* adversaries– (i) a malicious adversary that corrupts any subset of at most t out of n parties, (ii) a semi-honest adversary that corrupts any subset of at most $h^\star$ out of the remaining $n-t$ parties. A protocol secure against such an adversary is said to be $(t, h^\star)$-FaF secure. Further, the FaF model requires security to hold even when an adversary sends its view to other parties, thereby closely modeling our need. Hence, departing from the traditional MPC model, we identify the need to design FaF-secure MPC protocols for applications such as dark pools that deal with highly sensitive financial information that needs protection from all forms of misuse. As described earlier, existing lawsuits against dark pool operators showcase the temptation to misuse profitable information, thereby reasserting the need for stepping beyond traditional security. Further, our protocols also secure in the mixed adversarial model[2], as described in §E.3.

Small-party honest-majority FaF model. Alon et. al. [1] show that GOD can be achieved in the $(t, h^\star)$-FaF model iff $2t + h^\star < n$. Thus, obtaining GOD requires $n \geq 4$ for non-zero values of t and $h^\star$. Since our focus is on MPC with small number of parties, observe that instantiating $n = 4$ and $t = h^\star = 1$ provides the optimal threshold for 4 party computation (4PC) to achieve GOD. However, two corruptions result in a dishonest majority setting, which renders less efficient protocols than their honest majority counterparts. Hence, to design efficient protocols, we augment this setting with one additional honest party, and design 5PC protocols which are $(1,1)$-FaF secure. We remark that while $(t, h^\star)$ can be instantiated with a varied range of values to attain GOD such that $2t + h^\star < 5$, we set $t = h^\star = 1$ because of the following reasons: (a) this results in an honest majority setting; (b) we believe that $h^\star = 1$ suffices for most practical applications since honest parties (friends) are unlikely to collude with each other (note that when $h^\star = 1$, the only possible value of t is 1). We note that in the current setting of $n = 5$ and $t = h^\star = 1$, one could alternatively avoid the aforementioned weaknesses by deploying a traditional (5,2) malicious secure protocol since the latter protects against view leakage and also avoids reliance on a TTP when deployed in the presence of a single malicious party. However, since the traditional protocol is designed to cater to two malicious parties as opposed to one in our setting, it may lose out on performance. Hence, keeping efficiency for real-world applications at center stage, the objective is to leverage the presence of a semi-honest party to design customised efficient $(1,1)$-FaF secure protocols. We note that a traditional (n, t) malicious protocol is capable of protecting against view leakage attack and avoids the reliance on a TTP as long as at most $t-1$ parties are malicious.

1.1 Our contributions

$(1,1)$-FaF secure 5PC. We observe that traditionally secure MPC providing GOD is a misfit for several sensitive real-world applications. This necessitates designing GOD protocols in the FaF-secure

[1] A semi-honest adversary follows the protocol specification but always tries to learn more information that it is not entitled to.

[2] Mixed adversarial model is one where a single (centralized) adversary is allowed to corrupt t parties maliciously and a disjoint subset of $h^\star$ parties semi-honestly.

model. Towards this, with efficiency in mind, we work over the ring $\mathbb{Z}_{2^\ell}$, both arithmetic and Boolean ($\ell = 1$), and design (1,1)-FaF secure 5PC protocols. The protocols are cast in the preprocessing model since it offloads heavy input-independent computations to a preprocessing phase, resulting in a fast input-dependent online phase. The highlight here is the multiplication which requires–(i) *just three* parties to be online for most of the computation and (ii) requires one round (amortized) and eight ring elements of communication in the online phase. The efficiency and resource management (involvement of only 3 parties for most of the computation) of the multiplication results in a concretely efficient 5PC framework. We concretely showcase the benefit of having reduced number of online parties over a naive solution (all parties online) as well as the traditional $(5, 2)$ maliciously secure protocol.

Building blocks and generality. We resort to a modular approach to design various building blocks, as shown in Fig. 1, where protocols in each layer build on those in the previous layers. Layer 0 forms the core MPC, with layers above it providing the building blocks. This constitutes our generic and comprehensive framework since it provides support for a wide range of building blocks that suffice for various applications. While these building blocks have been well studied in the literature, our contribution lies in designing and optimizing these for the 5PC (1,1)-FaF setting.

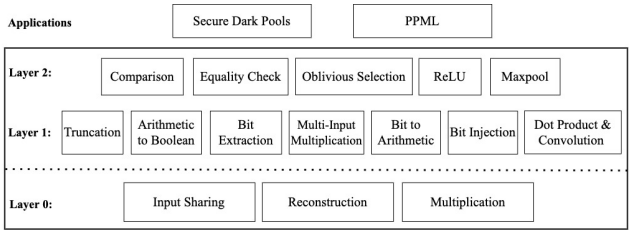


Figure 1: Designed (1, 1) FaF-secure 5PC framework

Applications and Benchmarks. The designed (1,1)-FaF secure 5PC is ideal for real-world applications that deal with highly sensitive information such as systems with financial transaction data [58], biometric data [62], allegations reported by victims or whistle-blowers [4, 39], personal health record data [27], etc. We consider two such applications to showcase the practicality of our framework.

(i) *Dark pools:* Although secure dark pools have been considered in the traditional MPC setting, we design improved protocols for the same in the 5PC (1,1)-FaF setting. Specifically, we optimize the continuous double auction (CDA) and volume-based matching algorithms. We identify several aspects of the matching algorithms that can be performed in parallel, which improves the efficiency of the designed protocols. We benchmark the performance of these secure matching algorithms and observe a throughput improvement of up to 62× in comparison to [16].

(ii) *PPML:* The designed building blocks have been extensively used in realizing privacy-preserving machine learning (PPML) [13, 18, 19, 36, 43, 45, 56], albeit in the traditional security model. Since PPML in itself is suitable for a wide range of application scenarios, we also demonstrate the practicality of the designed (1,1)-FaF secure 5PC for PPML. For this, we benchmark the performance of the designed protocols for secure inference using popular deep neural networks such as LeNet [40] and VGG16 [60].

1.2 Related Work

The work of [1] focuses on extending the standard security notion of MPC to the FaF-setting. In this regard, they provide both a full-security as well as fairness variants in this new setting. They further provide a detailed investigation of various feasibility results and limitations in the FaF-setting. The (1,1)-FaF secure 5PC protocol designed in the current work forms the first concrete instantiation of a FaF-secure protocol, particularly as the optimal case for an honest-majority setting for a small number of parties. We therefore next discuss relevant secret sharing based MPC works that provide GOD in small-party setting under the traditional security model. A concretely efficient protocol for achieving GOD was provided in [36], both for 3PC and 4PC setting, which improved over the 4PC of [13] and the 3PC of [11]. Note that [13] in turn improved upon the GOD protocols in [32]. The work in [24] proposed 4PC protocols on par with [36], albeit with security of *private robustness*. However, the security guarantees of both SWIFT and [24] are known to be theoretically equivalent. The recent work of [37] provides an improved multiplication protocol over [36] in the 4PC setting. The improvement is seen in the preprocessing phase, where [37] requires only 2 ring elements as opposed to 3. While there are no protocols explicitly designed for 5PC that attain GOD, [12] provides protocols for the n-party setting, from which a 5PC protocol can be derived. The work of [14] attains GOD in the 5PC setting, albeit relying on garbled circuits.

Organization. We describe the preliminaries and threat model in §2. The core (1, 1) FaF-secure 5PC is explained in §3. The building blocks follow in §4. Finally, the practicality of our framework is demonstrated through benchmarks for dark pools and PPML, together with improved dark pool algorithms, in §5.

2 PRELIMINARIES

Threat model. We design protocols that comprise five parties $\mathcal{P} = \{P_1, P_2, \ldots, P_5\}$ that are connected via pairwise private and authentic channels in a synchronous network. Our protocols are FaF-secure with a static, malicious probabilistic polynomial time (PPT) adversary that can corrupt up to one party, and a *different* semi-honest adversary that can corrupt at most one other party. We prove the security of our protocols in the standard real-world/ideal-world paradigm. The security definition as per this paradigm in FaF model is recalled in Appendix §A.

All our primitives assume a one-time shared key setup to facilitate each subset of parties to generate common randomness among themselves. We model this as an ideal functionality $\mathcal{F}_{\mathsf{setup}}$ (Fig. 8), which can be instantiated with any FaF protocol in our setting (say using that of [1]). Several works [2, 3, 11, 13, 18, 19, 36, 43, 56] rely on such a setup. Therefore, the high-level protocols, either application protocols or general 5PC, start with such a setup phase which is done once and for all. The set of computing parties $\mathcal{P}$ may be equivalently represented as $\mathcal{P} = \{P_i, P_j, P_k, P_l, P_m\}$ for ease of presentation. Our protocol works over the ring $\mathbb{Z}_{2^\ell}$ (and $\mathbb{Z}_2$). We use fixed-point arithmetic (FPA) notation to represent decimal values, in signed 2's complement notation. Here, the most significant bit denotes the sign, the last d bits represent the fractional part, and the remaining $\ell - d - 1$ bits denote the integer part. We let $\ell = 64$ and $d = 13$.

3 ROBUST $(1,1)$-FAF SECURE 5PC

Our protocols are cast in the preprocessing model, where during an input-independent preprocessing phase, computationally heavy operations are carried out which pave way for a fast input-dependent online phase. Our online phase requires active participation mostly from only three parties, P_1, P_2, P_3, while P_4, P_5 come online only for a short while, towards the end the computation, for verification.

All the building blocks, except reconstruction which is robust, follow a common paradigm: either the protocol is successful or the protocol finds a conflicting pair of parties, CP, that includes the malicious party; in the latter case, the input shares amongst 5 parties are reshared amongst the 3 parties outside the conflict set, without affecting the secrets, and the computation is rerun amongst the 3 parties. Since the malicious party is already excluded, the 3-party protocol needs to be only semi-honest secure tolerating one corruption. To enable this paradigm, we employ a share-conversion primitive (see §3.5), which reshares the state of the 5 parties to the 3 parties, to continue computation with the latter. In any application protocol such as the secure matching, as soon as a CP is detected as a part of the computation, the rest of the execution switches to the 3-party computation (3PC). Note that any semi-honest 3PC framework that respects the secret-sharing semantics (replicated secret sharing) can be deployed (such as the 3PC of [3, 18]), and hence we treat the 3PC as a black-box.

The above paradigm of identifying a CP and completing the computation in a smaller subset of parties is mostly facilitated by a message-passing primitive invoked in the sharing and multiplication protocols (which in turn form the basis of all the building blocks). We introduce this primitive below, followed by our secret-sharing semantics. We then describe the protocols for input sharing, multiplication – which forms the core of all our constructions, followed by output reconstruction.

*Joint message passing (*jmp*).* This primitive enables two parties to send a common message to a third party such that the recipient either receives the correct message or in case of an inconsistency in the received messages, a trusted third party (TTP) is identified [36]. The protocol involves one sender sending the value, while other sending the hash to the receiver, who then compares the received values; in case of an inconsistency, the parties proceed to identify a TTP, who then completes the computation of MPC on the clear after receiving inputs from the parties. As opposed to the protocol of SWIFT [36], we cannot use TTP in the same way in the $(1,1)$-FaF setting as the TTP learns all the inputs. Thus, we modify the jmp protocol in [36] to adapt it to the $(1,1)$-FaF setting as follows– in case of an inconsistency, we modify jmp to output a pair of parties in conflict, one of which is guaranteed to be maliciously corrupt, instead of identifying a TTP. Note that the jmp protocol consists of two phases (*send, verify*). The *send* phase consists of one of the two senders, denoted as the *speaker* party sending the message to the receiver, while the other sender party referred as the *silent* party keeps quite. This distinction between the *speaker* and a *silent* party is made only for the *send* phase. *Verify* phase comprises all the other steps of the jmp protocol, and either confirms that message delivery to the recipient was a success or identifies a conflict pair, CP. Looking ahead, our protocols rely on several invocations of jmp. Hence, to leverage amortization, in most cases, the *send* phase is executed on the flow, and *verify* phase is deferred to a later stage. This deferring of *verify* brings significant challenges in our protocols such as multiplication. A part of our novelty comes from handling these challenges. Fig. 9 details the modified (Π_{jmp}) protocol.

We say P_i, P_j jmp-send msg to P_k when they invoke only the *send* phase of $\Pi_{\mathsf{jmp}}(P_i, P_j, P_k, \mathsf{msg})$. Without loss of generality, we let P_i be the *speaker* and P_j be the *silent* party. Since verification can be deferred, we say that P_i, P_j jmp-vrfy towards P_k when they invoke only the deferred *verify* phase corresponding to $\Pi_{\mathsf{jmp}}(P_i, P_j, P_k, \mathsf{msg})$. Finally, we say P_i, P_j jmp-sv msg to P_k when they invoke the complete $\Pi_{\mathsf{jmp}}(P_i, P_j, P_k, \mathsf{msg})$ protocol and execute both the *send* and *verify* phases together. Note that, the *verify* phase (both deferred and in-place) was described with respect to a single message. However, the details of improving the amortization (in both cases) by bundling together the *verify* across several messages for a fixed ordered pair of senders and a given receiver is described in §B.1.

3.1 Secret sharing semantics

In the 5PC $(1,1)$-FaF setting, a (semi-honest) adversary may be entitled to the view of at most two parties (itself and the malicious party). Thus, to ensure that the view of two parties does not leak any additional information, we rely on a $(5,2)$ replicated secret sharing (RSS) scheme and its variants. A value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ is said to be RSS-shared among 5 parties with threshold 2 if for every subset of two parties, say $\{P_i, P_j\}$, the residual three parties hold share $\mathsf{v}_{ij} \in \mathbb{Z}_{2^\ell}$ such that $\mathsf{v} = \sum_{1 \le i < j \le 5} \mathsf{v}_{ij}$. Observe that since any set of two parties in $\mathcal{P}$ always miss one share of v, they cannot reconstruct the value, whereas any three parties can. The total number of shares of a value is thus $\binom{5}{2} = 10$ and the RSS-share possessed by $P_s \in \mathcal{P}$ is a tuple of $\binom{4}{2} = 6$ shares v_{ij} where $s \neq i, s \neq j$ and $1 \le i < j \le 5$. With this background, we define our sharing semantics below.

$[\mathsf{v}]$ denotes a value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ is $[\cdot]$-shared among parties in $\mathcal{P}$ if it is $(5,2)$ RSS-shared among them. We let $[\mathsf{v}]_s$ denote P_s's $[\cdot]$-shares of v. Note that $[\mathsf{v}]_s$ is a tuple of 6 elements and $[\mathsf{v}]$ is a tuple of 10 elements. $\langle \mathsf{v} \rangle$ denotes a value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ is $\langle \cdot \rangle$-shared among parties in $\mathcal{P}$ if there exists $\mathsf{v}_1, \mathsf{v}_2, \mathsf{v}_3, \mathsf{v}_4, \mathsf{v}_5 \in \mathbb{Z}_{2^\ell}$ such that $\mathsf{v} = \mathsf{v}_1 + \mathsf{v}_2 + \mathsf{v}_3 + \mathsf{v}_4 + \mathsf{v}_5$ and $P_s \in \mathcal{P}$ possess v_s. Let $\langle \mathsf{v} \rangle = (\mathsf{v}_1, \mathsf{v}_2, \mathsf{v}_3, \mathsf{v}_4, \mathsf{v}_5)$. $[\![\mathsf{v}]\!]$ denotes value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ is $[\![\cdot]\!]$-shared among $\mathcal{P}$, if–(i) there exists $\alpha_\mathsf{v} \in \mathbb{Z}_{2^\ell}$ that is $[\cdot]$-shared among parties in $\mathcal{P}$, and (ii) there exists $\beta_\mathsf{v} \in \mathbb{Z}_{2^\ell}$ such that $\beta_\mathsf{v} = \mathsf{v} + \alpha_\mathsf{v}$ is held by all parties in $\mathcal{P}$.

For a set of n values $\{\mathsf{v}_1, \ldots \mathsf{v}_n\}$, we let $\beta_{\mathsf{v}_1 \ldots \mathsf{v}_n} = \Pi_{i=1}^{n} \beta_{\mathsf{v}_i}$ and $\alpha_{\mathsf{v}_1 \ldots \mathsf{v}_n} = \Pi_{i=1}^{n} \alpha_{\mathsf{v}_i}$. We use the superscript $\mathbf{B}$ to denote the Boolean sharing over $\mathbb{Z}_2$, while the absence of it implies arithmetic sharing over $\mathbb{Z}_{2^\ell}$. All the above sharing schemes are linear, i.e., given shares of $\mathsf{v}_1, \mathsf{v}_2$, and public constants c_1, c_2, parties can locally compute the shares of $c_1\mathsf{v}_1 + c_2\mathsf{v}_2$.

Conversion between $[\![\cdot]\!]$ and $[\cdot]$ shares during preprocessing. Given $[\mathsf{v}]$, protocol $\Pi_{[\cdot] \to [\![\cdot]\!]}$ generates $[\![\mathsf{v}]\!]$ from it by setting $\beta_\mathsf{v} = 0$ and $[\alpha_\mathsf{v}] = -[\mathsf{v}]$. Conversely, $\Pi_{[\![\cdot]\!] \to [\cdot]}$ generates $[\mathsf{v}]$ from $[\![\mathsf{v}]\!]$. For this, parties set $\mathsf{v}_{12} = \beta_\mathsf{v} - \alpha_{\mathsf{v}12}$ while $\mathsf{v}_{ij} = -\alpha_{\mathsf{v}ij}$ for $1 \le i < j \le 5, (i,j) \neq (1,2)$.

3.2 Input sharing

Protocol Π_{sh} enables $P_i \in \mathcal{P}$ holding a value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ to generate $[\![\mathsf{v}]\!]$. For this, parties generate $[\cdot]$-shares of a random value $\alpha_\mathsf{v} \in \mathbb{Z}_{2^\ell}$ in the preprocessing phase, non-interactively, using their shared

key setup such that the dealer P_i learns all the $[\cdot]$-shares of α_v. This enables P_i to compute $\beta_v = v + \alpha_v$ in the online phase and jmp-sv it to all the parties. The protocol appears in Fig. 10. The protocol for generating $[\cdot]$-shares of $v \in \mathbb{Z}_{2^\ell}$ is similar as above and formal details appear in Fig. 11. Protocol Π_{JSh2} is a variant of input sharing Π_{sh}, which enables two parties P_i, P_j to jointly generate $[\![\cdot]\!]$-shares of a value $v \in \mathbb{Z}_{2^\ell}$ known to both. Looking ahead, this protocol is heavily used in designing the building blocks, and is similar to Π_{sh}. Here, during the preprocessing phase parties generate $[\alpha_v]$ for $\alpha_v \in \mathbb{Z}_{2^\ell}$ such that P_i, P_j learn all its shares. Following this, P_i, P_j generate and jmp-send $\beta_v = v + \alpha_v$ towards all the other parties with its jmp-vrfy deferred. The protocol appears in Fig. 12 and other optimizations are deferred to §B.3.

3.3 Multiplication

The multiplication protocol Π_{mult} allows parties to compute $[\![z]\!] = [\![a \cdot b]\!]$, where $a, b \in \mathbb{Z}_{2^\ell}$ are $[\![\cdot]\!]$-shared. The highlight of our protocol is that it requires a single *online* round for evaluating a multiplication gate and requires active participation from only three parties for most of the computation. The protocol proceeds as follows. In the preprocessing phase, parties first generate $[\alpha_z] \in \mathbb{Z}_{2^\ell}$, non-interactively, using their shared key setup. To generate $[\![z]\!]$, parties need to compute β_z which can be written as follows: $\beta_z = z + \alpha_z = ab + \alpha_z = (\beta_a - \alpha_a)(\beta_b - \alpha_b) + \alpha_z = \beta_{ab} - \beta_a\alpha_b - \beta_b\alpha_a + \alpha_{ab} + \alpha_z$, where $\beta_{ab} = \beta_a\beta_b$ and $\alpha_{ab} = \alpha_a\alpha_b$. Observe that parties already possess β_a, β_b and $[\cdot]$-shares of $\alpha_a, \alpha_b, \alpha_z$. Assuming that $[\alpha_{ab}]$ is also made available, parties can compute $[\beta_z]$, leveraging the linearity of $[\cdot]$-sharing. We discuss how to generate $[\alpha_{ab}]$ in the preprocessing phase later and focus on the remaining steps assuming that $[\alpha_{ab}]$ is given. Now, β_z can be reconstructed towards all the parties, thereby generating $[\![z]\!]$. This reconstruction towards $P_i \in \mathcal{P}$ can be performed using just two invocations of Π_{jmp} as follows. The four shares missing at P_i, which include $\{\beta_{z_{ij}}, \beta_{z_{ik}}, \beta_{z_{il}}, \beta_{z_{im}}\}$ are sent to it as– P_j, P_k jmp-sv $\{\beta_{z_{il}} + \beta_{z_{im}}\}$ while P_l, P_m jmp-sv $\{\beta_{z_{ij}} + \beta_{z_{ik}}\}$.

Towards an efficient online phase. The above approach requires all parties to be online. However, observe that P_1, P_2, P_3 possess the required shares to compute the entire function. Hence, to reduce the number of active parties in the online phase, whenever multiplication is invoked, we restrict the reconstruction of β_z only towards the *online parties*, P_1, P_2, P_3 (but without the correctness guarantee), and defer reconstruction towards P_4, P_5 to a later point. Thus, only the jmp-send with respect to following 6 jmps are invoked– $\Pi_{\mathsf{jmp}}(P_2, P_4, P_1, \beta_{z_{13}} + \beta_{z_{15}})$, $\Pi_{\mathsf{jmp}}(P_3, P_5, P_1, \beta_{z_{12}} + \beta_{z_{14}})$, $\Pi_{\mathsf{jmp}}(P_1, P_4, P_2, \beta_{z_{23}} + \beta_{z_{25}})$, $\Pi_{\mathsf{jmp}}(P_3, P_5, P_2, \beta_{z_{12}} + \beta_{z_{24}})$, $\Pi_{\mathsf{jmp}}(P_1, P_4, P_3, \beta_{z_{23}} + \beta_{z_{35}})$, $\Pi_{\mathsf{jmp}}(P_2, P_5, P_3, \beta_{z_{13}} + \beta_{z_{34}})$. Recall that since only the *send* of jmp is performed, the *silent* parties, P_4, P_5, can remain offline. To complete the generation of $[\![z]\!]$, and enable P_4, P_5 to obtain β_z, we let P_1, P_2 jmp-sv β_z to P_4, P_5. We can defer this step until output reconstruction stage, where βs corresponding to all the invocations of multiplication until output reconstruction are sent in a *single* round. Deferring the send of β_z to after output reconstruction, may result in incorrectly reconstructing z. With this approach, evaluating multiplications requires participation from only the online parties (P_1, P_2, P_3) for most of the computation and *offline parties* (P_4, P_5) become active only before output reconstruction. Further, only a single round (owing to the *send* phase of jmp where only *speaker* party communicates) is needed for reconstructing each β among the online parties. Observe that the following two issues may arise while executing the above approach– (a) *correctness:* β_z reconstructed among online parties P_1, P_2, P_3 may be incorrect; (b) *agreement:* online parties may not be in agreement with respect to the β_z they hold, let alone hold the correct β_z. Both the issues arise since only the *send* phase of jmp is executed among the online parties while reconstructing β_z, which may lead to incorrect reconstruction among them. We next describe how both these issues can be addressed in the verification phase. Looking ahead, resolution for both issues either results in successfully completing the protocol, or identification of CP. In the latter case, parties switch to 3PC (after share conversion) for rest of the computation.

(a) Ensuring correctness. Correctness of the β_z reconstructed towards the online parties can be enforced by executing the jmp-vrfy towards them, and requires P_4, P_5. For this, P_4, P_5 should possess the correct inputs used for generating β_z, which may themselves be outputs, β_a, β_b, of multiplications. As mentioned earlier, P_4, P_5 receive all these βs in a single invocation of jmp-sv from P_1, P_2 just before output reconstruction. However, P_1, P_2 may not be in *agreement* with respect to these βs due to incorrect reconstruction of the same. Performing jmp when the senders are not in agreement with respect to the value being sent may result in incorrectly identifying a pair of honest parties as a CP (conflict pair). This necessitates a consistency check to ensure that P_1, P_2 are in agreement, and is discussed later. Hence, assuming P_1, P_2 are in agreement after this consistency check, they proceed to jmp-sv βs for all these multiplications to P_4, P_5. If this jmp-sv towards P_4, P_5 succeeds (i.e., no CP identified), verification of βs, reconstructed among the online parties, is performed. This is done by invoking deferred jmp-vrfy corresponding to all the jmp-send performed among the online parties. The success of all the *verify* phases guarantees the correctness of βs. In case if any *verify* fails, a CP is identified.

(b) Ensuring agreement. We now describe the consistency check mentioned above. In order to ensure agreement among online parties P_1, P_2, P_3, they exchange the hash of βs for all the multiplications among themselves. If these are consistent, then they proceed with the correctness check as described above. If the consistency check fails, the goal is to identify a CP. Observe that the check may fail due to one of the following reasons: (i) an incorrect β was reconstructed towards some honest online party which led to sending an incorrect hash during the check, or (ii) an incorrect hash was deliberately sent. Note that case (i) arises if a malicious online party misbehaved during a jmp-send performed at some level (layer in the circuit comprising addition and multiplication gates.) during circuit evaluation. Hence, performing the jmp-vrfy of this particular jmp-send can identify a CP and address case (i). A keen observer would note the circularity involved in addressing the agreement issue by relying on *verify* of jmp (to identify a CP). The circularity arises due to the following reason. jmp-vrfy towards P_1, P_2, P_3 requires P_4, P_5 to hold consistent β. Since P_4, P_5 receive the β via jmp-sv from P_1, P_2, it requires the latter to already be in agreement, and hence the circularity. To break the circularity, online parties rely on a binary search of levels within the circuit. The search identifies consecutive levels L_p, L_{p+1} such that all βs up to level L_p are consistently held among P_1, P_2, P_3 while L_{p+1} onwards

they are inconsistent. The consistency of βs up to L_p thus enables usage of jmp. For the binary search, parties exchange the hash with respect to βs in the first (top) half of the circuit, say max/2 levels (where max denotes the maximum levels in the circuit). If the hash is inconsistent, they recursively proceed with the first half (L_1 to $L_{\mathsf{max}/2}$), else if consistent, they proceed with the second half ($L_{\mathsf{max}/2+1}$ to L_{max}). In this way, they recursively operate on the appropriate half that has inconsistent hash to identify L_p, L_{p+1}. Note that one is guaranteed to identify such a L_p, L_{p+1} since the above recursion would terminate and at least at the first level in the circuit is guaranteed to be consistent and correct, owing to the correctness of the input sharing.

On identifying levels L_p and L_{p+1}, P_1, P_2 jmp-sv all the βs up to level L_p to P_4, P_5. This is followed by the deferred jmp-vrfy towards P_1, P_2, P_3 for all βs up to level L_{p+1}. If no CP is identified during any of the *verify* phases, it implies case (ii), and hence, honest online parties will be guaranteed to be in agreement with respect to the *correct* βs up to level L_{p+1}. Thus, the correct hash that should have been sent at level L_{p+1} in the binary search can be computed locally (using the β) and matched against the hash received from others. This determines the corrupt party that deliberately sent an incorrect hash. This corrupt party, together with another honest party, is identified as a CP. Note that the binary search can terminate with the last level being identified as L_p. This happens when circuit evaluation is correct, but malicious party deliberately sends an incorrect hash in consistency check and behaves honestly in binary search. L_p being the last level implies that honest parties are guaranteed to be in agreement with respect to *all* βs. Hence, corrupt party can be identified as the party who sent incorrect hash in the consistency check. Similar to above, the CP can thus be formed.

Generating $[\alpha_{\mathsf{ab}}]$. Since $[\alpha_{\mathsf{a}}], [\alpha_{\mathsf{b}}]$ are available in the preprocessing phase, $[\alpha_{\mathsf{ab}}]$ can be computed there. For obtaining $[\alpha_{\mathsf{ab}}]$, we rely on a robust $(1,1)$-FaF secure multiplication protocol for 5PC which works on $[\cdot]$-shares (RSS shares), and is abstracted out as a functionality, $\mathcal{F}_{\mathsf{MulPre}}$, in Fig. 34. To leverage amortization, we preprocess several multiplication triples in single shot. Hence, $\mathcal{F}_{\mathsf{MulPre}}$ is defined with respect to several triples. We instantiate $\mathcal{F}_{\mathsf{MulPre}}$ using a variant of the protocol of [12] for the 5-party setting. Similar to the original protocol, the modified protocol involves performing a 5PC semi-honest multiplication followed by a verification phase to check the correctness of the semi-honest execution. The difference lies in the steps performed when verification fails, and it outputs a pair of conflicting parties. In such a case, we eliminate the pair of parties, and the computation proceeds via semi-honest 3PC (since there is at most one malicious party in our case), unlike the malicious 3PC used in the original protocol. The verification phase relies on distributed zero-knowledge proof system [10], and is designed such that its communication cost gets amortized over multiple instances of multiplication. Thus, the amortized communication cost of this 5PC protocol is the same as that of the semi-honest protocol. The original protocol [12] is secure according to the standard definition of security. We prove that the modified variant, for 5PC, is secure in the $(1,1)$-FaF model. We refer readers to §F for details. Our multiplication protocol appears in Fig. 2.

To showcase all the cases handled and improve the readability of our algorithm we also provide a flowchart of the verification phase in Fig. 3. The green arrows denote the steps that lead to successful circuit evaluation and also showcase the correctness of our protocol. The flow where one of the offline parties is malicious is trivial and follows from the verify of jmp towards parties P_1, P_2, P_3 in the verification phase.

Protocol $\Pi_{\mathsf{mult}}(\mathcal{P}, [\![\mathsf{a}]\!], [\![\mathsf{b}]\!])$

Preprocessing: Non-interactively generate $[\cdot]$-shares of a random $\alpha_{\mathsf{z}} \in \mathbb{Z}_{2^\ell}$, using the shared-key setup. Invoke $\mathcal{F}_{\mathsf{MulPre}}$ on $[\alpha_{\mathsf{a}}], [\alpha_{\mathsf{b}}]$ (Fig. 34) to generate $[\alpha_{\mathsf{ab}}]$.

Online:

- Compute $[\beta'] = -\beta_{\mathsf{a}}[\alpha_{\mathsf{b}}] - \beta_{\mathsf{b}}[\alpha_{\mathsf{a}}] + [\alpha_{\mathsf{ab}}] + [\alpha_{\mathsf{z}}]$.
- Send missing $[\beta']$-shares to P_1, P_2, P_3: (a) P_2, P_5 jmp-send $\beta'_{12} + \beta'_{15}$ to P_1, while P_3, P_4 jmp-send $\beta'_{12} + \beta'_{15}$ to P_1, (b) P_1, P_5 jmp-send $\beta'_{23} + \beta'_{24}$ to P_2, while P_3, P_4 jmp-send $\beta'_{12} + \beta'_{25}$ to P_2, (c) P_1, P_4 jmp-send $\beta'_{23} + \beta'_{35}$ to P_3, while P_2, P_5 jmp-send $\beta'_{13} + \beta'_{34}$ to P_3.
- P_1, P_2, P_3 reconstruct β' and compute $\beta_{\mathsf{z}} = \beta' + \beta_{\mathsf{ab}}$.

One-time Verification (for entire circuit): Let M be the set of all β_{z}s where each z is the output of a multiplication in the circuit. Parties do the following.

- $P_i \in \{P_1, P_2, P_3\}$ computes the hash, $\mathsf{H}_i = \mathsf{H}\left(\beta_{\mathsf{z}_1}, \ldots, \beta_{\mathsf{z}_{|\mathsf{M}|}}\right)$ where $\beta_{\mathsf{z}_j} \in \mathsf{M}$ and mutually exchange it among themselves.
- $P_i \in \{P_1, P_2, P_3\}$ broadcasts an inconsistency bit b to indicate whether all the obtained hashes are consistent (b = 0) or not (b = 1).
- If all parties in $\{P_1, P_2, P_3\}$ broadcast b = 0, then–(a) P_1, P_2 jmp-sv all $\beta_{\mathsf{z}} \in \mathsf{M}$ to P_4, P_5. (b) If this jmp-sv succeed, (i.e., no CP is identified), then parties perform the deferred jmp-vrfy with with respect to all $\beta_{\mathsf{z}} \in \mathsf{M}$.
- Else, if some $P_i \in \{P_1, P_2, P_3\}$ broadcasts b = 1, then
  - Each $P_i \in \{P_1, P_2, P_3\}$ broadcasts H_i.
  - If for any $P_i \in \{P_1, P_2, P_3\}$, the hash sent via broadcast does not match the hash received on point-to-point communication by some party, P_j, then P_i broadcasts its complaint against P_j. Parties set $\mathsf{CP} = (P_i, P_j)$ where P_i is the party with the least index that complained, and terminate.
  - If a CP was not identified via a complaint, then
    - Let $\mathsf{H}_i^{L_s}$ denote the hash computed by $P_i \in \{P_1, P_2, P_3\}$ on all β_{z} up to level L_s in the circuit.
    - P_1, P_2, P_3 perform a binary search to identify a pair of consecutive levels L_p, L_{p+1} in the circuit such that $\mathsf{H}_i^{L_p}$ is consistent, but $\mathsf{H}_i^{L_{p+1}}$ is inconsistent.
    - P_1, P_2 jmp-sv β_{z} up to level L_p to P_4, P_5.

      If the jmp-sv is a success, then parties perform deferred jmp-vrfy with respect to all β_{z} up to level L_{p+1}. If the jmp-vrfy is a success, P_i matches its hash $\mathsf{H}_i^{L_{p+1}}$ against the hashes received to identify the party that sent an incorrect $\mathsf{H}^{L_{p+1}}$. P_i broadcasts the identity of this corrupted party $P^\star$ to all parties in $\mathcal{P}$. All parties set $\mathsf{CP} = (P_i, P^\star)$ where P_i is the party with the least index.
    - If L_p is the same as the last level in the segment, then $P_i \in \{P_1, P_2, P_3\}$ matches its hash H_i against the hashes received to identify the party that sent an incorrect H in the first consistency check. P_i broadcasts the identity of this corrupted party $P^\star$ to all parties in $\mathcal{P}$. All parties set $\mathsf{CP} = (P_i, P^\star)$ where P_i is the party with the least index.

Figure 2: Multiplication protocol

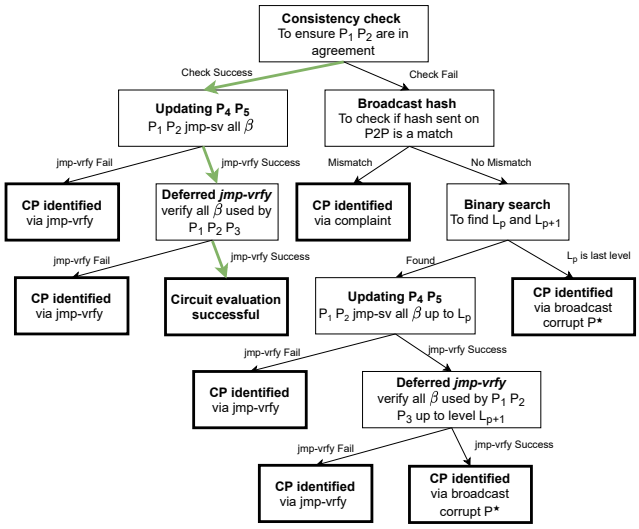


Figure 3: Flow of verification phase when online party is malicious

3.4 Reconstruction

Protocol Π_{rec} enables robust reconstruction of a $[\![\cdot]\!]$-shared value v towards P_i. For this, observe that each party misses 4 shares, and each such share is held by three other parties. Thus, to reconstruct v towards P_i, parties can send the missing shares to P_i. For each share, P_i uses the value which appears in the majority to reconstruct v. As on optimization, we let two parties send the value while the third send its hash to P_i. The protocol appears in Fig. 13.

3.5 The complete 5PC

We give an overview of the execution of 5PC for computing any function. The complete protocol can be divided into three stages: input sharing, evaluation, output reconstruction. Each stage is further cast in the preprocessing model, which comprises a preprocessing phase and an online phase. The protocol execution is preceded by a one-time shared key setup and begins by executing the preprocessing phase for each of the three stages. Note that protocols in each of these stages rely on several invocations of jmp. Thus, they either complete successfully or in case of a misbehaviour, a conflict pair CP is identified. To leverage amortization, only the *send* of all jmps are run on the flow while all *verify* steps are deferred until output reconstruction. Recall that identification of CP calls for rerunning of the protocol via 3PC. Thus, deferring verification until output reconstruction would result in the worst-case cost of executing 5PC and 3PC. To avoid this, a possible optimization is to divide the computation of the circuit into segments[3], with a checkpoint placed at the end of each segment. Computation carried out in a segment can be verified at each checkpoint. In this way, if a CP is identified in any segment, computation of this segment restarts with a 3PC execution. For this, a share conversion is performed to convert shares from 5PC to 3PC. The details of the same are provided next. All the subsequent segments can now be evaluated via the 3PC. The complete protocol appears in Fig. 4 and proofs in §E.

[3]A circuit is sliced depth-wise into segments comprising multiple levels/layers.

Protocol 5PC − FaF

One-time shared key setup is performed to generate common PRF keys which can be used to generate correlated randomness.

Preprocessing Phase:

– For each input gate u, parties execute preprocessing phase of Π_{sh} to obtain $[\alpha_{\text{u}}]$.

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

– For each multiplication gate with input wires u, v and output wire w, parties execute preprocessing phase of Π_{mult} to obtain $[\alpha_{\text{w}}], [\alpha_{\text{uv}}]$.

Online Phase:

– For each input v held by a party, parties invoke the online phase of Π_{sh} to generate $[\![\text{v}]\!]$.

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

– For each multiplication gate with input wires u, v and output wire w, parties execute the online phase of Π_{mult} to generate $[\![\text{w}]\!]$.

– For each output gate, parties execute Π_{rec} to reconstruct output w towards the designated party.

Semi-honest 3PC: If a CP is identified at any step, perform share conversion and continue computation with semi-honest 3PC.

Figure 4: 5PC FaF Protocol

Share conversion. We describe the (3, 1) replicated secret sharing (RSS) semantics for a 3PC protocol followed by the steps for share conversion, where the latter is similar to that described in [12] optimized for our setting. Let $\mathcal{P}' = \{P'_0, P'_1, P'_2\}$ denote the three parties. Let $\text{v} = \text{v}_0 + \text{v}_1 + \text{v}_2$ where $(\text{v}_i, \text{v}_{(i+1)\%3})$ are the shares held by P'_i that define a (3, 1) RSS scheme. Observe here that a value is split into three shares, each of which is held by two parties. Our goal is to convert from the 5PC $[\![\cdot]\!]$-sharing (which is an augmented (5, 2)-RSS sharing with an additional β held by all parties) to a (3, 1)-RSS sharing. The conversion proceeds as follows. Let P_i, P_j be parties to be eliminated. The residual three parties are arbitrarily assigned roles of P'_0, P'_1, P'_2. To generate the (3, 1)-RSS shares among parties in $\mathcal{P}' = \mathcal{P} \setminus \{P_i, P_j\}$, consider the following types of shares.

1. Shares that are known to either P_i or P_j: Such shares are already held by two other parties in $\mathcal{P} \setminus \{P_i, P_j\}$, which is what is needed for the (3, 1)-RSS sharing.

2. Shares that are not known to both P_i, P_j: Such shares are known to all the three residual parties. Since exactly two parties should hold each share, we let the party with the lowest index remove this share from its possession.

3. Shares that are known to both P_i and P_j: Such shares are known to exactly one other party, say P_k, in $\mathcal{P} \setminus \{P_i, P_j\}$. To enable one other party to hold this share to complete the (3, 1)-RSS sharing, we let P_k send this share to the remainder party, say P_l.

4. Shares that are held by all (β): We let parties enacting the role of P'_1, P'_2 incorporate this share in its set of common shares, and let P'_0 remove this share from its possession.

We explain the share conversion steps with a concrete example. Let P_1, P_2 be the parties to be eliminated, and let $P'_0 = P_3, P'_1 = P_4, P'_2 = P_5$. Consider conversion of $[\![\text{v}]\!]$ to a (3,1)-RSS share. For type 1 shares, shares that are held by P_1 or P_2 include $\alpha_{\text{v}_{13}}, \alpha_{\text{v}_{23}}, \alpha_{\text{v}_{14}}, \alpha_{\text{v}_{24}}, \alpha_{\text{v}_{15}}, \alpha_{\text{v}_{25}}$, where every consecutive pair of shares is held by $\{P_4, P_5\}$,

$\{P_3, P_5\}, \{P_3, P_4\}$, respectively. With respect to type 2 shares, shares that are not known to both P_1, P_2 include $\alpha_{v_{12}}$. These are included by $\{P_4, P_5\}$ in their set of shares. For type 3 shares, shares that are known to both P_1, P_2 include $\alpha_{v_{34}}, \alpha_{v_{35}}, \alpha_{v_{45}}$, which are held by P_5, P_4, P_3, respectively. Let P_3 send $\alpha_{v_{45}}$ to P_4, let P_4 send $\alpha_{v_{35}}$ to P_5, and let P_5 send $\alpha_{v_{34}}$ to P_3. Finally, for the last type of share, we let P_4, P_5 include β_v in its set of shares. The (3, 1)-RSS shares of v are now defined as $v_0 = -\alpha_{v_{25}} - \alpha_{v_{15}} - \alpha_{v_{45}}$ which is held by P_3, P_4, $v_2 = -\alpha_{v_{24}} - \alpha_{v_{14}} - \alpha_{v_{34}}$ which is held by P_3, P_5, and $v_1 = \beta_v - \alpha_{v_{23}} - \alpha_{v_{13}} - \alpha_{v_{35}} - \alpha_{v_{12}}$ which is held by P_4, P_5. This generates the (3, 1)-RSS shares of v from $[\![v]\!]$.

4 BUILDING BLOCKS

In this section, we discuss 5PC $(1, 1)$-FaF realizations of building blocks (Table 1) required for the applications considered. Most of these are well studied in the literature [36, 37, 43, 55]. Hence, here we only highlight those which were challenging to achieve in the 5PC $(1, 1)$-FaF setting.

Multi-input multiplication. To reduce the online communication cost as well as the round complexity, we design protocols to enable multiplication of 3 and 4 inputs in a single shot [37, 54, 55]. Compared to the naive approach of performing sequential multiplications to multiply 3 and 4 inputs, the multi-input multiplication protocol enjoys the benefit of having the same online phase complexity as that of the 2-input multiplication protocol. This brings in a 2× improvement in the online round complexity, while also improving the online communication cost. We extend the ideas of [37] to achieve this in our setting. For instance, the goal of 3-input multiplication is to generate $[\![z]\!]$ given $[\![\cdot]\!]$-shares of $a, b, c \in \mathbb{Z}_{2^\ell}$ where $z = abc$. Observe that, $\beta_z = abc + \alpha_z = \beta_{abc} - \beta_{ac}\alpha_b - \beta_{bc}\alpha_a - \beta_{ab}\alpha_c + \beta_a\alpha_{bc} + \beta_b\alpha_{ac} + \beta_c\alpha_{ab} - \alpha_{abc} + \alpha_z$. Thus, parties generate $[\alpha_{ab}], [\alpha_{ac}], [\alpha_{bc}], [\alpha_{abc}]$ during preprocessing by invoking $\mathcal{F}_{\mathsf{MulPre}}$ (Fig. 34), and proceed with a similar online phase as in 2-input multiplication. Similarly, for 4-input multiplication $[\cdot]$-shares of $\alpha_{ab}, \alpha_{ac}, \alpha_{ad}, \alpha_{bc}, \alpha_{bd}, \alpha_{cd}, \alpha_{abc}, \alpha_{abd}, \alpha_{acd}, \alpha_{bcd}$, α_{abcd} are needed.

Dot product. Given $[\![\cdot]\!]$-shares of vectors $\vec{x}, \vec{y}$ where each element of the vector is $[\![\cdot]\!]$-shared, protocol Π_{dotp}, enables generation of $[\![\cdot]\!]$-shares of $z = \vec{x} \odot \vec{y}$, where $\odot$ denotes the dot product operation. For this, observe that β_z can be written as

$$\beta_z = z + \alpha_z = \vec{x} \odot \vec{y} + \alpha_z = \sum_{i=1}^{n} x_i y_i + \alpha_z = \sum_{i=1}^{n} (\beta_{x_i y_i} - \beta_{x_i}\alpha_{y_i} - \beta_{y_i}\alpha_{x_i} + \alpha_{x_i y_i}) + \alpha_z \tag{1}$$

Thus, the goal of preprocessing phase is to generate $[\cdot]$-shares of $\sigma = \sum_{i=1}^{n} \alpha_{x_i y_i}$, which is a dot product of $\{\alpha_{x_i}\}_{i=1}^{n}, \{\alpha_{y_i}\}_{i=1}^{n}$. Given $[\sigma]$, parties proceed with a similar online phase as that in multiplication to compute β_z (Eq. (1)), where the terms are locally added before being sent, making the online communication independent of n [36, 56]. Similar to [36], to make the preprocessing communication for generating $[\sigma]$ independent of n, parties execute a semi-honest dot-product protocol [25] whose communication cost is independent of n. This is followed by a verification phase, similar to the one in [12], where parties invoke Π_{Verify}[4] (see Fig. 33, §F) on the dot product triple, $\left[\{\alpha_{x_i}\}_{i=1}^{n}\right], \left[\{\alpha_{y_i}\}_{i=1}^{n}\right], [\sigma]$, to verify correctness of $[\sigma]$. As opposed to verification of m multiplication triples which requires a communication cost of $O(\log(m))$ elements, the cost for verifying the correctness of m dot products with vectors of size n now becomes $O(\log(mn))$ elements. Thus, for large m, the verification cost can be amortized, making the preprocessing communication cost independent of n. Due to its similarity to multiplication, we omit formal protocol for dot product.

Matrix multiplication and convolution. Matrix multiplication can easily be reduced to dot product where each element in the resultant matrix can be computed via a dot product. Convolutions can also be reduced to matrix multiplication following standard techniques [61].

Truncation. Repeated multiplications in fixed point arithmetic (FPA) cause an overflow. This necessitates the need for truncation, which truncates the last d bits from the result of multiplication, to retain FPA semantics. We follow a similar approach as in [43, 45] for probabilistic truncation. Here, to truncate a value v, we rely on a (r, r^d)-pair, where $r \in \mathbb{Z}_{2^\ell}$ and r^d is the truncated value of r (i.e. $r^d = r/2^d$). The truncated value v^d of v, is computed as $v^d = (v - r)^d + r^d$.

Given $[\![r]\!], [\![r^d]\!]$ can be generated in the preprocessing phase, our multiplication protocol can be modified to incorporate truncation without incurring any overhead in the online phase as follows. Use r instead of α_z while computing β_z. Parties truncate β_z locally to generate ${\beta_z}^d = (z + r)^d$ and generate $[\![(z + r)^d]\!]$ non-interactively (see §B.3), followed by computing $[\![z^d]\!] = [\![(z + r)^d]\!] - [\![r^d]\!]$. To generate $[\![\cdot]\!]$-shares of the truncation pair (r, r^d), we extend ideas in [43] to our setting, and the resultant protocol is called Π_{TrPair}. For this, parties non-interactively generate $[\![r]\!]^{\mathbf{B}}$ using their shared-key setup, and truncate the last d bits of each of its share to generate $[\![r^d]\!]^{\mathbf{B}}$. To obtain $[\![r]\!]$ from $[\![r]\!]^{\mathbf{B}}$, parties proceed as follows. Analogous steps enable generation of $[\![r^d]\!]$ from $[\![r^d]\!]^{\mathbf{B}}$. Set $\beta_r = 0$ in $[\![r]\!]$. Let the other shares of $[\![r]\!]$ be denoted as r_{ij} for $1 \le i < j \le 5$. Without loss of generality, parties non-interactively sample all r_{ij} but r_{12}, as per the $[\![\cdot]\!]$-sharing. Enabling P_3, P_4, P_5 obtain $r_{12} = r - \sum_{ij \neq 12} r_{ij}$ will complete generation of $[\![r]\!]$. For this, observe that we can write $r = v_1 + v_2 + v_3 + v_4$ where $v_1 = r_{34} + r_{35} + r_{45}$ is held by P_1, P_2, $v_2 = r_{45} + r_{25}$ is held by P_1, P_3, $v_3 = r_{14} + r_{15}$ is held by P_2, P_3 and $v_4 = r_{12} + r_{13} + r_{23}$ is held by P_4, P_5. Thus, revealing $v_4 = r - v_1 - v_2 - v_3$ to P_4, P_5 enables them to compute $r_{12} = v_4 - r_{13} - r_{23}$ which they can send to P_3 by invoking Π_{jmp}, thereby generating $[\![r]\!]$. For this, given $[\![r]\!]^{\mathbf{B}}$, parties compute $[\![v_4]\!]^{\mathbf{B}} = [\![r]\!]^{\mathbf{B}} + \sum_{i=1}^{3} [\![-v_i]\!]^{\mathbf{B}}$ by evaluating Boolean addition circuit. Elaborately, P_1, P_2 generate $[\![-v_1]\!]^{\mathbf{B}}$, P_1, P_3 generate $[\![-v_2]\!]^{\mathbf{B}}$, and P_2, P_3 generate $[\![-v_3]\!]^{\mathbf{B}}$ by invoking the joint sharing protocol, Π_{JSh2} (§B.3). Note that this joint sharing generates $[\![-v_i[k]]\!]^{\mathbf{B}}$ for each bit $-v_i[k]$ for $-v_i$ for $i \in \{1, 2, 3\}, k \in \{0, \ldots, \ell - 1\}$. Parties proceed to compute the sum $[\![v_4[k]]\!]^{\mathbf{B}} = [\![r[k]]\!]^{\mathbf{B}} + \sum_{i=1}^{3} [\![-v_i[k]]\!]^{\mathbf{B}}$ for each bit using a full adder (FA) circuit, as described in [43]. It follows from [43] that $x = x_1 + x_2 + x_3$ can be expressed as $x = 2c + s$ where $\mathrm{FA}(x_1[k], x_2[k], x_3[k]) \rightarrow (c[k], s[k])$ for $k \in \{0, \ldots, \ell - 1\}$. Here, s and c denote the sum and carry bits respectively. Thus, parties compute $[\![v_4[k]]\!]^{\mathbf{B}}$ for $k \in \{0, \ldots, \ell - 1\}$, simultaneously, by executing the FA's as given below.

[4] Note that computations in Π_{Verify} remain unchanged except that its input parameters now correspond to dot product triples.

– $\text{FA}(\mathsf{r}[k], -\mathsf{v}_1[k], -\mathsf{v}_2[k]) \rightarrow (c_1[k], s_1[k])$
– $\text{FA}(-\mathsf{v}_3[k], c_1[k-1], s_1[k]) \rightarrow (c_2[k], s_2[k])$
– $\text{PPA}(2c_2, s_2) \rightarrow \mathsf{v}_4$

After the FA is executed, $[\![\mathsf{v}_4]\!]^{\mathbf{B}}$ is computed using the 2-input Parallel Prefix Adder (PPA) circuit [43] on inputs $2[\![c_2]\!]^{\mathbf{B}}$, $[\![s_2]\!]^{\mathbf{B}}$. The computations above are carried out on the $[\![\cdot]\!]^{\mathbf{B}}$-shares, and $2c_1[k] = c_1[k-1]$ and $c[-1] = 0$. Having obtained $[\![\mathsf{v}_4]\!]^{\mathbf{B}}$, parties reconstruct v_4 towards P_4, P_5, who compute $\mathsf{r}_{12} = \mathsf{v}_4 - \mathsf{r}_{13} - \mathsf{r}_{23}$, and invoke Π_{jmp} to send it to P_3. This completes generation of $[\![\mathsf{r}]\!]$.

Bit to arithmetic. Protocol Π_{bit2A} allows computation of arithmetic shares, $[\![\mathsf{b}^{\mathsf{R}}]\!]$ of a bit $\mathsf{b} \in \mathbb{Z}_2$ from its Boolean shares, $[\![\mathsf{b}]\!]^{\mathbf{B}}$, where b^{R} denotes arithmetic equivalent of b over $\mathbb{Z}_{2^\ell}$. Observe that, following [37],

$$\mathsf{b}^{\mathsf{R}} = (\beta_{\mathsf{b}} \oplus \alpha_{\mathsf{b}})^{\mathsf{R}} = {\beta_{\mathsf{b}}}^{\mathsf{R}} + {\alpha_{\mathsf{b}}}^{\mathsf{R}} - 2{\beta_{\mathsf{b}}}^{\mathsf{R}}{\alpha_{\mathsf{b}}}^{\mathsf{R}}. \tag{2}$$

Given $\left[{\alpha_{\mathsf{b}}}^{\mathsf{R}}\right]$ and $[\mathsf{r}]$ for $\mathsf{r} \in \mathbb{Z}_{2^\ell}$ can be generated in the preprocessing phase, parties can compute $\left[\mathsf{b}^{\mathsf{R}} + \mathsf{r}\right]$ in the online phase and reconstruct it towards all. Possession of $\mathsf{b}^{\mathsf{R}} + \mathsf{r}$ by all enables non-interactive generation of its $[\![\cdot]\!]$-shares (§B.3), from which $[\![\mathsf{b}^{\mathsf{R}}]\!] = [\![\mathsf{b}^{\mathsf{R}} + \mathsf{r}]\!] - [\![\mathsf{r}]\!]$ can be computed. To generate $\left[{\alpha_{\mathsf{b}}}^{\mathsf{R}}\right]$, parties first generate $[\![{\alpha_{\mathsf{b}}}^{\mathsf{R}}]\!]$, and convert it to $[\cdot]$-shares via $\Pi_{[\![\cdot]\!] \rightarrow [\cdot]}$ ((§3.1). To generate $[\![{\alpha_{\mathsf{b}}}^{\mathsf{R}}]\!]$, observe that the $[\cdot]^{\mathbf{B}}$-shares of α_{b} can be written as $\alpha_{\mathsf{b}} = \nu_1 \oplus \nu_2 \oplus \nu_3 \oplus \nu_4$ where $\nu_1 = \alpha_{\mathsf{b}_{34}} \oplus \alpha_{\mathsf{b}_{35}} \oplus \alpha_{\mathsf{b}_{45}}$, $\nu_2 = \alpha_{\mathsf{b}_{24}} \oplus \alpha_{\mathsf{b}_{25}}$, $\nu_3 = \alpha_{\mathsf{b}_{14}} \oplus \alpha_{\mathsf{b}_{15}}$ and $\nu_4 = \alpha_{\mathsf{b}_{12}} \oplus \alpha_{\mathsf{b}_{13}} \oplus \alpha_{\mathsf{b}_{23}}$. As seen in truncation pair generation, P_1, P_2 hold ν_1, P_1, P_3 hold ν_2, P_2, P_3 hold ν_3 and P_4, P_5 hold ν_4. Given $[\![\cdot]\!]^{\mathbf{B}}$-shares of each of $\nu_1, \nu_2, \nu_3, \nu_4$ can be generated via Π_{JSh2}, parties generate $[\cdot]$-shares of $\mathsf{p} = \nu_1 \oplus \nu_2$ and $\mathsf{q} = \nu_3 \oplus \nu_4$ using Eq. (2), and use these values to generate $[\![\alpha_{\mathsf{b}}{}^{\mathsf{R}}]\!] = [\![(\mathsf{p} \oplus \mathsf{q})^{\mathsf{R}}]\!]$. The protocol appears in Fig. 14.

Bit extraction. Bit extraction (Π_{bitext}) enables generation of $[\![\cdot]\!]^{\mathbf{B}}$-shares of the most significant bit (msb) of a value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ given $[\![\mathsf{v}]\!]$. Support for multi-input multiplication enables usage of the optimized bit extraction circuit proposed in [55], which takes two values as inputs and outputs the msb of the sum of these values. Given $[\![\mathsf{v}]\!]$, we generate the Boolean shares of the two inputs to the bit extraction circuit as follows. Observe that v can be written $\mathsf{v} = \beta_{\mathsf{v}} + (-\alpha_{\mathsf{v}})$. Thus, β_{v} and $-\alpha_{\mathsf{v}}$ serve as the two inputs. $[\![\beta_{\mathsf{v}}]\!]^{\mathbf{B}}$ can be generated non-interactively in the online phase since all parties hold β_{v} (see §B.3). To generate $[\![-\alpha_{\mathsf{v}}]\!]^{\mathbf{B}}$ from $[\alpha_{\mathsf{v}}]$, parties proceed as follows in the preprocessing phase. Parties first generate $-\alpha_{\mathsf{v}}$ by locally negating all their shares of α_{v}. For ease of presentation, let $\alpha = -\alpha_{\mathsf{v}}$ and $[\alpha] = [-\alpha_{\mathsf{v}}] = (\alpha_{ij})_{1 \le i < j \le 5}$. Recall that $\alpha = \nu_1 + \nu_2 + \nu_3 + \nu_4$ where $\nu_1 = \alpha_{34} + \alpha_{35} + \alpha_{45}$, $\nu_2 = \alpha_{24} + \alpha_{25}$, $\nu_3 = \alpha_{14} + \alpha_{15}$ and $\nu_4 = \alpha_{12} + \alpha_{13} + \alpha_{23}$, and each term is held by a pair of parties. Similar to Π_{TrPair}, after the different pairs of parties generate $[\![\nu_1]\!]^{\mathbf{B}}$, $[\![\nu_2]\!]^{\mathbf{B}}$, $[\![\nu_3]\!]^{\mathbf{B}}$, $[\![\nu_4]\!]^{\mathbf{B}}$, evaluating two sequential full adders followed by a PPA circuit generates $[\![\alpha]\!]^{\mathbf{B}}$. Having obtained $[\![\alpha]\!]^{\mathbf{B}}$ and $[\![\beta_{\mathsf{v}}]\!]^{\mathbf{B}}$, parties execute the optimized bit extraction circuit to extract the $\mathsf{msb}(\mathsf{v})$.

Arithmetic to Boolean. Protocol Π_{A2B} generates $[\![\cdot]\!]^{\mathbf{B}}$-shares for each bit of $\mathsf{v} \in \mathbb{Z}_{2^\ell}$, denoted as $[\![\mathsf{v}]\!]^{\mathbf{B}}$, from $[\![\mathsf{v}]\!]$. For this, observe that $\mathsf{v} = \beta_{\mathsf{v}} + (-\alpha_{\mathsf{v}})$. Thus, evaluating the optimized PPA circuit [55] on $[\![\beta_{\mathsf{v}}]\!]^{\mathbf{B}}$, $[\![-\alpha_{\mathsf{v}}]\!]^{\mathbf{B}}$ generates $[\![\mathsf{v}]\!]^{\mathbf{B}}$. For this, $[\![\beta_{\mathsf{v}}]\!]^{\mathbf{B}}$ can be generated non-interactively since all parties hold β_{v} (see §B.3). To generate $[\![-\alpha_{\mathsf{v}}]\!]^{\mathbf{B}}$ from $[\alpha_{\mathsf{v}}]$, parties follow the steps as described in Π_{bitext}.

Bit injection. Given $[\![\mathsf{b}]\!]^{\mathbf{B}}$, $[\![\mathsf{v}]\!]$ where $\mathsf{b} \in \mathbb{Z}_2$, $\mathsf{v} \in \mathbb{Z}_{2^\ell}$, bit injection ($\Pi_{\mathsf{BitInj}}$) generates $[\![\mathsf{b}^{\mathsf{R}} \cdot \mathsf{v}]\!]$. For this, parties run Π_{bit2A} to generate $[\![\mathsf{b}^{\mathsf{R}}]\!]$, followed by Π_{mult} to generate $[\![\mathsf{b}^{\mathsf{R}} \cdot \mathsf{v}]\!]$.

Oblivious select. Protocol Π_{sel} takes as input $[\![\mathsf{x}_1]\!]$, $[\![\mathsf{x}_2]\!]$, $[\![\mathsf{b}]\!]^{\mathbf{B}}$, where $\mathsf{x}_1, \mathsf{x}_2 \in \mathbb{Z}_{2^\ell}$ and $\mathsf{b} \in \mathbb{Z}_2$, and outputs re-randomized $[\![\cdot]\!]$-shares of $\mathsf{z} = \mathsf{x}_{\mathsf{b}}$. Since $\mathsf{z} = \mathsf{x}_{\mathsf{b}} = \mathsf{b}(\mathsf{x}_1 - \mathsf{x}_0) + \mathsf{x}_0$, computing $[\![\mathsf{z}]\!]$ requires one invocation of Π_{BitInj} and addition operations.

Equality check. On input $[\![\mathsf{x}]\!]$, $[\![\mathsf{y}]\!]$, equality check protocol (Π_{eq}) outputs $[\![\mathsf{b}]\!]^{\mathbf{B}}$ where $\mathsf{b} = 1$, if $\mathsf{x} = \mathsf{y}$, and $\mathsf{b} = 0$, otherwise. Similar to [55], the approach is to compute $\mathsf{v} = \mathsf{x} - \mathsf{y}$ and check if all bits of v are 0. Concretely, parties first obtain $[\![\mathsf{v}]\!]^{\mathbf{B}}$ by invoking Π_{A2B} on $[\![\mathsf{v}]\!]$, compute $[\![\bar{\mathsf{v}}]\!]^{\mathbf{B}}$ ($\bar{\mathsf{v}}$ denotes bit complement of v) non-interactively, followed by invoking the 4-input (Boolean) multiplication, recursively, to generate $[\![\mathsf{b}]\!]^{\mathbf{B}}$.

Comparison. On input $[\![\mathsf{x}]\!]$, $[\![\mathsf{y}]\!]$, Π_{comp} outputs $[\![\mathsf{b}]\!]^{\mathbf{B}}$ where $\mathsf{b} = 1$, if $\mathsf{x} < \mathsf{y}$, and $\mathsf{b} = 0$, otherwise. This reduces to checking msb of $\mathsf{v} = \mathsf{x} - \mathsf{y}$, and hence, Π_{bitext} can be used.

Maxpool/minpool. Maxpool allows computing the maximum element from a set of m elements. We follow a similar approach as in [37], where the elements are recursively compared in a pairwise manner to obtain the maximum element. Minpool can also be computed analogously.

ReLU. The relu function computes the maximum between 0 and a value v, and can be computed as $\text{ReLU}(\mathsf{v}) = \bar{\mathsf{b}} \cdot \mathsf{v}$, where $\mathsf{b} = 1$ if $\mathsf{v} < 0$ and $\mathsf{b} = 0$, otherwise. Here, $\bar{\mathsf{b}}$ denotes the complement of bit b. Given $[\![\mathsf{v}]\!]$, b can be computed via Π_{bitext}, followed by non-interactively computing $\bar{\mathsf{b}}$, followed by Π_{BitInj} to compute $[\![\bar{\mathsf{b}}^{\mathsf{R}} \cdot \mathsf{v}]\!]$.

Complexity of building blocks. Table 1 lists the complexities of the designed building blocks.

Building block	Online		Preprocessing
	Rounds	Comm. (in bits)	Comm. (in bits)
Multiplication	1	8ℓ	6ℓ
3-input Multiplication	1	8ℓ	24ℓ
4-input Multiplication	1	8ℓ	66ℓ
Dot product	1	8ℓ	6ℓ
Matrix Multiplication	1	$8\mathsf{pq}\ell$	$6\mathsf{pq}\ell$
Multiplication with Truncation	1	8ℓ	$27\ell + 6\ell \log_2 \ell$
Bit to arithmetic	1	8ℓ	22ℓ
Bit extraction	$\log_4 \ell$	u'_2	$16\ell + 6\ell \log_2 \ell + \mathsf{u}'_1$
Arithmetic to Boolean	$\log_4 \ell$	u_2	$16\ell + 6\ell \log_2 \ell + \mathsf{u}_1$
Bit Injection	2	16ℓ	28ℓ
Oblivious Select	2	16ℓ	28ℓ
Equality	$\log_4 \ell$	$\mathsf{u}_2 + 168$	$16\ell + 6\ell \log_2 \ell + \mathsf{u}_1 + 1386$
Comparison	$\log_4 \ell$	u'_2	$16\ell + 6\ell \log_2 \ell + \mathsf{u}'_1$
Maxpool/minpool	$\log_2 m(\log_4 \ell + 2)$	$(m-1)(\mathsf{u}'_2 + 16\ell)$	$(m-1)(44\ell + 6\ell \log_2 \ell + \mathsf{u}'_1)$
ReLU	$\log_4 \ell + 2$	$\mathsf{u}'_2 + 16\ell$	$44\ell + 6\ell \log_2 \ell + \mathsf{u}'_1$

- ℓ: size of ring in bits, instantiated with $\ell = 64$
- $\mathsf{p} \times \mathsf{q}$ denotes the dimension of resultant matrix after matrix multiplication
- $\mathsf{u}'_1 = 6\mathsf{n}_2 + 24\mathsf{n}_3 + 66\mathsf{n}_4$, $\mathsf{u}'_2 = 8\,(\mathsf{n}_2 + \mathsf{n}_3 + \mathsf{n}_4)$ where $\mathsf{n}_2 = 41, \mathsf{n}_3 = 27, \mathsf{n}_4 = 47$ denote the number of AND gates in the optimized bit extraction circuit of [55] with 2, 3, 4 inputs, respectively.
- $\mathsf{u}_1 = 6\mathsf{n}_2 + 24\mathsf{n}_3 + 66\mathsf{n}_4$, $\mathsf{u}_2 = 8\,(\mathsf{n}_2 + \mathsf{n}_3 + \mathsf{n}_4)$ where $\mathsf{n}_2 = 216$, $\mathsf{n}_3 = 184$, $\mathsf{n}_4 = 179$ denote the number of AND gates in the optimized PPA circuit of [55] with 2, 3, 4 inputs, respectively.
- m denotes number of elements to be compared via maxpool.

Table 1: Building blocks with their complexity

5 APPLICATIONS AND BENCHMARKS

As described in §3.3, rather than having a multiplication protocol with all parties online, considerable effort was spent in reducing the number of online parties to only 3. We now showcase the concrete improvements brought in by this approach. The results corroborate that the reduction in online parties is indeed beneficial. In addition to the above, we empirically show the practicality of our protocols for securely performing matching in dark pools, and for evaluating privacy-preserving machine learning (PPML) algorithms. For this, we consider the secure outsourced setting (see §A for details), and benchmark via the optimized variant of the multiplication protocol.

Our (1,1)-FaF 5PC is the first instantiation of a FaF secure protocol. In comparison to (traditional) protocols in the literature, we note the following. The setting of (5,1) is not popular and there is no concretely secure protocol. On the other hand, the setting of (4,1) achieving GOD is well-studied [24, 36, 37]. However, it is unsurprising, given the asymptotic complexity of (4,1) MPC protocol, it would naturally fare better. Hence, despite the mismatch in the number of parties, we can estimate the overhead incurred in moving from traditional 4PC [37] to FaF secure 5PC. We observe an overhead of 1.62× in the online and 3× in the preprocessing communication cost. We believe that the overhead, which is the price paid for obtaining FaF security, is reasonable enough. Further, in §5.1, we establish a concrete efficiency gain of up to 1.6× of our $(1, 1)$-FaF secure protocol over the traditional $(5, 2)$ protocol with respect to online efficiency. In §5.2 we benchmark the performance of the applications by instantiating them with our FaF-secure protocols.

Benchmark environment and parameters. We report results in LAN (1 Gbps bandwidth) with 2.3 GHz Quad-Core Intel Core i7 machines having 16GB RAM. The average round trip time (rtt) for communicating 1KB data between a pair of machines is 0.29 milliseconds (ms). The protocols build on the ENCRYPTO library [22] in C++17 over a 64 bit ring. We use multi-threading, wherever possible, to facilitate efficient computation and communication among the parties. We only estimate these results since the correctness of the dark pool algorithms and accuracy of NN algorithms follow from prior works [16, 45, 63]. Since there is no defined way to capture an adversary's misbehaviour, following standard practice [13, 36, 43, 56], we benchmark honest executions of the protocols, including the verification required to attain GOD. Hence the reported run time does not account for the 3PC execution. Note, however, that 5PC execution itself accounts for the worst-case computation because it has a higher number of parties, including one malicious corruption as opposed to 3PC. Further, considering perennially running protocols such as CDA, the cost of switching to 3PC and continuing a semi-honest execution would be much lesser compared to executing the protocol using 5PC perennially.

We use the time taken for the protocol to complete and communication between parties as the two parameters for benchmarks. We report these values separately for the online and preprocessing phases. Further, we also report online throughput (TP), which is the number of buy/sell orders that can be processed in a second for the dark pool algorithms, whereas it is the number of inference queries that can be processed in a second for PPML algorithms.

5.1 Benefit of having fewer parties online

We compare our optimized multiplication protocol, which requires only 3 out of the 5 parties for most of the online phase, with the non-optimized variant, that requires all parties to remain online. We also compare our protocol with the traditional $(5, 2)$ protocol obtained from [12], which also requires all parties to be online. To showcase the improvement achieved in the optimized variant, we benchmark synthetic circuits of varying depths (10, 100, 1000, 10000) with 100 multiplication gates at each layer. For all the variants, we report the time, throughput (number of circuit evaluations that can be performed in a second) and monetary cost of the system in Table 2. While throughput simultaneously captures the improvements in communication and round complexity, we additionally report monetary costs to showcase the effect of the number of parties on the operational cost of the system. We report these values only for the online phase. We estimate the monetary cost following standard Google Cloud pricing[5].

The round complexity of the non-optimized variant is roughly 3× that of the optimized variant, assuming that the time taken for jmp-vrfy gets amortized. This is evident from the reported online time, for circuit depth 100 and beyond. This is however not the case for circuit of depth 10. This is because the time taken for the jmp-vrfy in the optimized variant (2 rounds) is comparable to that of circuit evaluation (10 rounds for jmp-send) and hence does not get amortized. The improved online time is reflected as improvements in throughput as well, where the gain is up to 3×. Finally, the reduction in the number of online parties is clearly evident in monetary cost, since it captures the price paid to host the required number of parties (inclusive of its computation and communication). The optimized variant witnesses up to 69% savings in monetary cost compared to the non-optimized variant.

With respect to [12], our optimized variant has an improvement of up to 1.6× in run time and throughput. While the monetary cost reported in Table 2 is for online phase, to draw a fair comparison between our (optimized) protocol and [12], we also account for the monetary cost of preprocessing phase. In doing so, we observe that even for a circuit of depth 10000, the overall monetary cost of our protocol is 2.57×10^{-3} USD, which is only slightly higher than that of [12].

Circuit depth	Protocol type	Online time (s)	TP ($\times 10^2$)	Monetary cost ($\times 10^{-3}$ USD)
10	Optimized	0.005	121.189	0.002
	Non-optimized	0.011	59.435	0.006
	[12]	0.008	78.259	0.002
100	Optimized	0.034	19.037	0.018
	Non-optimized	0.107	6.006	0.057
	[12]	0.0543	11.783	0.031
1000	Optimized	0.329	1.948	0.178
	Non-optimized	1.059	0.604	0.571
	[12]	0.530	1.206	0.253
10000	Optimized	3.152	0.203	1.758
	Non-optimized	10.638	0.060	5.711
	[12]	5.154	0.124	2.452

Table 2: Comparison for synthetic circuits

[5]See https://cloud.google.com/vpc/network-pricing for network cost and https://cloud.google.com/compute/vm-instance-pricing for computation cost.

5.2 Dark pools

We consider two popular matching algorithms used in dark pools– continuous double auction (CDA) algorithm and volume-based matching algorithm. While the former processes orders in a continuous manner, the latter does so in scheduled intervals, and both the algorithms rely on different parameters for matching orders. Both these matching algorithms have been considered in prior works, albeit in the traditional MPC setting [16, 23]. Although the functionality of these algorithms remains the same as described in [16], we take advantage of possible parallelization and tweak the algorithms to improve their round complexity. This, in turn, improves the run time of the protocols and the number of orders that can be processed in unit time (throughput). We next detail each of these algorithms and their overall performance.

5.2.1 Continuous Double Auction. The CDA algorithm maintains a sorted list of buy orders ($\mathcal{B}$) and sell orders ($\mathcal{S}$) that are yet to be matched. A buy order comprises the client's identity, $name^b$, the units to be bought, b, and the buying price also known as *bid*, q. Analogously, a sell order comprises the client's identity $name^s$, the units to be sold s, and the selling price also known as *offer* p. All the unmatched buy orders in the list $\mathcal{B}$ (where $|\mathcal{B}| = M$) are sorted in descending order of their bid. Similarly, sell orders in list $\mathcal{S}$ (where $|\mathcal{S}| = N$) are sorted in ascending order of offer. The CDA algorithm maintains this as an invariant.

The CDA algorithm for processing a new order has two phases– (i) matching, and (ii) insertion. In the matching phase, the incoming order is matched with orders of the opposite type. Elaborately, a buy order is said to match a sell order if the following criteria are met– (i) *Price criteria*: the bid of the buy order must be greater than or equal to the offer of the sell order and, (ii) *Volume criteria*: the units of one order must be able to satisfy the units of the other. Thus, when a new buy order arrives, it is matched with the first order in $\mathcal{S}$ based on the matching criteria. The buy order may continue to be matched with other sell orders in $\mathcal{S}$, until either of the criteria for matching fails. Hence matches need not be one-to-one. An incoming sell order can also be processed analogously. The matching phase concludes with the incoming order being in one of the following two states. The order may be *satisfied* if all of its units are exhausted by getting matched to opposite orders, or, it may be *partially satisfied* if some of its units are still unmatched. If the incoming order is partially satisfied, the algorithm enters insertion phase that involves inserting this order into the corresponding list $\mathcal{B}$ or $\mathcal{S}$ while respecting the sorted order maintained within it. We refer to the algorithm in [16] for further details.

A secure variant of the CDA algorithm was given in [16], where all orders remain hidden until they are satisfied. However, the order type (buy or sell) and hence the size of $\mathcal{S}$ and $\mathcal{B}$ is not regarded as sensitive information. We now describe an improved secure protocol for the CDA algorithm to process an incoming buy order. An incoming sell order can be processed analogously.

In [16], the protocol identifies matching sell orders in $\mathcal{S}$ sequentially, and terminates when the incoming order can no longer be matched. Instead, we perform additional bookkeeping to identify all the matching sell orders in a single shot. This was not possible in [16] because the number of unmatched units remaining were tracked sequentially. However, we compute the cumulative sum w_i of the units of the first i sell orders in $\mathcal{S}$ which facilitates single shot identification of matching sell orders. While the satisfaction of the price criteria for all sell orders in $\mathcal{S}$ can be determined in parallel, w allows determining satisfaction of the volume criteria also, for all sell orders in parallel. Thus, one does not require to wait for the i^{th} order to be matched before processing the $i + 1^{\text{th}}$ order. Hence, all those sell orders where both the conditions are met can be executed and revealed in public. Note that the last sell order to be matched could either be fully satisfied or partially satisfied, and hence needs extra care. The protocol for the above matching phase is given in Fig. 5, where the changes made over the existing protocol are highlighted. The insertion phase follows this, where the incoming buy order is *obliviously* inserted into $\mathcal{B}$ in the correct slot that respects the ordering maintained as an invariant. Since the steps of protocol for the insertion phase as well as the overall CDA algorithm remain the same as in [16], we do not elaborate on them. However, we continue to execute independent instructions in parallel within these protocols, too, and render the overall execution as efficient as possible. The protocols for the insertion phase and overall CDA are given in Fig. 15 and Fig. 16 respectively in §D.

Protocol $\Pi_{\text{PSL}}\left(\mathcal{P}, (\llbracket name_0^b \rrbracket, \llbracket b_0 \rrbracket, \llbracket q_0 \rrbracket), \mathcal{S}\right)$

- Set $\llbracket w_0 \rrbracket = \llbracket 0 \rrbracket$
- For each $i = 1$ to N do in parallel: $\llbracket w_i \rrbracket = \sum_{j=1}^{i} \llbracket s_j \rrbracket$
- For each $i = 1$ to N do in parallel:
  - $\llbracket z_i \rrbracket^{\mathbf{B}} = \Pi_{\text{comp}}(\mathcal{P}, \llbracket w_{i-1} \rrbracket, \llbracket b_0 \rrbracket)$, $\llbracket z_i' \rrbracket^{\mathbf{B}} = \Pi_{\text{comp}}(\mathcal{P}, \llbracket p_i \rrbracket, \llbracket q_0 \rrbracket + 1)$
- For $i = 1$ to N do in parallel: $\llbracket f_i \rrbracket^{\mathbf{B}} = \Pi_{\text{mult}}(\mathcal{P}, \llbracket z_i \rrbracket^{\mathbf{B}}, \llbracket z_i' \rrbracket^{\mathbf{B}})$
- Reconstruct f_i's and set $k = i$ such that $f_i = 1$ and $f_{i+1} = 0$ for $i \in \{1, \ldots, N\}$. Else set $k = 0$.
- for each $i = 1$ to $k - 1$ do in parallel:
  - Reconstruct $(\llbracket name_i^s \rrbracket, \llbracket s_i \rrbracket, \llbracket p_i \rrbracket)$
- $\llbracket s_k' \rrbracket = \Pi_{\text{sel}}(\llbracket b_0 \rrbracket - \llbracket w_{k-1} \rrbracket, \llbracket s_k \rrbracket, \llbracket z_{k+1} \rrbracket^{\mathbf{B}})$
- Reconstruct $(\llbracket name_k^s \rrbracket, \llbracket s_k' \rrbracket, \llbracket p_k \rrbracket)$, set $\llbracket s_k \rrbracket = \llbracket s_k \rrbracket - \llbracket s_k' \rrbracket$
- Delete first $k - 1$ elements from $\mathcal{S}$.

Figure 5: CDA matching phase: processing sell list

5.2.2 Volume Matching. Unlike the CDA algorithm, where orders are processed in a continuous manner, the volume-based matching processes all the requests at fixed intervals. The algorithm matches orders based only on the volume. Hence, the i^{th} client only submits the number of units it wishes to buy b_i or sell s_i, and the matching is done on a first-come-first-serve basis. Similar to CDA the buy orders and sell orders are maintained in a separate list (queue), ordered by their arrival. Since the algorithm only accounts for volume, one is guaranteed that either all the sell orders or all buy orders are satisfied. That is, the type of orders whose total volume is lesser will be satisfied completely. After processing the orders, the algorithm outputs the sequence of updated buy/sell orders such that the value now at b_i' or s_i' denotes the number of units traded out of the original b_i or s_i request. Although the algorithm is the same as in [16], we provide a parallel variant of the same in Fig. 6 and highlight the changes made over the existing protocol. Unlike in [16], the algorithm can be improved to process each sell/buy order in parallel by some additional bookkeeping, as done in §5.2.1.

Protocol $\Pi_{\text{VM}}\left(\mathcal{P}, \{[\![s_i]\!]\}_{i=1}^{N}, \{[\![b_j]\!]\}_{j=1}^{M}\right)$

1. Compute $[\![S]\!] = \sum_{i=1}^{N} [\![s_i]\!]$ and $[\![B]\!] = \sum_{j=1}^{M} [\![b_j]\!]$
2. Compute $[\![f]\!]^{\mathbf{B}} = \Pi_{\text{comp}}(\mathcal{P}, [\![B]\!], [\![S]\!])$
3. Set $[\![T]\!] = \Pi_{\text{sel}}([\![S]\!], [\![B]\!], [\![f]\!]^{\mathbf{B}})$, $[\![s_0]\!] = 0$ and $[\![b_0]\!] = 0$
4. For i from 1 to N do in parallel: $[\![L_i^s]\!] = [\![T]\!] - \sum_{j=0}^{i-1} [\![s_j]\!]$
5. For i from 1 to M do in parallel: $[\![L_i^b]\!] = [\![T]\!] - \sum_{j=0}^{i-1} [\![b_j]\!]$
6. For i from 1 to N do in parallel:
   - $[\![z_1]\!]^{\mathbf{B}} = \Pi_{\text{comp}}(\mathcal{P}, [\![L_i^s]\!], [\![1]\!]))$ and $[\![z_2]\!]^{\mathbf{B}} = \Pi_{\text{comp}}(\mathcal{P}, ([\![L_i^s]\!], [\![s_i]\!])$
   - $[\![z_1]\!] = \Pi_{\text{bit2A}}(\mathcal{P}, [\![z_1]\!]^{\mathbf{B}})$ and $[\![z_2]\!] = \Pi_{\text{bit2A}}(\mathcal{P}, [\![z_2]\!]^{\mathbf{B}})$
   - $[\![s_i]\!] = (([\![L_i^s]\!] - [\![s_i]\!]) \cdot [\![z_2]\!] + [\![s_i]\!]) \cdot (1 - [\![z_1]\!])$
7. For j from 1 to M do in parallel:
   - $[\![z_1]\!]^{\mathbf{B}} = \Pi_{\text{comp}}(\mathcal{P}, [\![L_j^b]\!], [\![1]\!]))$ and $[\![z_2]\!]^{\mathbf{B}} = \Pi_{\text{comp}}(\mathcal{P}, ([\![L_j^b]\!], [\![b_j]\!])$
   - $[\![z_1]\!] = \Pi_{\text{bit2A}}(\mathcal{P}, [\![z_1]\!]^{\mathbf{B}})$ and $[\![z_2]\!] = \Pi_{\text{bit2A}}(\mathcal{P}, [\![z_2]\!]^{\mathbf{B}})$
   - $[\![b_i]\!] = (([\![L_j^b]\!] - [\![b_i]\!]) \cdot [\![z_2]\!] + [\![b_j]\!]) \cdot (1 - [\![z_1]\!])$
8. Reconstruct $[\![s_i]\!]$ and $[\![b_j]\!]$ for all i and j

Figure 6: Volume matching

5.2.3 Experimental results. Since the complexity of dark pool algorithms depend on the size of buy list (N) and sell list (M), following [16], we analyze these algorithms by varying N and M between 10 and 500. Moreover, since the complexity of the CDA algorithm additionally depends on the number of executed sell orders (s), we set this to be 10% of the maximum of N and M[6]. For CDA, these results are reported in Table 6 (§D) . As expected and evident from Fig. 7a, the run time of the algorithms increases with increasing N and M. However, this increase is more pronounced in the algorithm of [16] due to its sequential nature and heavy dependence on s. To capture this effect more clearly, we perform experiments with fixed $N = M = 100$ and vary s between 1 to 50, and report these results in Table 3.

As explained earlier and as is evident from Table 3, observe that the run time of CDA linearly depends on s for the algorithm of [16]. On the contrary, the parallelizations in our algorithm help in making the run time independent of s, and thereby bring up to 20× saving in run time. The poor run time of [16] in comparison to ours can also be attributed to the large number of reconstructions in the former's CDA algorithm that necessitate performing verification each time a value is reconstructed (in our $(1, 1)$-FaF setting). The improvement of our algorithm is also reflected in throughput (TP) where our algorithm's TP remains almost constant, whereas the algorithm of [16] sees a steady fall. Here, TP is computed as $1/t_o$ where t_o is the online run time of the protocol.

The results for volume matching appear in Table 4. As expected, the throughput (TP) of volume matching is better than CDA. Further, due to the parallelizations introduced by our work, our algorithm's runtime increases very slowly compared to that of [16] with increasing N, M. This is visually represented in Fig. 7a, which compares the online runtime of volume matching and CDA algorithm. Since TP for volume matching is computed as $N+M/t_o$, where t_o denotes the online run time of the protocol, the slow increase in our run time helps in obtaining higher TP as N, M increase. This is not the case for [16] whose TP remains almost constant. The gain in TP for us thus turns out to be up to 62× over the work of [16]. A visual comparison of TP for CDA and volume matching appears in Fig. 7b and Fig. 7c.

[6]Dark pools are not obligated to report the detailed information regarding volumes and types of transactions. Hence, we can only speculate the parameters such as s, N, M. Further, accounting for the recent trend of smaller traders entering into dark pools, we consider the possibility of a large volume order matched against several small volume orders and set s to be 10%. This is in contrast to the unrealistic case of $s \in \{0, 1, 2, 3\}$ as in [16]

s	Ref	Preprocessing		Online		
		Time (ms)	Com (KB)	Time (ms)	Com (KB)	TP (orders/s)
1	Ours	3.41	333.19	17.13	190.09	58.37
	[16]	2.42	158.41	16.58	79.24	60.30
2	Ours	3.25	333.19	15.98	192.90	62.57
	[16]	2.56	161.32	24.50	84.68	40.81
4	Ours	3.28	333.19	15.38	198.53	65.00
	[16]	2.55	167.13	37.22	96.11	26.87
5	Ours	3.37	333.19	15.40	201.34	64.95
	[16]	2.48	170.04	42.73	102.08	23.40
10	Ours	3.26	333.19	15.46	215.40	64.67
	[16]	2.59	184.57	75.20	134.58	13.30
40	Ours	3.33	333.19	17.16	299.78	58.26
	[16]	3.06	271.77	281.21	421.39	3.56
50	Ours	3.18	333.19	15.77	327.90	63.40
	[16]	3.13	301.12	350.70	551.95	2.85

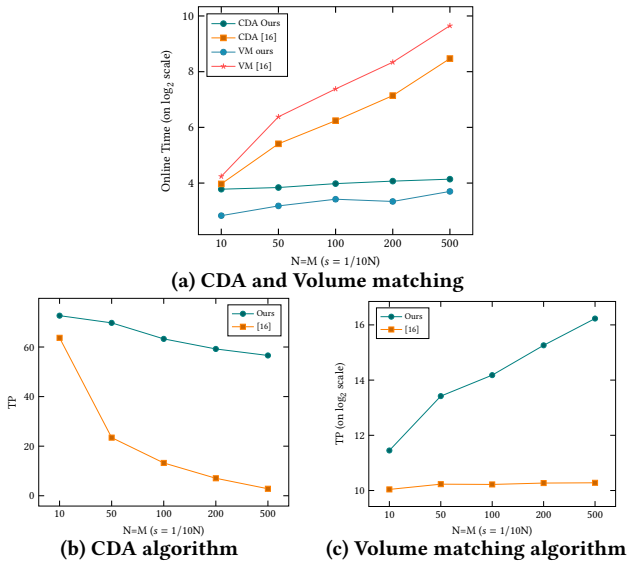
Table 3: Comparison for CDA for varying s and N=M=100.

N	M	Ref	Preprocessing		Online		
			Time (ms)	Com (KB)	Time (ms)	Com (KB)	TP (×10^3 orders/s)
10	10	Ours	1.72	47.82	7.13	32.70	2.81
		[16]	1.70	45.94	18.93	9.31	1.06
20	10	Ours	1.81	71.06	7.83	48.45	3.83
		[16]	1.89	90.54	37.61	17.96	0.80
20	20	Ours	1.92	94.30	7.86	64.20	5.09
		[16]	1.89	90.54	34.83	17.96	1.15
40	20	Ours	2.11	140.78	7.79	95.69	7.70
		[16]	2.28	179.74	66.50	35.26	0.90
50	50	Ours	2.65	233.78	9.10	158.68	10.99
		[16]	2.56	224.37	83.28	43.91	1.20
100	50	Ours	3.11	350.27	9.29	237.42	16.14
		[16]	3.50	447.69	163.73	87.17	0.92
100	100	Ours	4.03	466.55	10.77	316.15	18.57
		[16]	3.74	447.73	167.17	87.17	1.20
200	100	Ours	5.04	699.44	10.02	473.62	29.95
		[16]	6.64	875.48	326.77	173.67	0.92
200	200	Ours	7.89	932.34	10.18	631.09	39.31
		[16]	7.19	894.37	323.33	173.67	1.24
400	200	Ours	10.73	1397.75	12.20	946.03	49.18
		[16]	12.30	1787.73	640.70	346.66	0.94
500	500	Ours	26.98	2329.07	12.99	1575.92	76.98
		[16]	23.34	2234.94	803.91	433.17	1.24

Table 4: Comparison for volume matching for varying N, M.

5.3 Privacy-preserving ML (PPML)

To showcase that our FaF-secure protocols have wide applicability, we also benchmark the performance of popular neural networks



(a) CDA and Volume matching

(b) CDA algorithm **(c) Volume matching algorithm**

Figure 7: Online time (a) and TP (orders/sec) comparison (b, c) of our algorithm with [16]

in our setting. We consider a variety of network architectures, the accuracy of which follow from [43, 45, 63]. We begin with a fully connected 3 layer network (NN-1) that considers around 118K model parameters. We also consider a convolutional neural network (NN-2) comprising 2 hidden layers, with 100 and 10 nodes, respectively. Lastly, we consider the two popular deep neural networks of LeNet [40] and VGG16 [60]. LeNet comprises 2 convolutional and fully connected layers, followed by maxpool for convolutional layers, with approximately 431K parameters. On the other hand, VGG16 has 16 layers and contains fully-connected, convolutional, ReLU activation and max pool layers with around 138 million parameters. We rely on the standard MNIST [41] dataset to perform secure inference using NN-1 and LeNet, while the CIFAR-10 [38] dataset for NN-2 and VGG16 networks. The benchmarks for the different NNs appear in Table 5. As expected, the run time and communication of our protocols increase as the depth of the NNs increases from NN-1 to VGG16.

NN type	Preprocessing		Online		
	Time (s)	Com (MB)	Time (s)	Com (MB)	TP (queries/s)
NN-1	0.011	0.417	0.008	0.071	1010.86
NN-2	0.037	1.708	0.010	0.290	814.99
LeNet	0.560	35.898	0.053	6.298	152.21
VGG16	9.676	549.664	0.473	94.951	16.89

Table 5: NN inference.

6 CONCLUSION

We designed the first concretely efficient FaF-secure MPC protocol in the (1,1) 5 party setting. Further, we designed several building blocks and optimized them for the setting under consideration. Thus, we provide a comprehensive framework that allows designing secure variants of various applications where traditional security fails. We consider the specific case of dark pools and showcase that traditional MPC is a misfit for it. In the process of designing FaF-secure protocols for dark pools, we also improve the underlying algorithms and showcase it in the benchmarks. Given the popularity of PPML, we also benchmark deep neural networks.

ACKNOWLEDGEMENTS

Arpita Patra, Varsha Bhat Kukkala and Bhavish Raj Gopal 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. The authors would also like to acknowledge the support from Google Cloud for benchmarking.

REFERENCES

[1] Bar Alon, Eran Omri, and Anat Paskin-Cherniavsky. 2020. MPC with Friends and Foes. In *CRYPTO*.
[2] Toshinori Araki, Assi Barak, Jun Furukawa, Tamar Lichter, Yehuda Lindell, Ariel Nof, Kazuma Ohara, Adi Watzman, and Or Weinstein. 2017. Optimized Honest-Majority MPC for Malicious Adversaries - Breaking the 1 Billion-Gate Per Second Barrier. In *IEEE S&P*.
[3] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. 2016. High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority. In *ACM CCS*.
[4] Venkat Arun, Aniket Kate, Deepak Garg, Peter Druschel, and Bobby Bhattacharjee. 2020. Finding Safety in Numbers with Secure Allegation Escrows. In *NDSS*.
[5] Gilad Asharov, Tucker Hybinette Balch, Antigoni Polychroniadou, and Manuela Veloso. 2020. Privacy-Preserving Dark Pools. In *AAMAS*.
[6] Saikrishna Badrinarayanan, Aayush Jain, Nathan Manohar, and Amit Sahai. 2020. Secure MPC: Laziness Leads to GOD. In *ASIACRYPT*.
[7] Azer Bestavros, Andrei Lapets, and Mayank Varia. 2017. User-centric distributed solutions for privacy-preserving analytics. *Communications of ACM* (2017).
[8] Dan Bogdanov, Marko Jõemets, Sander Siim, and Meril Vaht. 2015. How the Estonian Tax and Customs Board Evaluated a Tax Fraud Detection System Based on Secure Multi-party Computation. In *FC*.
[9] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, et al. 2009. Secure multiparty computation goes live. In *FC*.
[10] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. 2019. Zero-Knowledge Proofs on Secret-Shared Data via Fully Linear PCPs. In *CRYPTO*.
[11] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. 2019. Practical Fully Secure Three-Party Computation via Sublinear Distributed Zero-Knowledge Proofs. In *ACM CCS*.
[12] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. 2020. Efficient Fully Secure Computation via Distributed Zero-Knowledge Proofs. In *ASIACRYPT*.
[13] Megha Byali, Harsh Chaudhari, Arpita Patra, and Ajith Suresh. 2020. FLASH: Fast and Robust Framework for Privacy-preserving Machine Learning. *PETS* (2020).
[14] Megha Byali, Carmit Hazay, Arpita Patra, and Swati Singla. 2019. Fast actively secure five-party computation with security beyond abort. In *ACM CCS*.
[15] Megha Byali, Arun Joseph, Arpita Patra, and Divya Ravi. 2018. Fast Secure Computation for Small Population over the Internet. In *ACM CCS*.
[16] John Cartlidge, Nigel P Smart, and Younes Talibi Alaoui. 2019. MPC joins the dark side. In *ACM ASIACCS*.
[17] John Cartlidge, Nigel P Smart, and Younes Talibi Alaoui. 2021. Multi-party computation mechanism for anonymous equity block trading: A secure implementation of turquoise plato uncross. *Intell. Syst. Account. Finance Manag.* (2021).
[18] Harsh Chaudhari, Ashish Choudhury, Arpita Patra, and Ajith Suresh. 2019. ASTRA: High Throughput 3PC over Rings with Application to Secure Prediction. In *ACM CCSW@CCS*.
[19] Harsh Chaudhari, Rahul Rachuri, and Ajith Suresh. 2020. Trident: Efficient 4PC Framework for Privacy Preserving Machine Learning. *NDSS* (2020).
[20] David Chaum. 1989. The Spymasters Double-Agent Problem: Multiparty Computations Secure Unconditionally from Minorities and Cryptographically from Majorities. In *CRYPTO*.
[21] Richard Cleve. 1986. Limits on the Security of Coin Flips when Half the Processors Are Faulty (Extended Abstract). In *ACM STOC*.
[22] Cryptography and Privacy Engineering Group at TU Darmstadt. [n. d.]. ENCRYPTO Utils. https://github.com/encryptogroup/ENCRYPTO_utils.

[23] Mariana Botelho da Gama, John Cartlidge, Antigoni Polychroniadou, Nigel P Smart, and Younes Talibi Alaoui. 2021. Kicking-the-Bucket: Fast Privacy-Preserving Trading Using Buckets. *IACR Cryptol. ePrint Arch.* (2021).

[24] Anders Dalskov, Daniel Escudero, and Marcel Keller. 2020. Fantastic Four: Honest-Majority Four-Party Secure Computation With Malicious Security. USENIX security.

[25] Ivan Damgård and Jesper Buus Nielsen. 2007. Scalable and unconditionally secure multiparty computation. In *CRYPTO*.

[26] Danny Dolev, Cynthia Dwork, Orli Waarts, and Moti Yung. 1993. Perfectly Secure Message Transmission. *J. ACM* (1993).

[27] Xiao Dong, David A Randolph, Chenkai Weng, Abel N Kho, Jennie M Rogers, and Xiao Wang. 2021. Developing High Performance Secure Multi-Party Computation Protocols in Healthcare: A Case Study of Patient Risk Stratification. In *AMIA*.

[28] Matthias Fitzi, Martin Hirt, and Ueli M. Maurer. 1998. Trading Correctness for Privacy in Unconditional Multi-Party Computation (Extended Abstract). In *CRYPTO*.

[29] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. 2017. High-Throughput Secure Three-Party Computation for Malicious Adversaries and an Honest Majority. In *EUROCRYPT*.

[30] Hossein Ghodosi and Josef Pieprzyk. 2009. Multi-Party Computation with Omnipresent Adversary. In *PKC*.

[31] Oded Goldreich. 2007. *Foundations of cryptography: volume 1, basic tools*. Cambridge university press.

[32] S. Dov Gordon, Samuel Ranellucci, and Xiao Wang. 2018. Secure Computation with Low Communication from Cross-Checking. In *ASIACRYPT*.

[33] Martin Hirt, Ueli M. Maurer, and Vassilis Zikas. 2008. MPC vs. SFE : Unconditional and Computational Security. In *ASIACRYPT*.

[34] Martin Hirt and Marta Mularczyk. 2020. Efficient MPC with a Mixed Adversary. In *ITC*.

[35] T Ryan Hoens, Marina Blanton, and Nitesh V Chawla. 2010. A private and reliable recommendation system for social networks. In *IEEE*. IEEE.

[36] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. 2021. SWIFT: Super-fast and Robust Privacy-Preserving Machine Learning. *IACR Cryptol. ePrint Arch.* (2021).

[37] Nishat Koti, Arpita Patra, Rahul Rachuri, and Ajith Suresh. 2022. Tetrad: Actively Secure 4PC for Secure Training and Inference. IACR Cryptol. ePrint Arch..

[38] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. 2014. The CIFAR-10 dataset. (2014). https://www.cs.toronto.edu/~kriz/cifar.html.

[39] Benjamin Kuykendall, Hugo Krawczyk, and Tal Rabin. 2019. Cryptography for# metoo. *PETS* (2019).

[40] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* (1998).

[41] Yann LeCun and Corinna Cortes. 2010. MNIST handwritten digit database. (2010). http://yann.lecun.com/exdb/mnist/.

[42] Yehuda Lindell. 2017. How to simulate it–a tutorial on the simulation proof technique. In *Tutorials on the Foundations of Cryptography*.

[43] Payman Mohassel and Peter Rindal. 2018. ABY^3: A Mixed Protocol Framework for Machine Learning. In *ACM CCS*.

[44] Payman Mohassel, Mike Rosulek, and Ye Zhang. 2015. Fast and Secure Three-party Computation: The Garbled Circuit Approach. In *ACM CCS*.

[45] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *IEEE S&P*.

[46] Peter Sebastian Nordholt and Meilof Veeningen. 2018. Minimising Communication in Honest-Majority MPC by Batchwise Multiplication Verification. In *ACNS*.

[47] United States of America before the Securities and Exchange Commission. 2005. SEC institutes enforcement action against 20 former New York Stock Exchange specialists alleging pervasive course of fraudulent trading. Press Release. https://www.sec.gov/news/press/2005-54.htm.

[48] United States of America before the Securities and Exchange Commission. 2011. In the Matter of Pipeline Trading Systems LLC, et al., Securities Exchange Act of 1934 Release No. 65609. https://www.sec.gov/litigation/admin/2011/33-9271.pdf.

[49] United States of America before the Securities and Exchange Commission. 2012. In the Matter of eBX, LLC Securities Exchange Act of 1934 Release No. 67979. https://www.sec.gov/litigation/admin/2012/34-67969.pdf.

[50] United States of America before the Securities and Exchange Commission. 2014. In the Matter of LavaFlow, Inc. Securities Exchange Act of 1934 Release No. 72673. https://www.sec.gov/litigation/admin/2014/34-72673.pdf.

[51] United States of America before the Securities and Exchange Commission. 2014. In the Matter of Liquidnet, Inc., Securities Exchange Act of 1934 Release No. 72339. https://www.sec.gov/litigation/admin/2014/33-9596.pdf.

[52] United States of America before the Securities and Exchange Commission. 2016. In the Matter of Credit Suisse Securities (USA) LLC, Securities Exchange Act of 1934 Release No. 77002. https://www.sec.gov/litigation/admin/2016/33-10013.pdf.

[53] United States of America before the Securities and Exchange Commission. 2018. In the Matter of ITG Inc. and Alternet Securities, Inc., Securities Exchange Act of 1934 Release No. 84548. https://www.sec.gov/litigation/admin/2018/33-10572.pdf.

[54] Satsuya Ohata and Koji Nuida. 2020. Communication-Efficient (Client-Aided) Secure Two-Party Protocols and Its Application.

[55] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. 2021. ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation. In *USENIX*, Michael Bailey and Rachel Greenstadt (Eds.).

[56] Arpita Patra and Ajith Suresh. 2020. BLAZE: Blazing Fast Privacy-Preserving Machine Learning. *NDSS* (2020).

[57] Phillip Rogaway and Thomas Shrimpton. 2004. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In *FSE*. Springer.

[58] Alex Sangers, Maran van Heesch, Thomas Attema, Thijs Veugen, Mark Wiggerman, Jan Veldsink, Oscar Bloemen, and Daniël Worm. 2019. Secure multiparty PageRank algorithm for collaborative fraud detection. In *FC*. Springer.

[59] Erez Shmueli and Tamir Tassa. 2017. Secure multi-party protocols for item-based collaborative filtering. In *ACM RecSys*.

[60] Karen Simonyan and Andrew Zisserman. 2015. Very deep convolutional networks for large-scale image recognition. *ICLR* (2015).

[61] Stanford. [n. d.]. CS231n: Convolutional Neural Networks for Visual Recognition. https://cs231n.github.io/convolutional-networks/

[62] Christina-Angeliki Toli, Abdelrahaman Aly, and Bart Preneel. 2016. A privacy-preserving model for biometric fusion. In *CANS*.

[63] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. 2021. FALCON: Honest-Majority Maliciously Secure Framework for Private Deep Learning. *PoPETs* (2021).

A PRELIMINARIES

Security model. We prove the security of our protocols following the standard ideal-world/real-world simulation paradigm [31, 42]. In this security notion, an ideal functionality $\mathcal{F}$ is considered, to which the corrupted and uncorrupted parties send their inputs over a perfectly secure channel. $\mathcal{F}$ executes the computation and sends the output to all. Informally, a protocol is said to be secure if whatever the adversary can do in the real world can also be done in the ideal world. In the traditional 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 [1], the additional requirement of simulating the view of any subset of uncorrupted (or semi-honest) parties necessitates the use of two simulators. Thus, to prove the security, two simulators are constructed in the ideal-world, one for the malicious adversary and one for the semi-honest adversary. Further, the malicious adversary is allowed to send its entire view to the semi-honest adversary in the ideal world (to capture the behaviour where the malicious adversary may send non-protocol messages to uncorrupted parties in the real world).

Let $\mathcal{A}$ denote the probabilistic polynomial time (PPT) real-world malicious adversary 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 real-world semi-honest adversary corrupting $h^{\star}$ parties in $\mathcal{H} \subset \mathcal{P} \setminus \mathcal{I}$, and $\mathcal{S}_{\mathcal{A},\mathcal{H}}$, be the ideal-world simulator. Let $\mathcal{F}$ be the ideal-world functionality. Let $\text{VIEW}^{\text{REAL}}_{\mathcal{A},\Pi}$ be $\mathcal{A}$'s view and $\text{OUT}^{\text{REAL}}_{\mathcal{A},\Pi}$ denote the output of the uncorrupted parties (in $\mathcal{P} \setminus \mathcal{I}$) during a random execution of a protocol Π. Correspondingly, let $\text{VIEW}^{\text{REAL}}_{\mathcal{A},\mathcal{A}_{\mathcal{H}},\Pi}$ be $\mathcal{A}_{\mathcal{H}}$'s view during an execution of protocol Π running alongside $\mathcal{A}$. Note that $\text{VIEW}^{\text{REAL}}_{\mathcal{A},\mathcal{A}_{\mathcal{H}},\Pi}$ consists of the non-protocol messages sent by the $\mathcal{A}$ to $\mathcal{A}_{\mathcal{H}}$. Similarly, let $\text{VIEW}^{\text{IDEAL}}_{\mathcal{A},\mathcal{F}}$ be the malicious adversary's simulated view and $\text{OUT}^{\text{IDEAL}}_{\mathcal{A},\mathcal{F}}$ denote the output of the uncorrupted parties during a random execution of ideal-world functionality $\mathcal{F}$. Further, let $\text{VIEW}^{\text{IDEAL}}_{\mathcal{A},\mathcal{A}_{\mathcal{H}},\mathcal{F}}$ be the

semi-honest adversary's simulated view during an execution of $\mathcal{F}$ running alongside $\mathcal{A}$.

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

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

Shared key setup. Following several recent works [2, 3, 13, 19, 36, 43, 56], to enable non-interactive communication between the parties, a one-time setup is performed that establishes common random keys for a pseudo-random function (PRF) F. Here $F : \{0,1\}^\kappa \times \{0,1\}^\kappa \to X$ is a secure PRF, with co-domain X being $\mathbb{Z}_{2^\ell}$.

The key setup is modeled via a functionality $\mathcal{F}_{\text{setup}}$ (Fig. 8) that can be realized using any FaF-secure MPC protocol. The goal is to establish a common key between every set of 2, 3, 4, and all parties.

To sample a random value $\mathsf{r} \in \mathbb{Z}_{2^\ell}$ among a set of 3 parties P_i, P_j, P_k non-interactively, each of these parties invoke $F_{k_{ijk}}(id_{ijk})$ and obtain r. Here, id_{ijk} denotes a counter maintained by these three parties, and is updated after every PRF invocation. The appropriate keys used to sample the common randomness is implicit from the context and from the identities of the parties that sample.

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

$\mathcal{F}_{\text{setup}}$ interacts with the parties in $\mathcal{P}$ and the adversaries $\mathcal{S}_{\mathcal{A}}, \mathcal{S}_{\mathcal{A},\mathcal{H}}$. $\mathcal{F}_{\text{setup}}$ picks the following keys.

- A common random key $k_{\mathcal{P}}$ for all the parties.
- A common key k_{ij} between every pair of parties P_i, P_j where $1 \le i < j \le 5$.
- A common key k_{ijk} between every set of 3 parties P_i, P_j, P_k where $1 \le i < j < k \le 5$.
- A common key k_{ijkl} between every set of 4 parties P_i, P_j, P_k, P_l where $1 \le i < j < k < l \le 5$.

Output: Keys $\{k_{\mathcal{P}}, k_{si}, k_{js}, k_{sjk}, k_{isk}, k_{ijs}, k_{sjkl}, k_{iskl}, k_{ijsl}, k_{ijks}\}$, generated as above, are output to every $P_s \in \mathcal{P}$.

Figure 8: Ideal functionality for shared-key setup

Collision-resistant hash. A family of hash functions [57] $\{\mathsf{H} : \mathcal{K} \times \mathsf{M} \to \mathcal{Y}\}$ is called collision resistant if for all probabilistic polynomial time adversaries $\mathcal{A}$, given the hash function H_k for $k \in_R \mathcal{K}$, the following holds: $\Pr[(x, x') \leftarrow \mathcal{A}(k) : (x \neq x') \wedge \mathsf{H}_k(x) = \mathsf{H}_k(x')] = \text{negl}(\kappa)$, where $x, x' \in \{0,1\}^m$, $m = \text{poly}(\kappa)$, and κ is security parameter.

Outsourced setting. In this setting, the required computation (e.g., matching for dark pools) is outsourced to external servers. Since these servers are external to the system, it is required that all the information regarding the system (dark pools) must be hidden from the servers. Hence, it is also essential to consider how the data owner or client would share his private data among the external servers. Thus, we elaborate on the agreement protocol executed among the computing parties (servers) to agree on the value sent by the client U. At a high-level, the protocol proceeds as follows. Let β_{v_i} denote the value received by P_i from U. To agree on β_{v} received from U, the parties first arrive on an agreement regarding each β_{v_i} received by P_i. This is followed by selecting the majority value among $\beta_{\mathsf{v}_1}, \beta_{\mathsf{v}_2}, \beta_{\mathsf{v}_3}, \beta_{\mathsf{v}_4}, \beta_{\mathsf{v}_5}$. For parties to agree on β_{v_i}, P_i first sends β_{v_i} to all other parties. This is followed by $P_j \in \mathcal{P} \setminus P_i$ exchanging β_{v_i} among themselves. Thus, each $P_j \in \mathcal{P} \setminus P_i$ receives four versions of β_{v_i} and sets the majority value among the four values received as β_{v_i}. Since there can be at most one malicious corruption among the parties, the majority rule ensures that all honest parties are on the same page. Once each of the values are agreed on, every party takes the majority among $\beta_{\mathsf{v}_1}, \beta_{\mathsf{v}_2}, \beta_{\mathsf{v}_3}, \beta_{\mathsf{v}_4}, \beta_{\mathsf{v}_5}$ as the value sent by U. If no value appears in majority, the malicious intent of the client is captured and hence the input is discarded.

B 5PC $(1,1)$-FAF SECURE PROTOCOLS

B.1 Joint message passing (jmp)

The modified protocol for jmp appears in Fig. 9. The protocol is described with respect to a single message v for a fixed ordered pair of senders and a given receiver. However, we note that *verify* phase across several messages for the same ordered pair of senders and receiver can be bundled together. This would involve party P_j (silent party) sending a single hash corresponding to all the messages under consideration and performing the verification accordingly.

Protocol $\Pi_{\text{jmp}}(P_i, P_j, P_k, \mathsf{v})$

Each party P_s for $s \in \{i, j, k\}$ initializes bit $\mathsf{b}_s = 0$. Let CP denote the conflict pair which is the pair of parties in conflict, one of which is guaranteed to be corrupt. Let P_i, P_j denote the senders who wish to send v to receiver P_k. Let H denote a collision-resistant hash function.

Send Phase: P_i sends v to P_k.
Verify Phase: P_j sends $\mathsf{H}(\mathsf{v})$ to P_k.

– P_k broadcasts "(accuse, P_i)", if P_i is silent, and all take $\text{CP} = (P_i, P_k)$ as the conflict pair. Analogously for P_j. If P_k accuses both P_i, P_j, then $\text{CP} = (P_i, P_k)$. Otherwise, P_k receives some $\tilde{\mathsf{v}}$ and either sets $\mathsf{b}_k = 0$ when the value and the hash are consistent or sets $\mathsf{b}_k = 1$. P_k then sends b_k to P_i, P_j and terminates if $\mathsf{b}_k = 0$.

– If P_i does not receive a bit from P_k, it broadcasts "(accuse, P_k)" and $\text{CP} = (P_i, P_k)$. Analogously for P_j. If both P_i, P_j accuse P_k, then $\text{CP} = (P_i, P_k)$. Otherwise, P_s for $s \in \{i, j\}$ sets $\mathsf{b}_s = \mathsf{b}_k$.

– P_i, P_j exchange their bits with each other. If P_i does not receive b_j from P_j, it broadcasts "(accuse, P_j)" and $\text{CP} = (P_i, P_j)$. Analogously for P_j. Otherwise, P_i resets its bit to $\mathsf{b}_i \vee \mathsf{b}_j$ and likewise P_j resets its bit to $\mathsf{b}_j \vee \mathsf{b}_i$.

– P_s for $s \in \{i, j, k\}$ broadcasts $\mathsf{H}_s = \mathsf{H}(\mathsf{v}^*)$ if $\mathsf{b}_s = 1$, where $\mathsf{v}^* = \mathsf{v}$ for $s \in \{i, j\}$ and $\mathsf{v}^* = \tilde{\mathsf{v}}$ otherwise. If P_k does not broadcast, terminate. If either P_i or P_j does not broadcast, then $\text{CP} = (P_i, P_j)$. Otherwise,

- If $\mathsf{H}_i \neq \mathsf{H}_j$: $\text{CP} = (P_i, P_j)$.
- Else if $\mathsf{H}_i \neq \mathsf{H}_k$: $\text{CP} = (P_i, P_k)$.
- Else if $\mathsf{H}_i = \mathsf{H}_j = \mathsf{H}_k$: $\text{CP} = (P_j, P_k)$.

Figure 9: Joint Message Passing Protocol

B.2 Input sharing

The protocol for Π_{sh} appears in Fig. 10.

Protocol $\Pi_{\mathsf{sh}}(P_i, \mathsf{v})$

Preprocessing:

– Parties non-interactively generate $[\cdot]$-shares of a random $\alpha_{\mathsf{v}} \in \mathbb{Z}_{2^\ell}$ such that P_i learns all shares of α_{v}, using the shared-keys.

Online:

– P_i computes and sends $\beta_{\mathsf{v}} = \mathsf{v} + \alpha_{\mathsf{v}}$ to one other party, say P_j.

– P_i, P_j then jmp-sv β_{v} to all other parties.

Figure 10: Generating $[\![\mathsf{v}]\!]$ by party P_i

Protocol for generating $[\cdot]$-shares appears in Fig. 11.

Protocol $\Pi_{\mathsf{RSS-Sh}}(P_i, \mathsf{v})$

Let v_{lm} be a share of v held by $P_i, P_j, P_k \in \mathcal{P}$.

– Parties in $\mathcal{P} \setminus \{P_p, P_q\}$ for $1 \le p < q \le 5$ and $p \ne l, q \ne m$, non-interactively generate $\mathsf{v}_{pq} \in \mathbb{Z}_{2^\ell}$ together with P_i, using the shared-key setup.

– P_i computes and sends $\mathsf{v}_{lm} = \mathsf{v} - \sum_{1\le p<q\le 5, p\ne l, q\ne m} \mathsf{v}_{pq}$ to P_j, following which P_i, P_j jmp-sv v_{lm} to P_k.

Figure 11: Generating $[\mathsf{v}]$ by party P_i

B.3 Joint sharing

Here we discuss the various optimizations possible in the joint sharing protocol. When the value to be shared is available with P_i, P_j in the preprocessing phase, the protocol can be optimized as follows. All parties set $\beta_{\mathsf{v}} = 0$. P_i, P_j, P_k non-interactively sample a random $\mathsf{r}_{lm} \in \mathbb{Z}_{2^\ell}$ and set the common $[\cdot]$-share of α_{v} they possess as $\alpha_{\mathsf{v}_{\mathsf{lm}}} = \mathsf{r}_{lm}$. Similarly, P_i, P_j, P_l non-interactively sample a random $\mathsf{r}_{km} \in \mathbb{Z}_{2^\ell}$ and set the common $[\cdot]$-share of α_{v} they possess as $\alpha_{\mathsf{v}_{\mathsf{km}}} = \mathsf{r}_{km}$. P_i, P_j set the common share of α_{v} held together with P_m as $\alpha_{\mathsf{v}_{\mathsf{kl}}} = -(\mathsf{v} + \mathsf{r}_{lm} + \mathsf{r}_{km})$ and jmp-sv $\alpha_{\mathsf{v}_{\mathsf{kl}}}$ to P_m. The other $[\cdot]$-shares of α_{v} are set as 0.

When the value to be shared is held by three parties, say P_i, P_j, P_k, the protocol proceeds similarly to Π_{JSh2}, with the following difference– in the preprocessing phase, α_{v} will be also be learned by P_k, and in the online phase, only two jmp-sv are required. We call the resultant protocol Π_{JSh3}, and omit the formal protocol due to its close resemblance to Π_{JSh2}. Moreover, when the value is available with these three parties in the preprocessing phase, the protocol can be made completely non-interactive. For this, similar to the previous case, β_{v} is set as 0, and the common $[\cdot]$-share of α_{v} held by P_i, P_j, P_k is set as $-\mathsf{v}$ and all other shares are set as 0.

Finally, when all parties hold a value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$, they can generate $[\![\mathsf{v}]\!]$ by setting $\beta_{\mathsf{v}} = \mathsf{v}$ and all $[\cdot]$-shares of α_{v} as 0.

Protocol $\Pi_{\mathsf{JSh2}}(P_i, P_j, \mathsf{v})$

Preprocessing:

– Parties non-interactively generate $[\cdot]$-shares of a random $\alpha_{\mathsf{v}} \in \mathbb{Z}_{2^\ell}$ such that P_i, P_j learn all shares of α_{v}, using the shared keys.

Online:

– P_i, P_j compute and jmp-sv $\beta_{\mathsf{v}} = \mathsf{v} + \alpha_{\mathsf{v}}$ to all other parties.

Figure 12: Joint sharing of v by P_i, P_j

B.4 Reconstruction

The protocol for reconstruction appears in Fig. 13.

Protocol $\Pi_{\mathsf{rec}}(P_i, \mathsf{v})$

Let the missing shares at P_i be $\mathsf{v}_{ij}, \mathsf{v}_{ik}, \mathsf{v}_{il}, \mathsf{v}_{im}$.

– Let P_k, P_l, P_m possess v_{ij}. P_k, P_l send v_{ij} to P_i while P_m sends its hash to P_i. Analogous steps are carried out for the other three shares.

– P_i uses the value which appears in majority for the received missing shares, together with its own shares, for reconstructing v as $\mathsf{v} = \sum_{1\le p<q\le 5} \mathsf{v}_{pq}$.

Figure 13: Reconstruction of v towards P_i

C BUILDING BLOCKS

Bit to arithmetic. The protocol appears in Fig. 14.

Protocol $\Pi_{\mathsf{bit2A}}(\mathcal{P}, [\![\mathsf{b}]\!]^{\mathsf{B}})$

Preprocessing:

– P_1, P_2 jointly share $\nu_1 = (\alpha_{\mathsf{b}_{34}} \oplus \alpha_{\mathsf{b}_{35}} \oplus \alpha_{\mathsf{b}_{45}})$, P_1, P_3 jointly share $\nu_2 = (\alpha_{\mathsf{b}_{24}} \oplus \alpha_{\mathsf{b}_{25}})$, P_2, P_3 jointly share $\nu_3 = (\alpha_{\mathsf{b}_{14}} \oplus \alpha_{\mathsf{b}_{15}})$ and P_4, P_5 jointly share $\nu_4 = (\alpha_{\mathsf{b}_{12}} \oplus \alpha_{\mathsf{b}_{13}} \oplus \alpha_{\mathsf{b}_{23}})$ to generate $[\![\nu_1^{\mathsf{R}}]\!], [\![\nu_2^{\mathsf{R}}]\!], [\![\nu_3^{\mathsf{R}}]\!], [\![\nu_4^{\mathsf{R}}]\!]$, respectively.

– Parties execute Π_{mult} on $([\![\nu_1^{\mathsf{R}}]\!], [\![\nu_2^{\mathsf{R}}]\!])$ and $([\![\nu_3^{\mathsf{R}}]\!], [\![\nu_4^{\mathsf{R}}]\!])$ to generate $[\![\nu_1^{\mathsf{R}} \cdot \nu_2^{\mathsf{R}}]\!]$ and $[\![\nu_3^{\mathsf{R}} \cdot \nu_4^{\mathsf{R}}]\!]$, respectively.

– Parties non-interactively compute $[\![\mathsf{p}^{\mathsf{R}}]\!] = [\![(\nu_1 \oplus \nu_2)^{\mathsf{R}}]\!] = [\![\nu_1^{\mathsf{R}}]\!] + [\![\nu_2^{\mathsf{R}}]\!] - 2 \cdot [\![\nu_1^{\mathsf{R}} \cdot \nu_2^{\mathsf{R}}]\!]$ and $[\![\mathsf{q}^{\mathsf{R}}]\!] = [\![(\nu_3 \oplus \nu_4)^{\mathsf{R}}]\!] = [\![\nu_3^{\mathsf{R}}]\!] + [\![\nu_4^{\mathsf{R}}]\!] - 2 \cdot [\![\nu_3^{\mathsf{R}} \cdot \nu_4^{\mathsf{R}}]\!]$.

– Parties execute Π_{mult} on $[\![\mathsf{p}^{\mathsf{R}}]\!], [\![\mathsf{q}^{\mathsf{R}}]\!]$ to generate $[\![\mathsf{p}^{\mathsf{R}} \cdot \mathsf{q}^{\mathsf{R}}]\!]$, and compute $[\![\alpha_{\mathsf{b}}{}^{\mathsf{R}}]\!] = [\![(\mathsf{p} \oplus \mathsf{q})^{\mathsf{R}}]\!] = [\![\mathsf{p}^{\mathsf{R}}]\!] + [\![\mathsf{q}^{\mathsf{R}}]\!] - 2 \cdot [\![\mathsf{p}^{\mathsf{R}} \cdot \mathsf{q}^{\mathsf{R}}]\!]$.

– Parties non-interactively generate $[\![\mathsf{r}]\!]$ for $\mathsf{r} \in \mathbb{Z}_{2^\ell}$, and invoke $\Pi_{[\![\cdot]\!]\to[\cdot]}$ (§3.1) to generate $[\alpha_{\mathsf{b}}{}^{\mathsf{R}}]$, $[\mathsf{r}]$.

Online:

– Compute $[\mathsf{b}^{\mathsf{R}} + \mathsf{r}] = \beta_{\mathsf{b}}{}^{\mathsf{R}} + [\alpha_{\mathsf{b}}{}^{\mathsf{R}}] - 2\beta_{\mathsf{b}}{}^{\mathsf{R}} [\alpha_{\mathsf{b}}{}^{\mathsf{R}}] + [\mathsf{r}]$, and reconstruct $\mathsf{b}^{\mathsf{R}} + \mathsf{r}$ towards all, similar to multiplication.

– Non-interactively generate $[\![\mathsf{b}^{\mathsf{R}} + \mathsf{r}]\!]$ (§B.3).

– Non-interactively compute $[\![\mathsf{b}^{\mathsf{R}}]\!] = [\![\mathsf{b}^{\mathsf{R}} + \mathsf{r}]\!] - [\![\mathsf{r}]\!]$.

Figure 14: Bit to arithmetic conversion

Note that the preprocessing phase can be optimized further. Instead of invoking the entire Π_{mult} in the preprocessing phase which requires communicating 14ℓ elements, we can generate the required multiplicative terms by invoking $\mathcal{F}_{\mathsf{MulPre}}$ (whose current realization via the modified variant of [12] as described in §F, requires 6ℓ elements). For this, $[\![\cdot]\!]$-shares of $\nu_1, \nu_2, \nu_3, \nu_4$ are converted to $[\cdot]$-shares by invoking $\Pi_{[\![\cdot]\!]\to[\cdot]}$, followed by invoking $\mathcal{F}_{\mathsf{MulPre}}$ on the respective terms. The result of multiplication, generated as $[\cdot]$-shares, can be converted to $[\![\cdot]\!]$-shares by invoking $\Pi_{[\cdot]\to[\![\cdot]\!]}$.

D SECURE PROTOCOLS FOR CDA

In protocol, Π_{Insert} (Fig. 15), for insertion phase of CDA, we note that each of the f_i's for $i \in \{1, 2, \dots\}$ can be computed in parallel. Subsequently, so can f_i'''s followed by f_i''''s.

Protocol $\Pi_{\mathsf{Insert}}\left(\mathcal{P}, (\llbracket name_0^b \rrbracket, \llbracket b_0 \rrbracket, \llbracket q_0 \rrbracket), \mathcal{B}\right)$

- Insert $(\llbracket 0 \rrbracket, \llbracket 0 \rrbracket, \llbracket 0 \rrbracket)$ to the end of $\mathcal{B}$
- Compute $\llbracket f_0 \rrbracket^{\mathbf{B}} = \Pi_{\mathsf{comp}}(\mathcal{P}, \llbracket q_0 \rrbracket, \llbracket q_1 \rrbracket))$
- $\llbracket f_0 \rrbracket = \Pi_{\mathsf{bit2A}}(\llbracket f_0 \rrbracket^{\mathbf{B}})$
- For $i = 1$ to $M + 1$ do
  - $\llbracket f_i \rrbracket^{\mathbf{B}} = \Pi_{\mathsf{comp}}(\mathcal{P}, \llbracket q_0 \rrbracket, \llbracket q_i \rrbracket + 1))$
  - $\llbracket f_i \rrbracket = \Pi_{\mathsf{bit2A}}(\llbracket f_i \rrbracket^{\mathbf{B}})$
  - $\llbracket f_i' \rrbracket = (1 - \llbracket f_i \rrbracket) \cdot \llbracket f_{i-1} \rrbracket)$
  - $\llbracket f_i'' \rrbracket = (1 - \llbracket f_i \rrbracket) \cdot (1 - \llbracket f_i' \rrbracket)$
- For $i = 1$ to $M + 1$ do in parallel
  - $\llbracket name_i'^b \rrbracket = \llbracket f_i \rrbracket \cdot \llbracket name_i^b \rrbracket + \llbracket f_i' \rrbracket \cdot \llbracket name_0^b \rrbracket + \llbracket f_i'' \rrbracket \cdot \llbracket name_{i-1}^b \rrbracket$
  - $\llbracket b_i' \rrbracket = \llbracket f_i \rrbracket \cdot \llbracket b_i \rrbracket + \llbracket f_i' \rrbracket \cdot \llbracket b_0 \rrbracket + \llbracket f_i'' \rrbracket \cdot \llbracket b_{i-1} \rrbracket$
  - $\llbracket q_i' \rrbracket = \llbracket f_i \rrbracket \cdot \llbracket q_i \rrbracket + \llbracket f_i' \rrbracket \cdot \llbracket q_0 \rrbracket + \llbracket f_i'' \rrbracket \cdot \llbracket q_{i-1} \rrbracket$

Figure 15: Obliviously inserting into buy list

The instructions in Π_{CDA} are all sequential.

Protocol $\Pi_{\mathsf{CDA}}\left(\mathcal{P}, (\llbracket name_0^b \rrbracket, \llbracket b_0 \rrbracket, \llbracket q_0 \rrbracket), \mathcal{B}, \mathcal{S}\right)$

- Invoke Π_{PSL} on $(\llbracket name_0^b \rrbracket, \llbracket b_0 \rrbracket, \llbracket q_0 \rrbracket)$
- Compute $\llbracket e \rrbracket^{\mathbf{B}} = \Pi_{\mathsf{eq}}(\mathcal{P}, \llbracket b_0 \rrbracket, 0)$
- Compute $\llbracket q_0 \rrbracket = \Pi_{\mathsf{sel}}(\llbracket q_0 \rrbracket, \llbracket 0 \rrbracket, \llbracket e \rrbracket^{\mathbf{B}})$
- Invoke Π_{Insert} to insert $(\llbracket name_0^b \rrbracket, \llbracket b_0 \rrbracket, \llbracket q_0 \rrbracket)$ into buy list

Figure 16: Overall CDA

The time and communication of the secure protocol for CDA algorithm, ours as well as that given in [16], is reported in Table 6.

N	M	Ref	Preprocessing		Online		
			Time (ms)	Com (KB)	Time (ms)	Com (KB)	TP (orders/s)
10	10	Ours	1.67	37.36	13.76	26.61	72.65
		[16]	1.62	24.15	15.70	15.41	63.71
20	10	Ours	1.73	52.28	14.41	36.52	69.38
		[16]	1.70	41.97	23.88	27.95	41.87
20	20	Ours	1.82	70.19	14.63	47.58	68.37
		[16]	1.70	41.97	22.69	27.95	44.06
40	20	Ours	1.94	100.02	14.60	67.39	68.52
		[16]	1.87	77.61	37.19	53.56	26.89
50	50	Ours	2.28	168.68	14.34	110.47	69.74
		[16]	1.98	95.44	42.64	66.62	23.45
100	50	Ours	2.73	243.27	15.10	159.99	66.23
		[16]	2.43	184.56	75.54	134.58	13.24
100	100	Ours	3.25	333.19	15.80	215.40	63.28
		[16]	2.66	184.57	75.61	134.58	13.23
200	100	Ours	4.09	482.37	16.73	314.45	59.78
		[16]	3.53	363.10	143.25	283.62	6.98
200	200	Ours	5.74	662.14	16.89	425.26	59.22
		[16]	4.26	363.14	141.81	283.62	7.05
400	200	Ours	7.73	960.83	17.58	623.36	56.90
		[16]	7.04	720.10	281.36	634.16	3.55
500	500	Ours	18.95	1648.69	17.67	1054.63	56.59
		[16]	10.87	898.78	354.43	835.64	2.82

Table 6: Comparison for CDA for varying N, M, and s = 1/10(max(N, M)).

E SECURITY OF OUR PROTOCOLS

The simulation based security proofs for our protocols are presented in this section. The simulations for 5PC are provided in the $(\mathcal{F}_{\mathsf{setup}}, \mathcal{F}_{\mathsf{jmp}})$-hybrid model. The ideal functionality, $\mathcal{F}_{\mathsf{jmp}}$ appears in Fig. 17. The two simulators considered are $\mathcal{S}_{\mathcal{A}}$ and $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ which denote the ideal-world malicious adversary and the ideal-world semi-honest adversary, respectively. We let $\mathcal{S}_{\mathcal{A}}^{P_i}$ denote the malicious simulator when party P_i is maliciously corrupt and $\mathcal{S}_{\mathcal{A},\mathcal{H}}^{P_j}$ denote the simulator for the semi-honest corruption of party P_j. We omit the superscript when it is understood from the context.

We use the following strategy for simulating the computation of a function f. The simulation begins with the simulator emulating the shared-key setup $\mathcal{F}_{\mathsf{setup}}$ functionality and giving the respective keys to the adversary. This is followed by the input sharing phase in which $\mathcal{S}_{\mathcal{A}}$ obtains the input of $\mathcal{A}$, using the known keys, and sets the inputs of the honest parties to be 0. Note the $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ already knows the inputs of $\mathcal{A}_{\mathcal{H}}$. Since $\mathcal{S}_{\mathcal{A}}$ knows all the inputs, it can honestly carry out the computation and obtain all the intermediate values as required for simulating the view of $\mathcal{A}$. Further, on invoking the ideal functionality $\mathcal{F}_{\mathsf{5PC-FaF}}$ with $\mathcal{A}$'s input (and $\mathcal{A}_{\mathcal{H}}$'s input), $\mathcal{S}_{\mathcal{A}}$ can obtain the output of the function. $\mathcal{S}_{\mathcal{A}}$ proceeds to simulate the various sub-protocols in topological order using the aforementioned values (inputs of $\mathcal{A}$ ($\mathcal{A}_{\mathcal{H}}$), intermediate values and circuit output). A similar approach is taken by $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ while ensuring that the messages sent to $\mathcal{A}_{\mathcal{H}}$ are consistent with that in the view received from $\mathcal{S}_{\mathcal{A}}$.

The simulation steps are provided separately for the sub-protocols to ensure modularity. Carrying out these simulation steps in the respective order results in simulating the computation of the desired function f. While emulating $\mathcal{F}_{\mathsf{jmp}}$, if a CP is identified, the simulator stops the simulation at that step, and continues with simulation of 3PC using the respective semi-honest 3PC simulator.

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

$\mathcal{F}_{\mathsf{jmp}}$ interacts with parties in $\mathcal{P}$ and adversary $\mathcal{S}_{\mathcal{A}}$ and $\mathcal{S}_{\mathcal{A},\mathcal{H}}$.

- $\mathcal{F}_{\mathsf{jmp}}$ receives $(\mathsf{Input}, \mathsf{v}_s)$ from P_s for $s \in \{i, j\}$, while it receives $(\mathsf{Select}, \mathsf{CP})$ from $\mathcal{S}_{\mathcal{A}}$. Here, CP denotes the pair of parties that $\mathcal{S}_{\mathcal{A}}$ wishes to choose as the conflict pair. Let $P^{\star} \in \mathcal{P}$ denote the party corrupted by $\mathcal{S}_{\mathcal{A}}$.
- If $\mathsf{v}_i = \mathsf{v}_j$ and $\mathsf{CP} = \perp$, then set $\mathsf{msg}_i = \mathsf{msg}_j = \perp, \mathsf{msg}_k = \mathsf{v}_i$.
- Else, if $P^{\star} \in \mathsf{CP}$, then set $\mathsf{msg}_i = \mathsf{msg}_j = \mathsf{msg}_k = \mathsf{CP}$.
- Else, set $\mathsf{CP} = \{P^{\star}, P\}$ where $P \in \mathsf{CP}$. Set $\mathsf{msg}_i = \mathsf{msg}_j = \mathsf{msg}_k = \mathsf{CP}$
- Send $(\mathsf{Output}, \mathsf{msg}_s)$ to $P_s \in \mathcal{P}$.

$\mathcal{S}_{\mathcal{A}}$ sends its view to $\mathcal{S}_{\mathcal{A},\mathcal{H}}$.

Figure 17: Ideal functionality for jmp

E.1 Simulations for 5PC protocols

In this section, we describe the simulation steps for input sharing, multiplication and reconstruction, followed by the complete 5PC.

E.1.1 Sharing. The ideal functionality for Π_{sh} (Fig. 10) appears in Fig. 18.

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

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

- Receive $(\mathsf{Input}, \mathsf{v})$ from dealer $P_d \in \mathcal{P}$. Let $P^\star$ be the party corrupted by $\mathcal{S}_{\mathcal{A}}$.
- Receive continue or abort with $(\mathsf{Select}, \mathsf{C})$ from $\mathcal{S}_{\mathcal{A}}$. Here, C denotes pair of parties that $\mathcal{S}_{\mathcal{A}}$ wants to choose as conflict pair.
- If received continue, randomly pick $\alpha_{\mathsf{v}_{\mathsf{ij}}} \in \mathbb{Z}_{2^\ell}$, for $1 \le i < j \le 5$ and compute $\beta_{\mathsf{v}} = \mathsf{v} + \sum_{1 \le i < j \le 5} \alpha_{\mathsf{v}_{\mathsf{ij}}}$. Set $\mathsf{msg}_s = (\beta_{\mathsf{v}}, \{\alpha_{\mathsf{v}_{\mathsf{ij}}}\}_{i \ne s, j \ne s})$, for each $P_s \in \mathcal{P}$.
- Else if received abort, then:
- If $P^\star \in \mathsf{C}$, then set $\mathsf{CP} = \mathsf{C}$ and $\mathsf{msg}_s = \mathsf{CP}$ for each $P_s \in \mathcal{P}$.
- Else set CP to include $P^\star$ and one other party from $\mathcal{P}$, and $\mathsf{msg}_s = \mathsf{CP}$ for each $P_s \in \mathcal{P}$.

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

- $\mathcal{S}_{\mathcal{A}}$ sends it's view to $\mathcal{S}_{\mathcal{A},\mathcal{H}}$.

Figure 18: Ideal functionality for Π_{sh}

The simulator for the sharing protocol appears in Fig. 19.

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

Malicious Simulation:

Preprocessing:

- $\mathcal{S}_{\mathcal{A}}$ emulates $\mathcal{F}_{\mathsf{setup}}$ and gives the respective keys to $\mathcal{A}$. The shares of α_{v} that are held by $\mathcal{A}$ are sampled non-interactively using the shared keys. Other values ($\alpha_{\mathsf{v}_{\mathsf{ij}}}$ for $1 \le i < j \le 5$ and $\alpha_{\mathsf{v}_{\mathsf{ji}}}$ for $1 \le j < i \le 5$), not known to P_i, are sampled randomly.

Online:

- If P_i is the dealer, $\mathcal{S}_{\mathcal{A}}$ receives β_{v} from $\mathcal{A}$. Given the knowledge of all shares of α_{v}, $\mathcal{S}_{\mathcal{A}}$ obtains $\mathcal{A}$'s input as $\mathsf{v} = \beta_{\mathsf{v}} - \alpha_{\mathsf{v}}$. Following this, $\mathcal{S}_{\mathcal{A}}$ emulates $\mathcal{F}_{\mathsf{jmp}}$ with $\mathcal{A}$ as one of the senders, to deliver β_{v} to all parties. Depending on $\mathcal{A}$'s behaviour, $\mathcal{S}_{\mathcal{A}}$ sets CP and invokes $\mathcal{F}_{\mathsf{sh}}$ with $(\mathsf{Input}, \mathsf{v})$, and continue/abort and $(\mathsf{Select}, \mathsf{CP})$.
- Else, $\mathcal{S}_{\mathcal{A}}$ honestly generates β_{v} by setting the input, v, of honest dealer as $\mathsf{v} = 0$. $\mathcal{S}_{\mathcal{A}}$ either sends β_{v} to $\mathcal{A}$ and/or emulates $\mathcal{F}_{\mathsf{jmp}}$ to deliver β_{v} to all, with $\mathcal{A}$ either as the sender or receiver, depending on the identity of P_i. Depending on $\mathcal{A}$'s behaviour, $\mathcal{S}_{\mathcal{A}}$ sets CP and invokes $\mathcal{F}_{\mathsf{sh}}$ with continue or abort, and $(\mathsf{Select}, \mathsf{CP})$.

Semi-Honest Simulation:

Preprocessing:

- $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ receives the shared keys generated during $\mathcal{F}_{\mathsf{setup}}$ from $\mathcal{S}_{\mathcal{A}}$, and the corresponding shares of α_{v}. The shares of α_{v} that are held by $\mathcal{A}_{\mathcal{H}}$, other than the ones held by $\mathcal{A}$, are sampled non-interactively using the shared keys. Shares not known to P_j are sampled randomly.

Online:

- If P_i is the dealer, $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ sends the β_{v} received from $\mathcal{S}_{\mathcal{A}}$ to $\mathcal{A}_{\mathcal{H}}$ and/or emulates $\mathcal{F}_{\mathsf{jmp}}$. Else, it performs these steps with a β_{v} generated by setting $\mathsf{v} = 0$.

Figure 19: Simulator for Π_{sh} for sharing v

Lemma E.1 (Security). *Protocol Π_{sh} (Fig. 10) realizes $\mathcal{F}_{\mathsf{sh}}$ (Fig. 18) with computational security in the $(\mathcal{F}_{\mathsf{setup}}, \mathcal{F}_{\mathsf{jmp}})$-hybrid model against FaF adversaries $\mathcal{S}_{\mathcal{A}}, \mathcal{S}_{\mathcal{A},\mathcal{H}}$ controlling P_i, P_j respectively.*

Proof. Claim 1: the view generated by $\mathcal{S}_{\mathcal{A}}^{P_i}$ is indistinguishable from $\mathcal{A}$'s real-world view.

This is argued as follows. When P_i is the dealer, $\mathcal{A}$'s view consists of the random shares of α_{v} generated using the random keys provided by $\mathcal{S}_{\mathcal{A}}^{P_i}$ while emulating $\mathcal{F}_{\mathsf{setup}}$. This is indistinguishable from $\mathcal{A}$'s view in the real-world. When P_i is a non-dealer, $\mathcal{A}$'s view consists of a subset of the random shares of α_{v} generated using the random keys provided by $\mathcal{S}_{\mathcal{A}}^{P_i}$ while emulating $\mathcal{F}_{\mathsf{setup}}$. Additionally, it also sees $\beta_{\mathsf{v}} = 0 + \alpha_{\mathsf{v}}$. Since, the missing shares of α_{v} at $\mathcal{A}$ are chosen randomly by $\mathcal{S}_{\mathcal{A}}^{P_i}$, β_{v} remains random, and hence the views are indistinguishable.

Claim 2: the view generated by $\mathcal{S}_{\mathcal{A},\mathcal{H}}^{P_j}$ is indistinguishable from $\mathcal{A}_{\mathcal{H}}$'s real-world view, where $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ knows the input and output of $\mathcal{A}_{\mathcal{H}}$, and view sent by $\mathcal{S}_{\mathcal{A}}^{P_i}$.

This is argued as follows. If P_j is the dealer, the argument follows similar to before, and $\mathcal{S}_{\mathcal{A},\mathcal{H}}^{P_j}$'s view is indistinguishable from $\mathcal{A}_{\mathcal{H}}$'s view. If P_j is a non-dealer, then $\mathcal{A}_{\mathcal{H}}$'s view consists of β_{v}, the six random shares of α_{v}, and among the four missing shares of α_{v}, it also sees three shares which are received as part of the view sent by $\mathcal{A}$ to $\mathcal{A}_{\mathcal{H}}$. Since $\mathcal{A}_{\mathcal{H}}$ still misses the share $\alpha_{\mathsf{v}_{\mathsf{ij}}}$, the β_{v} sent by $\mathcal{S}_{\mathcal{A},\mathcal{H}}^{P_j}$ remains random, and hence the views are indistinguishable. □

E.1.2 Joint sharing. The simulator for the joint sharing protocol where two parties jointly share a value v in the preprocessing phase appears in Fig. 20. The simulations for joint sharing when the value to be shared is available in the online phase is similar.

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

Malicious Simulation:

- If P_i is one among the two dealers, $\mathcal{S}_{\mathcal{A}}$ emulates $\mathcal{F}_{\mathsf{jmp}}$ with $\mathcal{A}$ as one of the senders to send one share of α_{v} to one other party.
- Else if P_i is the recipient of the share of α_{v}, then $\mathcal{S}_{\mathcal{A}}$ emulates $\mathcal{F}_{\mathsf{jmp}}$ with $\mathcal{A}$ as the receiver.
- Else, there is nothing to simulate.

Semi-Honest Simulation:

- If P_j is one of the dealers, $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ emulates $\mathcal{F}_{\mathsf{jmp}}$ with $\mathcal{A}_{\mathcal{H}}$ as one of the senders to send the share of α_{v} to one other honest party.
- Else, if P_j is the recipient of the share of α_{v}, then $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ emulates $\mathcal{F}_{\mathsf{jmp}}$ with $\mathcal{A}_{\mathcal{H}}$ as the receiver.
- Else if P_j is neither the dealer nor the receiver, there is nothing to simulate.

Figure 20: Simulator for Π_{jsh} for sharing v

Observe that view generated by $\mathcal{S}_{\mathcal{A}}^{P_i}$ is indistinguishable from $\mathcal{A}$'s real-world view. This is because values received by $\mathcal{A}$ are random which is as per the real-world protocol. Similarly, view of $\mathcal{A}_{\mathcal{H}}$ generated by $\mathcal{S}_{\mathcal{A},\mathcal{H}}^{P_j}$ is indistinguishable from real-world view.

E.1.3 Reconstruction. The ideal functionality for Π_{rec} (Fig. 13) appears in Fig. 21.

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

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

- Receive $(\mathsf{Input}, [\![\mathsf{v}]\!]_s, P_i)$ from each $P_s \in \mathcal{P}$.
- Set $\mathsf{msg}_i = \beta_{\mathsf{v}} - \sum_{1 \le i < j \le 5} \alpha_{\mathsf{v}_{\mathsf{ij}}}$ and $\mathsf{msg}_s = \perp$ for $P_s \in \mathcal{P} \setminus \{P_i\}$.

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

- $\mathcal{S}_{\mathcal{A}}$ sends it's view to $\mathcal{S}_{\mathcal{A},\mathcal{H}}$.

Figure 21: Ideal functionality for Π_{rec}

The simulator for the reconstruction protocol appears in Fig. 22.

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

Malicious Simulation:

- To simulate reconstruction towards $\mathcal{A}$:
  - Invoke $\mathcal{F}_{\mathsf{rec}}$ with $(\mathsf{Input}, [\![\mathsf{v}]\!]_i)$.
  - $\mathcal{S}_{\mathcal{A}}$ sets a missing share of $\alpha_{\mathsf{v}_{\mathsf{ij}}}$ of v, not held by P_i (and P_j) as $\alpha_{\mathsf{v}_{\mathsf{ij}}} = \beta_{\mathsf{v}} - \mathsf{v} - \sum_{1\le p<q\le 5, p\neq i, q\neq j} \alpha_{\mathsf{v}_{\mathsf{pq}}}$, where $\alpha_{\mathsf{v}_{\mathsf{pq}}}$ were sampled using the shared keys, and v is the output obtained by $\mathcal{S}_{\mathcal{A}}$ from the ideal functionality.
  - $\mathcal{S}_{\mathcal{A}}$ sends $\alpha_{\mathsf{v}_{\mathsf{ij}}}$ and its hash to $\mathcal{A}$ on behalf of the honest parties that hold $\alpha_{\mathsf{v}_{\mathsf{ij}}}$. $\mathcal{S}_{\mathcal{A}}$ sends the other shares of α_{v} which include $\alpha_{\mathsf{vik}}, \alpha_{\mathsf{vil}}, \alpha_{\mathsf{vim}}$ (and were sampled randomly), together with its hash to $\mathcal{A}$ on behalf of honest parties that hold these shares.

Semi-Honest Simulation:

- $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ receives the view from $\mathcal{S}_{\mathcal{A}}$. To simulate reconstruction towards $\mathcal{A}_{\mathcal{H}}$, $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ sends the missing shares and their hashes to $\mathcal{A}_{\mathcal{H}}$ on behalf of the honest parties by using these values as present in the view received from $\mathcal{S}_{\mathcal{A}}$.

Figure 22: Simulator for Π_{rec} of output $[\![\mathsf{v}]\!]$

Lemma E.2 (Security). *Protocol Π_{rec} (Fig. 13) realizes $\mathcal{F}_{\mathsf{rec}}$ (Fig. 21) with computational security in the $\mathcal{F}_{\mathsf{setup}}$-hybrid model against FaF adversaries $\mathcal{S}_{\mathcal{A}}, \mathcal{S}_{\mathcal{A},\mathcal{H}}$ controlling P_i, P_j respectively.*

Proof. The view generated by $\mathcal{S}_{\mathcal{A}}^{P_i}$ is indistinguishable from $\mathcal{A}$'s real-world view. This is argued as follows. $\mathcal{A}$'s view consists of random $\alpha_{\mathsf{v}_{\mathsf{pq}}}$ for $1 \le p < q \le 5, p \neq i, q \neq i$ such that one share, say, $\alpha_{\mathsf{v}_{\mathsf{ij}}}$ (unknown to $\mathcal{A}$) is adjusted as $\alpha_{\mathsf{v}_{\mathsf{ij}}} = \beta_{\mathsf{v}} - \mathsf{v} - \sum_{1\le p<q\le 5, p\neq i, q\neq j} \alpha_{\mathsf{v}_{\mathsf{pq}}}$ to ensure reconstruction of correct output. Since, these missing shares are chosen randomly by $\mathcal{S}_{\mathcal{A}}^{P_i}$, the β_{v} remains random and, the views are indistinguishable. Similarly, the view generated by $\mathcal{S}_{\mathcal{A},\mathcal{H}}^{P_j}$ is indistinguishable from $\mathcal{A}_{\mathcal{H}}$'s real-world view, since $\mathcal{A}_{\mathcal{H}}$ still misses one random share $\alpha_{\mathsf{v}_{\mathsf{ij}}}$, which keeps β_{v} random. □

E.1.4 Multiplication. The ideal functionality for Π_{mult} (Fig. 2) appears in Fig. 23.

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

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

- Receive $(\mathsf{Input}, [\![\mathsf{a}]\!]_s, [\![\mathsf{b}]\!]_s, [\alpha_{\mathsf{z}}]_s)$ from $P_s \in \mathcal{P}$. Let $P^\star$ be the malicious party controlled by $\mathcal{S}_{\mathcal{A}}$.
- Receive continue or abort with $(\mathsf{Select}, \mathsf{C})$ from $\mathcal{S}_{\mathcal{A}}$. Here, C denotes pair of parties that $\mathcal{S}_{\mathcal{A}}$ wants to choose as conflict pair.
- If received continue, compute $[\![\mathsf{z}]\!]$ where $\mathsf{z} = \mathsf{ab} + \alpha_{\mathsf{z}}$. Set $\mathsf{msg}_s = [\![\mathsf{z}]\!]_s$, for each $P_s \in \mathcal{P}$.
- Else if received abort, then:
- If $P^\star \in \mathsf{C}$, then set $\mathsf{CP} = \mathsf{C}$ and $\mathsf{msg}_s = \mathsf{CP}$ for each $P_s \in \mathcal{P}$.
- Else set CP to include $P^\star$ and one other party from $\mathcal{P}$, and $\mathsf{msg}_s = \mathsf{CP}$ for each $P_s \in \mathcal{P}$.

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

- $\mathcal{S}_{\mathcal{A}}$ sends it's view to $\mathcal{S}_{\mathcal{A},\mathcal{H}}$.

Figure 23: Ideal functionality for Π_{mult}

Due to the asymmetry in our multiplication protocol, we consider the following two cases for simulation– (i) when the maliciously corrupt P_i is one among P_1, P_2, P_3, and (ii) when the maliciously corrupt P_i is one among P_4, P_5. The simulator for case(i) appears in Fig. 24.

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

Malicious Simulation:

Preprocessing: $\mathcal{S}_{\mathcal{A}}$ emulates $\mathcal{F}_{\mathsf{MulPre}}$.

Online:

- $\mathcal{S}_{\mathcal{A}}$ honestly generates shares of β_{z} on behalf of honest parties.
- $\mathcal{S}_{\mathcal{A}}$ simulates *send* of jmp with $\mathcal{A}$ as one of the senders to send the missing share of β_{z} to the other two online parties (P_1, P_2, P_3). $\mathcal{S}_{\mathcal{A}}$ simulates *send* of jmp with $\mathcal{A}$ as the receiver to send the missing shares of β_{z} to $\mathcal{A}$ on behalf of the honest parties.

Verification:

- $\mathcal{S}_{\mathcal{A}}$ honestly generates hash on all β_{z}s involved in verification on behalf of the honest online parties, and sends the hash to $\mathcal{A}$.
- If $\mathcal{A}$ sends an inconsistency bit b = 0, $\mathcal{S}_{\mathcal{A}}$ simulates *send* and *verify* of jmp with $\mathcal{A}$ as one of the senders to send β_{z} to the offline parties (P_4, P_5), if $P_i \in \{P_1, P_2\}$. This is followed by simulation of *verify* of jmp towards $\mathcal{A}$.
- Else, if $\mathcal{A}$ sends an inconsistency bit b = 1, $\mathcal{S}_{\mathcal{A}}$ simulates the binary search where hashes are sent until $\mathcal{A}$ broadcasts an inconsistency bit with b = 0 and levels L_p, L_{p+1} are identified. $\mathcal{S}_{\mathcal{A}}$ simulates *send* and *verify* of jmp with $\mathcal{A}$ as one of the senders if $P_i \in \{P_1, P_2\}$ to send β_{z} up to level L_p. This is followed by simulation of *verify* of jmp towards $\mathcal{A}$ for β_{z}s up to level L_{p+1}. If the simulation of *verify* of latter jmp did not output a CP, $\mathcal{S}_{\mathcal{A}}$ sends the identity of P_j to $\mathcal{A}$.
- Depending on $\mathcal{A}$'s behaviour, $\mathcal{S}_{\mathcal{A}}$ sets CP and invokes $\mathcal{F}_{\mathsf{mult}}$ with $(\mathsf{Input}, [\![\mathsf{a}]\!]_i, [\![\mathsf{b}]\!]_i, [\alpha_{\mathsf{z}}]_i)$, and continue/abort and $(\mathsf{Select}, \mathsf{CP})$.

Semi-Honest Simulation:

Preprocessing: $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ emulates $\mathcal{F}_{\mathsf{MulPre}}$.

Online: If P_j is one of the online parties, then $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ simulates *send* of jmp with $\mathcal{A}_{\mathcal{H}}$ as one of the senders to send the missing share of β_{z} to the remainder honest online party. $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ simulates *send* of jmp with $\mathcal{A}_{\mathcal{H}}$ as the receiver to send the missing share of β_{z} to $\mathcal{A}_{\mathcal{H}}$ on behalf of the honest party.

Verification: If P_j is one of the online parties, then

- $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ honestly generates hash on all β_{z}s involved in verification on behalf of the honest online parties, and sends the hash to $\mathcal{A}_{\mathcal{H}}$.
- Depending on the bit obtained in the view from $\mathcal{S}_{\mathcal{A}}$, $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ either proceeds with simulating jmp with $\mathcal{A}_{\mathcal{H}}$ as one of the senders if $P_j \in \{P_1, P_2\}$ for sending β_{z} towards offline parties, or it simulates the hash based consistency check. For the latter, $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ recursively performs the hash exchange until levels L_p, L_{p+1} as present in the view of $\mathcal{S}_{\mathcal{A}}$ are identified. Following this, $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ simulates *send* and *verify* of jmp with $\mathcal{A}_{\mathcal{H}}$ as one of the senders if P_j is one among P_1 or P_2 for sending β_{z} up to level L_p to offline parties. Then, simulation of *verify* of jmp towards $\mathcal{A}_{\mathcal{H}}$ for β_{z}s up to level L_{p+1} is performed.

If P_j is one of the offline parties, then $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ simulates the similar steps as above which are carried out after the hash based consistency check.

Figure 24: Simulator for Π_{mult} when $P_i \in \{P_1, P_2, P_3\}$

The simulator for case(ii) appears in Fig. 25.

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

Malicious Simulation:

Preprocessing: $\mathcal{S}_{\mathcal{A}}$ emulates $\mathcal{F}_{\mathsf{MulPre}}$.
Online: There is nothing to simulate.
Verification:

- $\mathcal{S}_{\mathcal{A}}$ honestly generates β_z on behalf of honest parties.
- $\mathcal{S}_{\mathcal{A}}$ emulates $\mathcal{F}_{\mathsf{jmp}}$ with $\mathcal{A}$ as the receiver to send β_z to $\mathcal{A}$ on behalf of the honest parties.
- Depending on $\mathcal{A}$'s behaviour, $\mathcal{S}_{\mathcal{A}}$ sets CP and invokes $\mathcal{F}_{\mathsf{mult}}$ with $(\mathsf{Input}, [\![a]\!]_i, [\![b]\!]_i, [\alpha_z]_i)$, and continue/abort and $(\mathsf{Select}, \mathsf{CP})$.

Semi-Honest Simulation:

Preprocessing: $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ emulates $\mathcal{F}_{\mathsf{MulPre}}$.
Online:

- If P_j is one of the online parties:
  - $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ emulates $\mathcal{F}_{\mathsf{jmp}}$ with $\mathcal{A}_{\mathcal{H}}$ as one of the senders to send the missing share of β_z (generated honestly) to the other two online parties. $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ emulates $\mathcal{F}_{\mathsf{jmp}}$ with $\mathcal{A}_{\mathcal{H}}$ as the receiver to send the missing shares of β_z to $\mathcal{A}_{\mathcal{H}}$ on behalf of the honest parties.
- If P_j is one of the offline parties, there is nothing to simulate.

Verification:

- If P_j is one of the online parties, $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ sends the hash of all β_z in this segment to $\mathcal{A}_{\mathcal{H}}$ and emulates $\mathcal{F}_{\mathsf{jmp}}$ with $\mathcal{A}_{\mathcal{H}}$ as one of the senders to send β_z to the honest offline party.
- If P_j is one of the offline parties, then $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ emulates $\mathcal{F}_{\mathsf{jmp}}$ with $\mathcal{A}_{\mathcal{H}}$ as receiver to send β_z (reused from the view received from $\mathcal{S}_{\mathcal{A}}$) to $\mathcal{A}_{\mathcal{H}}$ on behalf of honest parties.

Figure 25: Simulator for Π_{mult} when $P_i \in \{P_4, P_5\}$

Lemma E.3 (Security). *Protocol Π_{mult} (Fig. 2) realizes $\mathcal{F}_{\mathsf{mult}}$ (Fig. 23) with computational security in the $(\mathcal{F}_{\mathsf{setup}}, \mathcal{F}_{\mathsf{jmp}})$-hybrid model against FaF adversaries $\mathcal{S}_{\mathcal{A}}, \mathcal{S}_{\mathcal{A},\mathcal{H}}$ controlling P_i, P_j respectively.*

Proof. We argue indistinguishability in the following two cases.

Case 1: When the maliciously corrupt P_i is one among P_1, P_2, P_3. Observe that the view generated in this case by $\mathcal{S}_{\mathcal{A}}^{P_i}$ is indistinguishable from $\mathcal{A}$'s real-world view. This is because $\mathcal{A}$ receives random shares of β_z which are generated honestly by the simulator. Since $\mathcal{A}$ still misses one share of the mask α_z, the β_z received via $\mathcal{F}_{\mathsf{jmp}}$ remains random. Hence, the views are indistinguishable. A similar argument applies to $\mathcal{A}_{\mathcal{H}}$'s view being indistinguishable.

Case 2: When the maliciously corrupt P_i is one among P_4, P_5. Similar to case 1, the real-world view of $\mathcal{A}$ is indistinguishable from the view generated by $\mathcal{S}_{\mathcal{A}}$ since $\mathcal{A}$ misses one share of the α_z which keeps β_z random. A similar argument, as before, holds for indistinguishability of the view of $\mathcal{A}_{\mathcal{H}}$. □

E.1.5 The complete 5PC. The ideal functionality for computing a function f via $(1,1)$-FaF secure 5PC appears in Fig. 26.

Overview of the simulation steps. Observe that the complete 5PC protocol begins with the input sharing phase, followed by an evaluation phase where addition and multiplication gates are evaluated and concludes with a reconstruction phase. For each of these phases, we use the simulation steps described above depending on the identity of the maliciously corrupt P_i and a semi-honest P_j. The simulation proceeds as follows. The simulator is able to extract malicious $\mathcal{A}$'s input while performing the simulation steps for input sharing, and knows $\mathcal{A}_{\mathcal{H}}$'s input. Thus, it can invoke the ideal functionality, $\mathcal{F}_{\mathsf{5PC-FaF}}$ (Fig. 26) to obtain the output of the function being simulated. Simulation is not required for addition gates as it is a local operation. For multiplication gates, the simulation steps as described for multiplication are invoked. Observe that in all steps, the view of $\mathcal{A}$, as generated by $\mathcal{S}_{\mathcal{A}}^{P_i}$, is indistinguishable from its real-world view. Similar is the case for $\mathcal{A}_{\mathcal{H}}$. If at any step, $\mathcal{F}_{\mathsf{jmp}}$ outputs a CP, 5PC simulation stops and the rest of the steps are simulated using the semi-honest 3PC simulator. Steps for share conversion have to be simulated towards $\mathcal{A}_{\mathcal{H}}$, where the simulator carries out steps as per the honest protocol execution, reusing the shares held by $\mathcal{A}$, wherever necessary. Finally, for reconstructing the output, the simulator uses the output received from $\mathcal{F}_{\mathsf{5PC-FaF}}$ to adjust the value of the missing share that has to be sent to $\mathcal{A}$ and $\mathcal{A}_{\mathcal{H}}$. Indistinguishability of the views follows from the indistinguishability of the views for each of the phases. Thus, the view generated by $\mathcal{S}_{\mathcal{A}}^{P_i}$ is indistinguishable from $\mathcal{A}$'s real-world view, and the view generated by $\mathcal{S}_{\mathcal{A},\mathcal{H}}^{P_j}$ is indistinguishable from $\mathcal{A}_{\mathcal{H}}$'s real-world view.

Functionality $\mathcal{F}_{\mathsf{5PC-FaF}}$

$\mathcal{F}_{\mathsf{5PC-FaF}}$ 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. $(y_1, y_2, y_3, y_4, y_5) = f(x_1, x_2, x_3, x_4, x_5)$.

- $\mathcal{F}_{\mathsf{5PC-FaF}}$ receives (Input, x_s) from $P_s \in \mathcal{P}$ and computes $(y_1, y_2, y_3, y_4, y_5) = f(x_1, x_2, x_3, x_4, x_5)$.

Output: Send (Output, y_s) to $P_s \in \mathcal{P}$.
$\mathcal{S}_{\mathcal{A}}$ sends its view to $\mathcal{S}_{\mathcal{A},\mathcal{H}}$.

Figure 26: Ideal functionality for evaluating f in 5PC $(1,1)$-FaF Model

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

Malicious Simulation:

- $\mathcal{S}_{\mathcal{A}}$ emulates $\mathcal{F}_{\mathsf{setup}}$ to generate common PRF keys.
- $\mathcal{S}_{\mathcal{A}}$ invokes the simulator for input sharing and extracts $\mathcal{A}$'s input. $\mathcal{S}_{\mathcal{A}}$ invokes $\mathcal{F}_{\mathsf{5PC-FaF}}$ on $\mathcal{A}$'s input to obtain the function output v.
- For addition operations, there is nothing to simulate. For multiplications, $\mathcal{S}_{\mathcal{A}}$ invokes the simulator for multiplication.
- $\mathcal{S}_{\mathcal{A}}$ invokes the reconstruction simulator to reconstruct output v.
- $\mathcal{S}_{\mathcal{A}}$ sends its view to $\mathcal{S}_{\mathcal{A},\mathcal{H}}$.

Semi-Honest Simulation:

- $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ invokes the simulator for input sharing.
- For addition operations, there is nothing to simulate. For multiplications, $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ invokes the simulator for multiplication.
- $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ invokes the reconstruction simulator to reconstruct output v.

Figure 27: Simulator $\mathcal{S}_{\mathcal{A}}^{P_i}$ for 5PC − FaF

Theorem E.4. *Assuming collision resistant hash functions exist, protocol* 5PC − FaF *(Fig. 4) realizes $\mathcal{F}_{\mathsf{5PC-FaF}}$ (Fig. 26) with computational security in the $\mathcal{F}_{\mathsf{setup}}$-hybrid model with $(1,1)$-FaF security.*

Proof. The view of the adversaries generated by the simulators is indistinguishable from their real-world views. The indistinguishability of the views from input sharing and multiplication follows

from Lemma E.1 and Lemma E.3, respectively. With respect to reconstruction, on obtaining the output from $\mathcal{F}_{\text{5PC-FaF}}$, the simulators either simulate the reconstruction steps (see Lemma E.2 for indistinguishability argument), or execute the simulator for semi-honest 3PC. In both cases, the simulated view is indistinguishable from the real-world view. □

E.2 Simulations for Building Blocks

In this section, we describe the simulation steps for the building blocks described in §4. We begin with the simulation steps for multi-input multiplication, dot product, bit to arithmetic, bit injection, bit extraction and arithmetic to Boolean.

Since the multi-input multiplication and dot product protocol are very similar to the multiplication protocol, we omit simulation steps for the same. Further, observe that the protocol for bit to arithmetic essentially invokes the joint sharing and multiplication protocols. Hence, simulation steps for bit to arithmetic involves executing the simulation steps for joint sharing and multiplication in the order in which they appear in the protocol. Indistinguishability follows from the indistinguishability of the simulation steps in the underlying protocols. Similar to bit to arithmetic, bit injection involves an invocation of bit to arithmetic followed by a multiplication. Hence, the simulation steps follow from the simulation of the underlying protocols. Finally, bit extraction, truncation as well as arithmetic to Boolean rely on invocation of joint sharing following by evaluating the bit extraction or the PPA circuit. Both the circuit evaluations rely on invoking the multiplication protocol. Hence, similar to the previous protocols, simulation steps for bit extraction, truncation and arithmetic to Boolean can be obtained by following the steps for simulating joint sharing and multiplication, in the order in which they appear in the resultant protocol.

Similarly, it is easy to observe that the protocols for oblivious select, equality check, comparison, maxpool and ReLU build on top of the prior building blocks. Hence, their simulation follows from simulation of the underlying protocols.

E.3 Security against a $(1,1)$-mixed adversary

A closely related notion to FaF is that of mixed adversarial model [6, 20, 26, 28, 30, 33, 34], where a single (centralized) adversary is allowed to corrupt t parties maliciously and a disjoint subset of $h^\star$ parties semi-honestly. A protocol secure against such an adversary is said to be $(t, h^\star)$-mixed secure. It may seem that the mixed notion subsumes the FaF notion, but [1] shows otherwise. However, we show that our designed protocols are also secure in the $(1,1)$ mixed adversarial model. The intuition for our protocols being secure in the mixed adversarial model as well is as follows. Observe that since the mixed model comprises a centralized adversary, as opposed to the decentralized one in the FaF model, the view of the semi-honest parties is available to the adversary while deciding the attack strategy for the malicious parties. The design of our protocols is such that it inherently is capable of withstanding such attacks due to the threshold of our secret-sharing scheme being set as $t + h^\star$, thus lending our protocols secure against the centralized $(1,1)$-mixed adversary as well. We next provide the simulation proof for the same. Since the proofs follow easily from the simulation proofs for FaF security, in our case, we restrict to discussing the mixed-secure simulation for the sharing protocol.

The ideal functionality for the the sharing protocol secure against a mixed adversary appears in Fig. 28.

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

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

– Receive $(\text{Input}, \mathsf{v})$ from dealer $P_d \in \mathcal{P}$. Let $P^\star$ be the malicious party corrupted by $\mathcal{S}_{\text{mixed}}$.

– Receive continue or abort with $(\text{Select}, \mathsf{C})$ from $\mathcal{S}_{\text{mixed}}$. Here, C denotes pair of parties that $\mathcal{S}_{\text{mixed}}$ wants to choose as conflict pair.

– If received continue, randomly pick $\alpha_{\mathsf{v}_{ij}} \in \mathbb{Z}_{2^\ell}$, for $1 \le i < j \le 5$ and compute $\beta_{\mathsf{v}} = \mathsf{v} + \sum_{1 \le i < j \le 5} \alpha_{\mathsf{v}_{ij}}$. Set $\text{msg}_s = (\beta_{\mathsf{v}}, \{\alpha_{\mathsf{v}_{ij}}\}_{i \ne s, j \ne s})$, for each $P_s \in \mathcal{P}$.

– Else if received abort, then:

– If $P^\star \in \mathsf{C}$, then set $\mathsf{CP} = \mathsf{C}$ and $\text{msg}_s = \mathsf{CP}$ for each $P_s \in \mathcal{P}$.

– Else set CP to include $P^\star$ and one other party from $\mathcal{P}$, and $\text{msg}_s = \mathsf{CP}$ for each $P_s \in \mathcal{P}$.

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

Figure 28: Mixed-secure ideal functionality for input sharing

The simulator for the sharing protocol secure against a mixed adversary appears in Fig. 29.

Simulator $\mathcal{S}_{\text{mixed}}$

Let P_l be the malicious party and P_m be the semi-honest party controlled by adversary $\mathcal{A}$.

Preprocessing

– $\mathcal{S}_{\text{mixed}}$ emulates $\mathcal{F}_{\text{setup}}$ and gives the respective keys to $\mathcal{A}$. The shares of α_{v} that are held by $\mathcal{A}$ are sampled non-interactively using the shared keys. Other values ($\alpha_{\mathsf{v}_{ij}}$ for $1 \le i < j \le 5$ and $\alpha_{\mathsf{v}_{ji}}$ for $1 \le j < i \le 5$), not known to $\mathcal{A}$, are sampled randomly.

Online

– If P_l or P_m is the dealer, $\mathcal{S}_{\text{mixed}}$ receives β_{v} from $\mathcal{A}$. Given the knowledge of all shares of α_{v}, $\mathcal{S}_{\text{mixed}}$ obtains $\mathcal{A}$'s input as $\mathsf{v} = \beta_{\mathsf{v}} - \alpha_{\mathsf{v}}$. Following this, $\mathcal{S}_{\text{mixed}}$ emulates $\mathcal{F}_{\text{jmp}}$ with $\mathcal{A}$ as one of the senders, to deliver β_{v} to all parties. Depending on $\mathcal{A}$'s behaviour, $\mathcal{S}_{\text{mixed}}$ sets CP and invokes $\mathcal{F}_{\text{Sh}}^{\text{mixed}}$ with $(\text{Input}, \mathsf{v})$, and continue/abort and $(\text{Select}, \mathsf{CP})$.

– Else, $\mathcal{S}_{\text{mixed}}$ honestly generates β_{v} by setting the input, v, of honest dealer as $\mathsf{v} = 0$. $\mathcal{S}_{\text{mixed}}$ either sends β_{v} to $\mathcal{A}$ and/or emulates $\mathcal{F}_{\text{jmp}}$ to deliver β_{v} to all, with $\mathcal{A}$ either as the sender or receiver, depending on the identity of P_i. Depending on $\mathcal{A}$'s behaviour, $\mathcal{S}_{\text{mixed}}$ sets CP and invokes $\mathcal{F}_{\text{Sh}}^{\text{mixed}}$ with continue or abort, and $(\text{Select}, \mathsf{CP})$.

Figure 29: Simulator corresponding to $\mathcal{F}_{\text{Sh}}^{\text{mixed}}$

Observe that the view generated by the simulator is indistinguishable from the real-world view, and the argument follows similar to as given in Lemma E.1.

F PREPROCESSING PHASE OF MULTIPLICATION

Here we discuss the protocol carried out in the preprocessing phase to perform multiplication. The protocol is similar to the one proposed in [12], where first a semi-honest protocol is executed, followed by verifying the correctness of the semi-honest execution.

The difference lies in the steps performed when the verification fails and a pair of conflicting parties is output. In such a case, owing to the presence of at most one malicious party in our setting, we eliminate the pair of parties in conflict, and the computation proceeds via *semi-honest* 3PC unlike the malicious 3PC used in the original protocol. Further, we do not require use of tags (or message authentication codes) to ensure a consistent share conversion, due to the presence of only a single malicious party. The verification protocol has a communication cost which is sublinear in the number of multiplication triples to be verified, and thus, its cost can be amortized away for multiple multiplications. Thus, the cost of the preprocessing phase boils down to the cost of the semi-honest 5PC protocol which is 6 ring elements. While the protocol of [12] is proven to be secure according to the standard security definition, we prove that the variant described above is $(1,1)$-FaF secure in the 5PC setting. We provide the details of the protocol (mostly follows from [12]) as well for ease of understanding of the proof.

The verification of the semi-honest execution can be reduced to the problem of verifying the correctness of multiplications (several degree-2 equations). We begin with discussing the protocol for verifying the correctness of a degree-2 equations(realized by the ideal functionality $\mathcal{F}_{\mathsf{CheatIdentify}}$). This protocol serves as the basis for the verification protocol (realized by the ideal functionality $\mathcal{F}_{\mathsf{Verify}}$) which is discussed subsequently. The verification protocol relies on 5 invocations (one for each party in $\mathcal{P}$) of $\mathcal{F}_{\mathsf{CheatIdentify}}$ to verify the correctness of the multiplication triples. Due to the top-down approach of explaining the functionalities, the use of $\mathcal{F}_{\mathsf{CheatIdentify}}$ may not be evident until the details of $\mathcal{F}_{\mathsf{Verify}}$ are described. Hence, we request a reader to read §F.0.1 as an independent section. Finally, we discuss the main protocol Π_{mulPre}, which involves executing a semi-honest 5PC protocol followed by an invocation of $\mathcal{F}_{\mathsf{Verify}}$. On the way, we also prove that these protocols are $(1,1)$-FaF secure in the 5PC setting as well as discuss their communication complexities.

F.0.1 Checking correctness of degree-2 relations. We first discuss a protocol that allows parties to prove the correctness of a degree-2 computation carried out on their shares. The protocol follows along the lines of the protocol in [12] and we demonstrate that it is secure in the $(1,1)$-FaF model for 5PC. We begin with the protocol for fields and discuss how it can be extended to work over rings as shown in [12].

Specifically, party P_i wants to prove the correctness of the following equation:

$$c - \sum_{k=1}^{L} \left(\vec{\mathsf{a}}_k \diamond \vec{\mathsf{b}}_k\right) = 0 \tag{3}$$

where $c, \{\vec{\mathsf{a}}_k\}_{k=1}^{L}, \{\vec{\mathsf{b}}_k\}_{k=1}^{L}$ are known to P_i and $[\cdot]$-shared among parties in $\mathcal{P}$. Further, we assume that P_i knows all $[\cdot]$-shares of c. Looking ahead, $\{\vec{\mathsf{a}}_k\}_{k=1}^{L}, \{\vec{\mathsf{b}}_k\}_{k=1}^{L}$ represent P_i's $[\cdot]$-shares of $\{a_k\}_{k=1}^{L}, \{b_k\}_{k=1}^{L}$, while c represents P_i's additive share ($\langle\cdot\rangle$-share) of $\sum_{k=1}^{L} \mathsf{a}_k \cdot \mathsf{b}_k$ obtained by operating on its shares $\{\vec{\mathsf{a}}_k\}_{k=1}^{L}, \{\vec{\mathsf{b}}_k\}_{k=1}^{L}$, which is denoted by the operation $\diamond$[7]. Note here that we abuse the vector notation to mean $[\cdot]$-sharing. By virtue of $[\cdot]$-sharing, given $[\cdot]$-sharing of $\{\mathsf{a}_k, \mathsf{b}_k\}_{k=1}^{L}$ (which will be the case in the final protocol), parties can locally generate $[\cdot]$-sharing of the $[\cdot]$-shares $(\{\vec{\mathsf{a}}_k\}_{k=1}^{L}, \{\vec{\mathsf{b}}_k\}_{k=1}^{L})$ held by P_i. This holds because for every share held by P_i, 2 other parties also possess it. Hence, it is possible to define a sharing where the share of one subset of 3 parties is P_i's share itself, while the other shares are 0. For instance, if v is $[\cdot]$-shared and $\vec{\mathsf{v}} = (\mathsf{v}_{23}, \mathsf{v}_{24}, \mathsf{v}_{25}, \mathsf{v}_{34}, \mathsf{v}_{35}, \mathsf{v}_{45})$ denote the tuple of shares held by P_1 (where the subscript denotes the pair of parties which does not possess this share), then $[\vec{\mathsf{v}}] = ([\mathsf{v}_{23}], [\mathsf{v}_{24}], [\mathsf{v}_{25}], [\mathsf{v}_{34}], [\mathsf{v}_{35}], [\mathsf{v}_{45}])$, where $[\mathsf{v}_{jk}]$ is generated by setting all but one of its shares as 0, and the non-zero share being v_{jk} (which is held by all 3 parties in $\mathcal{P} \setminus \{P_j, P_k\}$).

Relying on the distributed zero-knowledge proof system from [10] allows to prove the correctness of Eq. (3) with sublinear communication complexity. Note that in the scenario that the proof is rejected due to one of the parties' misbehaviour, the prover will be able to identify the cheating party. In this case, the prover together with this party are regarded as a pair of conflicting parties, one of which is guaranteed to be corrupt. This is captured by the ideal functionality $\mathcal{F}_{\mathsf{CheatIdentify}}$ which checks for correctness of Eq. (3) and either outputs an accept, or a pair of parties that are in conflict with each other (one among which is guaranteed to be corrupt). The functionality is defined in Fig. 30.

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

Let $\mathcal{S}_{\mathcal{A}}$ be an ideal world malicious adversary and $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ be the ideal world semi-honest adversary. Let honest parties hold consistent $[\cdot]$-sharings $[c], \{[\vec{\mathsf{a}}_k]\}_{k=1}^{L}, \left\{\left[\vec{\mathsf{b}}_k\right]\right\}_{k=1}^{L}$. The functionality is invoked by an index i sent by honest parties and works as follows.

(1) $\mathcal{F}_{\mathsf{CheatIdentify}}$ receives from honest parties their shares of $c, \{\vec{\mathsf{a}}_k\}_{k=1}^{L}, \left\{\vec{\mathsf{b}}_k\right\}_{k=1}^{L}$.

(2) $\mathcal{F}_{\mathsf{CheatIdentify}}$ computes $c, \{\vec{\mathsf{a}}_k\}_{k=1}^{L}, \left\{\vec{\mathsf{b}}_k\right\}_{k=1}^{L}$. It computes the corrupted party's shares of these values and sends them to $\mathcal{S}_{\mathcal{A}}$. If P_i is corrupted, then it also sends $[\cdot]$-shares of c, and $\{\vec{\mathsf{a}}_k\}_{k=1}^{L}, \left\{\vec{\mathsf{b}}_k\right\}_{k=1}^{L}$ to $\mathcal{S}_{\mathcal{A}}$. $\mathcal{F}_{\mathsf{CheatIdentify}}$ sends $P_{\mathcal{H}}$'s shares of $c, \{\vec{\mathsf{a}}_k\}_{k=1}^{L}, \left\{\vec{\mathsf{b}}_k\right\}_{k=1}^{L}$ to $\mathcal{S}_{\mathcal{A},\mathcal{H}}$, where $P_{\mathcal{H}}$ is controlled by $\mathcal{S}_{\mathcal{A},\mathcal{H}}$.

(3) $\mathcal{F}_{\mathsf{CheatIdentify}}$ checks that Eq 3 holds.
- If it holds then $\mathcal{F}_{\mathsf{CheatIdentify}}$ sends accept to $\mathcal{S}_{\mathcal{A}}$, and receives out $\in$ {accept, reject} from it. $\mathcal{F}_{\mathsf{CheatIdentify}}$ forwards out to honest parties.
- If it does not hold then $\mathcal{F}_{\mathsf{CheatIdentify}}$ sends reject to honest parties.

(4) If honest parties received reject:
- If P_i is corrupt, $\mathcal{S}_{\mathcal{A}}$ sends an index $j \in \{1,2,\ldots,5\}$ to $\mathcal{F}_{\mathsf{CheatIdentify}}$
- If P_i is honest, $\mathcal{S}_{\mathcal{A}}$ sends an index $j \in \{1,2,\ldots,5\}$ to $\mathcal{F}_{\mathsf{CheatIdentify}}$, where P_j is corrupt.
- $\mathcal{F}_{\mathsf{CheatIdentify}}$ sends the pair (i,j) to honest parties.

(5) $\mathcal{S}_{\mathcal{A}}$ sends its view to $\mathcal{S}_{\mathcal{A},\mathcal{H}}$.

Figure 30: Ideal functionality for proving correctness of degree-2 equation by prover P_i

[7] $[a]$ consists of 10 shares $\{a_{1,2}, a_{1,3}, \ldots, a_{4,5}\}$. Similar is the case with $[b]$. The product $c = a \cdot b$ can thus be written as the sum of products of the form $a_{i,j} b_{k,l}$ $\forall 1 \le i \le j \le 5$ and $1 \le k \le l \le 5$. Thus, the additive shares of c can be obtained by splitting each term $a_{i,j} b_{k,l}$ contributed by some party who has both the shares. This operation of obtaining additive shares of c using local shares of a, b is captured by the $\diamond$ operator.

The concrete protocol for $\mathcal{F}_{\text{CheatIdentify}}$ We begin with a high level idea of the protocol. Given a g-gate which is defined as follows:

$$g(\vec{\mathbf{v}_1}, \ldots, \vec{\mathbf{v}_L}) = \sum_{l=1}^{L/2} \mathbf{v}_{2\vec{l}-1} \diamond \vec{\mathbf{v}_{2l}}$$

where $\diamond$ denotes the operation of obtaining the additive shares of $\mathsf{v}_i \cdot \mathsf{v}_j$ given their $[\cdot]$-sharing, i.e. $\vec{\mathbf{v}_i}, \vec{\mathbf{v}_j}$.

Protocol $\Pi_{\text{CheatIdentify}}\left(\mathcal{P}, P_i, [c], \{[\vec{\mathbf{a}}]_k\}_{k=1}^{L}, \left\{\left[\vec{\mathbf{b}}\right]_k\right\}_{k=1}^{L}\right)$

(1) Parties set $\bar{L} = L$.
(2) For $l = 1$ to $\log \bar{L} - 1$:
   - Parties define linear polynomials $\vec{\mathbf{f}_1}, \vec{\mathbf{f}_2}, \ldots, \vec{\mathbf{f}_L}$ such that for each $e \in \{1, 2, \ldots, L\}$, polynomial $\vec{\mathbf{f}_e}$ is defined by the following two points:

$$\vec{\mathbf{f}_e}(1) = \begin{cases} \vec{\mathbf{a}}_{\lceil e/2 \rceil} & \text{if } e \bmod 2 = 1 \\ \vec{\mathbf{b}}_{e/2} & \text{if } e \bmod 2 = 0 \end{cases} \quad \vec{\mathbf{f}_e}(2) = \begin{cases} \vec{\mathbf{a}}_{L/2+\lceil e/2 \rceil} & \text{if } e \bmod 2 = 1 \\ \vec{\mathbf{b}}_{L/2+e/2} & \text{if } e \bmod 2 = 0 \end{cases}$$

   - Let $q(x) = g(\vec{\mathbf{f}_1}(x), \vec{\mathbf{f}_2}(x), \ldots, \vec{\mathbf{f}_L}(x))$ be a degree-2 polynomial where

$$g(\vec{\mathbf{f}_1}(x), \vec{\mathbf{f}_2}(x), \ldots, \vec{\mathbf{f}_L}(x)) = \sum_{j=1}^{L/2} \vec{\mathbf{f}}_{2j-1}(x) \diamond \vec{\mathbf{f}}_{2j}(x)$$

     P_i computes $q(1), q(2), q(3)$ and shares them among parties in $\mathcal{P}$ (via the field equivalent of $\Pi_{\text{RSS-Sh}}$ protocol in Fig. 11).
   - Parties locally compute $[b_l] = [c] - [q(1)] - [q(2)]$ and store the result.
   - Parties generate a random $\mathsf{r} \in \mathbb{F}$ non-interactively using their shared key setup.
   - Parties locally compute $[q(\mathsf{r})]$ and $\left[\vec{\mathbf{f}_1}(\mathsf{r})\right], \left[\vec{\mathbf{f}_2}(\mathsf{r})\right], \ldots, \left[\vec{\mathbf{f}_L}(\mathsf{r})\right]$ via Lagrange interpolation.
   - Parties set $c \leftarrow q(\mathsf{r})$, and $\forall k \in \{1, 2, \ldots, L/2\} : \vec{\mathbf{a}}_k \leftarrow \vec{\mathbf{f}}_{2k-1}(\mathsf{r})$, $\vec{\mathbf{b}}_k \leftarrow \vec{\mathbf{f}}_{2k}(\mathsf{r})$ and $L \leftarrow L/2$.
(3) Parties exit the loop with $L = 2$ and inputs $c, \vec{\mathbf{a}}_1, \vec{\mathbf{a}}_2, \vec{\mathbf{b}}_1, \vec{\mathbf{b}}_2$ that are known to P_i and secret shared among other parties. Next,
   - Parties non-interactively generate $[\vec{\mathbf{w}}_1], [\vec{\mathbf{w}}_2]$ where $\vec{\mathbf{w}}_1, \vec{\mathbf{w}}_2 \in \mathbb{F}^{\mathsf{g}}$ are known to P_i. Here, g denotes the number of shares as part of $[\cdot]$-sharing held by each party. Parties define polynomials $\vec{\mathbf{f}_1}, \vec{\mathbf{f}_2}$ of degree 2 such that $\vec{\mathbf{f}_1}(0) = \vec{\mathbf{w}}_1, \vec{\mathbf{f}_1}(1) = \vec{\mathbf{a}}_1, \vec{\mathbf{f}_1}(2) = \vec{\mathbf{a}}_2$ and $\vec{\mathbf{f}_2}(0) = \vec{\mathbf{w}}_2, \vec{\mathbf{f}_2}(1) = \vec{\mathbf{b}}_1, \vec{\mathbf{f}_2}(2) = \vec{\mathbf{b}}_2$.
   - P_i defines the degree-4 polynomial $q(x) = g(\vec{\mathbf{f}_1}(x), \vec{\mathbf{f}_2}(x))$ where $g(\vec{\mathbf{f}_1}(x), \vec{\mathbf{f}_2}(x)) = \vec{\mathbf{f}_1}(x) \diamond \vec{\mathbf{f}_2}(x)$, and computes $q(0), q(1), \ldots, q(4)$
   - P_i shares $q(0), q(1), \ldots, q(4)$ among parties in $\mathcal{P}$ (via the field equivalent of $\Pi_{\text{RSS-Sh}}$ protocol in Fig. 11).
   - Parties locally compute $\left[b_{\log L}\right] = [c] - [q(1)] - [q(2)]$.
   - Parties non-interactively generate $\mathsf{r}, \gamma_1, \ldots, \gamma_{\log L} \in \mathbb{F}$, and compute $[b] = \sum_{l=1}^{\log L} \gamma_l \cdot [b_l]$.
   - Parties locally compute $\left[\vec{\mathbf{f}_1}(\mathsf{r})\right], \left[\vec{\mathbf{f}_2}(\mathsf{r})\right]$ and $[q(\mathsf{r})]$ via Lagrange interpolation.
   - Parties reconstruct $b, q(\mathsf{r}), \vec{\mathbf{f}_1}(\mathsf{r}), \vec{\mathbf{f}_2}(\mathsf{r})$ towards each party where each missing share is broadcast. If reconstruction has an inconsistency, or if $q(\mathsf{r}) \neq g(\vec{\mathbf{f}_1}(\mathsf{r}), \vec{\mathbf{f}_2}(\mathsf{r}))$ of if $b \neq 0$, then parties output reject. Else, parties output accept.
   - If parties output reject, P_i identifies a party P_j who sent incorrect messages in the previous step, and broadcasts j to all the parties. Parties output the conflict pair (i, j).

Figure 31: Realizing $\mathcal{F}_{\text{CheatIdentify}}$

Eq. (3) can be written as

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

The prover, knowing all the inputs, can compute the output of the two g-gates and $[\cdot]$-share them among the parties in $\mathcal{P}$. Let $g_1 = g(\vec{\mathbf{a_1}}, \vec{\mathbf{b_1}}, \ldots, \vec{\mathbf{a_{L/2}}}, \vec{\mathbf{b_{L/2}}})$ and $g_2 = g(\vec{\mathbf{a_{L/2+1}}}, \vec{\mathbf{b_{L/2+1}}}, \ldots, \vec{\mathbf{a_L}}, \vec{\mathbf{b_L}})$. Thus, parties can compute $[b] = [c] - [g_1] - [g_2]$ and check if $b = 0$ by reconstructing b. To ensure that a corrupt P_i did not cheat while generating $[\cdot]$-shares of g_1, g_2, parties perform an additional test. For this, parties define polynomials $\vec{\mathbf{f_1}}, \ldots, \vec{\mathbf{f_L}}$ as follows: for each $e \in \{1, 2, \ldots, L\}$, $\vec{\mathbf{f}_e}(1)$ is the eth input vector to the 1st g-gate and $\vec{\mathbf{f}_e}(2)$ is the eth input vector to the 2nd g-gate. $\vec{\mathbf{f}_e}$ is thus a linear function. Next, define polynomial $q(x) = g(\vec{\mathbf{f}_1}(x), \ldots, \vec{\mathbf{f}_L}(x))$. Thus, $q(1), q(2)$ are the outputs of the first and second g-gate, respectively, and q is of degree 2 (since the multiplicative depth of g-gate is 1 and degree of $\vec{\mathbf{f}_e}$ is 1). To ensure that P_i shared the correct $g(1), g(2)$, it suffices for the parties to check if $q(\mathsf{r}) = g(\vec{\mathbf{f}_1}(\mathsf{r}), \ldots, \vec{\mathbf{f}_L}(\mathsf{r}))$ for a random r in the ring/field. For this, parties compute $[\cdot]$-shares of $q(\mathsf{r}), \vec{\mathbf{f}_1}(\mathsf{r}), \ldots, \vec{\mathbf{f}_L}(\mathsf{r})$ via Lagrange interpolation on their local shares and check for the equality on clear. This also requires P_i to share $q(3)$ so that parties have sufficient points on q. To reduce the cost from L shares which is linear in L, to logarithmic in L, P_i is made to prove that

$$q(\mathsf{r}) - g(\vec{\mathbf{f}_1}(\mathsf{r}), \ldots, \vec{\mathbf{f}_L}(\mathsf{r})) = 0 \tag{4}$$

by repeating the same process (since Eq. (4) has the same form as that of Eq. (3)). Parties repeat the process $\log L$ times until a constant number of inputs are left, which are verified on clear. Since $\vec{\mathbf{f}_e}(\mathsf{r})$ is a linear combination of the inputs, to avoid leaking any information about the inputs, in the final step, the $\vec{\mathbf{f}}$ polynomials are randomized by adding one additional random point one each polynomial. This increases the degree of $\vec{\mathbf{f}}$ to 2 and that of q to 4, and requires P_i to generate and share additional points on q. In case parties reject the proof, the prover is asked to identify the cheating party. The pair of parties including the prover and the party identified by the prover, are then regarded as the corrupted pair of parties. For this, observe that every message sent by a party other than the prover is a function of (i) the messages received from the prover, (ii) the inputs to the protocol, and (iii) the randomness used. Since the prover knows all of these, it can compute the message that should have been sent by the other parties and identify inconsistencies, if any. The protocol appears in Fig. 31.

Cheating probability over finite fields. There are two cases which lead to the parties outputting accept even when Eq. (3) does not hold– (i) the linear combination of the b values yields a 0, and (ii) when P_i cheats during sharing points on q and thus $q \neq g(\vec{\mathbf{f}_1}, \ldots, \vec{\mathbf{f}_L})$ and $h(x) = q(x) - g(\vec{\mathbf{f}_1}(x), \ldots, \vec{\mathbf{f}_L}(x))$ is a non-zero polynomial. While (i) happens with probability $\frac{1}{\mathbb{F}}$, for (ii), the probability that $h(\mathsf{r}) = 0$ for a random $\mathsf{r} \in \mathbb{F} \setminus \{1, 2, 3\}$ is bounded by $\frac{2}{\mathbb{F}-2}$ (since degree of polynomial h is 2) in the first $\log L - 1$ rounds and $\frac{4}{\mathbb{F}-5}$ in the last round (since degree of h is now 4). Thus, the overall cheating probability is bounded by

$$\frac{2(\log L - 1)}{\mathbb{F} - 3} + \frac{4}{\mathbb{F} - 5} < \frac{2 \log L + 4}{\mathbb{F} - 5}$$

Extension to rings. While the protocol described works over fields, using the extension techniques from [10–12], the protocol can be extended to work over rings. The challenge lies in performing interpolation where not all elements have an inverse over the ring $\mathbb{Z}_{2^\ell}$. To overcome this, the solution is to work over the extended ring $\mathbb{Z}_{2^\ell}[x]/f(x)$, i.e. the ring of all polynomials with coefficients in $\mathbb{Z}_{2^\ell}$ working modulo a polynomial f that is of the right degree and irreducible over $\mathbb{Z}_2$. When working over this extension rings, the number of roots of a polynomial is greater than its degree, and thus changes the error probability. For a protocol which verifies L values, the error is roughly $\frac{2\log L+4}{2^d}$, where d is the extension degree. We refer readers to [10, 11] for more details.

Communication cost. In the first $\log L - 1$ iterations, the prover shares 3 elements each. In the last round, it shares 5 elements, followed by public reconstruction of 4 elements via broadcast. Generation of randomness can be done non-interactively and does not incur any cost. Thus, the total communication cost is

$$6(\log L - 1) + 10 + 7 \text{ elements.}$$

Thus, the per party cost is approximately $\log L + 3$ elements.

Theorem F.1. *Protocol* $\Pi_{\mathsf{CheatIdentify}}$ *(Fig. 31) securely computes* $\mathcal{F}_{\mathsf{CheatIdentify}}$ *over field* $\mathbb{F}$ *in the* $(1,1)$*-FaF model with error* $\leq \frac{2\log L+4}{\mathbb{F}-5}$ *in the 5PC setting .*

Proof. Let $\mathcal{S}_{\mathcal{A}}$ be the ideal world malicious simulator, $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ be the ideal world semi-honest simulator, $\mathcal{A}$ be the real world malicious adversary and $\mathcal{A}_{\mathcal{H}}$ be the semi-honest real-world adversary. Consider the following cases.

Case 1: P_i is corrupt. In this case $\mathcal{S}_{\mathcal{A}}$ receives P_i's inputs and honest parties $[\cdot]$-shares of c. This implies that $\mathcal{S}_{\mathcal{A}}$ can perfectly simulate the opening of $[b]$ and $q(\mathrm{r})$ since it has the honest parties' $[\cdot]$-shares of c, and receives honest parties' $[\cdot]$-shares of points on $q(\cdot)$ from $\mathcal{A}$ during the simulation. We next show how to simulate the opening of $\vec{\mathbf{f}}_1(\mathrm{r}), \vec{\mathbf{f}}_2(\mathrm{r})$. For this, since $\mathcal{S}_{\mathcal{A}}$ knows the inputs of P_i, it knows the actual values of $\vec{\mathbf{f}}_1(\mathrm{r}), \vec{\mathbf{f}}_2(\mathrm{r})$. Thus, $\mathcal{S}_{\mathcal{A}}$ is only required to choose random values for shares of the honest parties while ensuring that together with P_i's shares, it opens to the correct values.

To see that the view of $\mathcal{A}$ is same here as in the real execution, observe that for each $e \in \{1,2\}$,

$$\vec{\mathbf{f}}_e(\mathrm{r}) = \vec{\lambda}_0(\mathrm{r}) \diamond \vec{\mathbf{f}}_e(0) + \vec{\lambda}_1(\mathrm{r}) \diamond \vec{\mathbf{f}}_e(1) + \vec{\lambda}_2(\mathrm{r}) \diamond \vec{\mathbf{f}}_e(2) \tag{5}$$

where $\vec{\lambda}_0(\mathrm{r}), \vec{\lambda}_1(\mathrm{r}), \vec{\lambda}_2(\mathrm{r})$ are the Lagrange coefficients. Since shares of $\vec{\mathbf{f}}_e(0)$ held by honest parties are random under the constraint that together with P_i's shares they open to $\vec{\mathbf{f}}_e(0)$, so are the shares of $\vec{\mathbf{f}}_e(\mathrm{r})$. Thus, the distribution is same in both the executions. If some honest party outputs reject, then $\mathcal{A}$ broadcasts an index j, which $\mathcal{S}_{\mathcal{A}}$ forwards to $\mathcal{F}_{\mathsf{CheatIdentify}}$. If out = reject, but honest parties output accept, then $\mathcal{S}_{\mathcal{A}}$ outputs fail and halts. Observe that when $\mathcal{S}_{\mathcal{A}}$ does not output fail, the simulation is perfect. The main difference is when $\mathcal{S}_{\mathcal{A}}$ outputs fail. This event occurs when Eq 3 does not hold, yet honest parties output accept. This occurs with probability $\leq \frac{2\log L+4}{\mathbb{F}-5}$, which is the error probability of the simulation. Finally, $\mathcal{S}_{\mathcal{A}}$ sends its view to $\mathcal{S}_{\mathcal{A},\mathcal{H}}$.

Subcase: P_j is semi-honest. P_j's view consists of (i) shares of $q(1), q(2), q(3)$ received in each of the $\log \bar{L} - 1$ iterations, (ii) shares of $q(0), q(1), \ldots, q(4)$ received in the last iteration, and, (iii) the shares received for reconstructing $b, q(\mathrm{r}), \vec{\mathbf{f}}_1(\mathrm{r}), \vec{\mathbf{f}}_2(\mathrm{r})$. While (i), (ii) are received as part of view of $\mathcal{S}_{\mathcal{A}}$, the (iii) can be simulated by sending random shares under the constraint that the reconstructed values are consistent with the ones in the view received from $\mathcal{S}_{\mathcal{A}}$. Thus the simulation is perfect.

Case 2: P_i is honest and P_j is corrupt. In this case, $\mathcal{S}_{\mathcal{A}}$ receives accept from $\mathcal{F}_{\mathsf{CheatIdentify}}$. This implies that although $\mathcal{S}_{\mathcal{A}}$ does not know the input, it knows that b should be 0 in each iteration and $q(\mathrm{r})$ should equal $g(\vec{\mathbf{f}}_1(\mathrm{r}), \vec{\mathbf{f}}_2(\mathrm{r}))$ in the last iteration, unless P_j misbehaves. Since $\mathcal{S}_{\mathcal{A}}$ knows P_j's shares of the inputs, it can simulate the openings correctly. Elaborately, for each sharing of $q(1), q(2)$ and $q(3)$ (and $\vec{\mathbf{f}}_1(0), \vec{\mathbf{f}}_2(0), q(0), \ldots, q(4)$ in the last step) in the simulation, $\mathcal{S}_{\mathcal{A}}$ sends random shares on behalf of P_i to $\mathcal{A}$. Since $\mathcal{S}_{\mathcal{A}}$ knows P_j's shares of $c, q(1), q(2)$, it can computes its shares of $b_l = c - q(1) - q(2)$. It then chooses the honest parties shares under the constraint that $b = \sum_{l=1}^{\log L} \gamma_l b_l$ will reconstruct to 0. Following this, $\mathcal{S}_{\mathcal{A}}$ uses P_j's shares of $\vec{\mathbf{f}}_1(e), \ldots, \vec{\mathbf{f}}_L(e)$ for $e \in \{1,2\}$, and $q(1), q(2), q(3)$ to compute P_j's shares of $\vec{\mathbf{f}}_1(\mathrm{r}), \ldots, \vec{\mathbf{f}}_L(\mathrm{r})$ and $q(\mathrm{r})$. Then, it can simulate the next iteration as before. Finally, $\mathcal{S}_{\mathcal{A}}$ uses P_j's shares of $\vec{\mathbf{f}}_1(0), \vec{\mathbf{f}}_1(1), \vec{\mathbf{f}}_1(2), \vec{\mathbf{f}}_2(0), \vec{\mathbf{f}}_2(1), \vec{\mathbf{f}}_2(2)$ and $q(0), \ldots, q(4)$ to compute P_j's shares of $\vec{\mathbf{f}}_1(\mathrm{r}), \vec{\mathbf{f}}_2(\mathrm{r}), q(\mathrm{r})$. $\mathcal{S}_{\mathcal{A}}$ simulates the opening of $b, \vec{\mathbf{f}}_1(\mathrm{r}), \vec{\mathbf{f}}_2(\mathrm{r}), q(\mathrm{r})$ as follows.

- To simulate opening of b, $\mathcal{S}_{\mathcal{A}}$ chooses random shares for the honest parties under the constraint that all the shares together will reconstruct to 0.
- To simulate the opening of $\vec{\mathbf{f}}_1(\mathrm{r}), \vec{\mathbf{f}}_2(\mathrm{r})$, $\mathcal{S}_{\mathcal{A}}$ chooses random shares for the honest parties.
- To simulate the opening of $q(\mathrm{r})$, $\mathcal{S}_{\mathcal{A}}$ chooses random shares for the honest parties under the constraint that the reconstructed $q(\mathrm{r})$ will satisfy the equation: $q(\mathrm{r}) = g(\vec{\mathbf{f}}_1(\mathrm{r}), \vec{\mathbf{f}}_2(\mathrm{r}))$.

If $\mathcal{A}$ sends consistent shares, $\mathcal{S}_{\mathcal{A}}$ sends out = accept to $\mathcal{F}_{\mathsf{CheatIdentify}}$. Else, since $\mathcal{S}_{\mathcal{A}}$ knows P_i's shares, it can compute the message that should have been sent by $\mathcal{A}$, and identifies the cheater on behalf of P_i. $\mathcal{S}_{\mathcal{A}}$ sends reject with index j to $\mathcal{F}_{\mathsf{CheatIdentify}}$ in this case.

We claim that $\mathcal{A}$'s view in the real and ideal execution is identically distributed. $\mathcal{A}$'s view consists of (i) shares sent by P_i for points on q, (ii) shares for points $\vec{\mathbf{f}}_1(0), \vec{\mathbf{f}}_2(0)$, (iii) the opened b, and (iv) the opened $\vec{\mathbf{f}}_1(\mathrm{r}), \vec{\mathbf{f}}_2(\mathrm{r}), q(\mathrm{r})$. Shares in (i) and (ii) are uniformly distributed, with respect to (iii), $\mathcal{A}$ sees random shares which open to 0 in both worlds. Finally, the claim in (iv) follows from Eq 5, similar to that in case 1, where $\vec{\mathbf{f}}_e(\mathrm{r})$ for $e \in \{1,2\}$ is randomly distributed in the ideal as well as the real world. Given that $\vec{\mathbf{f}}_e(\mathrm{r})$ for $e \in \{1,2\}$ is random, we obtain $q(\mathrm{r})$ being random as long as $q(\mathrm{r}) = g(\vec{\mathbf{f}}_1(\mathrm{r}), \vec{\mathbf{f}}_2(\mathrm{r}))$ holds.

Subcase: P_i is semi-honest. $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ has all inputs of P_i. Thus, the simulation can be carried out honestly, taking into consideration the view received from $\mathcal{S}_{\mathcal{A}}$. Thus, the simulation is perfect.

Subcase: P_k is semi-honest. This is similar to case 2. Since P_i is honest, b should be 0 in each iteration and $q(\mathrm{r})$ should equal $g(\vec{\mathbf{f}}_1(\mathrm{r}), \vec{\mathbf{f}}_2(\mathrm{r}))$ in the last iteration. Since $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ knows P_k's shares

of the inputs, it can simulate the openings correctly. Thus, the simulation is perfect. □

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

Let $\mathcal{S}_{\mathcal{A}}$ be an ideal world malicious adversary and $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ be the ideal world semi-honest adversary. The functionality is invoked by honest parties sending their $[\cdot]$-shares of m multiplication triples $\{(\mathsf{x}_k, \mathsf{y}_k, \mathsf{z}_k)_{k=1}^m\}$ to $\mathcal{F}_{\mathsf{Verify}}$.

(1) $\mathcal{F}_{\mathsf{Verify}}$ computes all secrets and corrupted party's shares, and sends these shares to $\mathcal{S}_{\mathcal{A}}$. $\mathcal{F}_{\mathsf{Verify}}$ sends $P_{\mathcal{H}}$'s shares to $\mathcal{S}_{\mathcal{A},\mathcal{H}}$, where $P_{\mathcal{H}}$ is controlled by $\mathcal{S}_{\mathcal{A},\mathcal{H}}$.
(2) $\mathcal{F}_{\mathsf{Verify}}$ verifies if $\mathsf{z}_k = \mathsf{x}_k \cdot \mathsf{y}_k$ for all $k \in \{1, 2, \ldots, m\}$.
  - If it holds, it sends accept to $\mathcal{S}_{\mathcal{A}}$.
  - Else, it sends reject to $\mathcal{S}_{\mathcal{A}}$ and $\mathsf{d}_k = \mathsf{z}_k - \mathsf{x}_k \cdot \mathsf{y}_k$ for each $k \in \{1, 2, \ldots, m\}$ such that $\mathsf{d}_k \neq 0$.
(3) If $\mathcal{F}_{\mathsf{Verify}}$ sent accept, it receives out $\in$ {accept, reject} from $\mathcal{S}_{\mathcal{A}}$, which is forwarded to the honest parties and $\mathcal{S}_{\mathcal{A},\mathcal{H}}$.
  - If out = reject, $\mathcal{S}_{\mathcal{A}}$ send a pair of indices (i, j) to $\mathcal{F}_{\mathsf{Verify}}$, where at least one among P_i, P_j is corrupt.
  - $\mathcal{F}_{\mathsf{Verify}}$ forwards (i, j) to honest parties and $\mathcal{S}_{\mathcal{A},\mathcal{H}}$.
(4) If $\mathcal{F}_{\mathsf{Verify}}$ sent reject, then $\mathcal{S}_{\mathcal{A}}$ does one of the following.
  (1) $\mathcal{S}_{\mathcal{A}}$ sends a pair of indices (i, j) to $\mathcal{F}_{\mathsf{Verify}}$, where at least one among P_i, P_j is corrupt. $\mathcal{F}_{\mathsf{Verify}}$ forwards (i, j) to honest parties and $\mathcal{S}_{\mathcal{A},\mathcal{H}}$.
  (2) $\mathcal{S}_{\mathcal{A}}$ asks $\mathcal{F}_{\mathsf{Verify}}$ to find a pair of conflicting parties in $\bar{k}^{\text{th}}$ multiplication, $1 \leq \bar{k} \leq m$. Next, $\mathcal{F}_{\mathsf{Verify}}$ asks the honest parties to send their inputs, randomness and views in the execution to compute the $\bar{k}^{\text{th}}$ triple. Based on the received information, $\mathcal{F}_{\mathsf{Verify}}$ computes the messages that should have been sent by the corrupted party, and finds a pair of parties P_i, P_j, where P_j received an incorrect message. $\mathcal{F}_{\mathsf{Verify}}$ sends (i, j) to honest parties, $\mathcal{S}_{\mathcal{A}}$ and $\mathcal{S}_{\mathcal{A},\mathcal{H}}$.
(5) $\mathcal{S}_{\mathcal{A}}$ sends its view to $\mathcal{S}_{\mathcal{A},\mathcal{H}}$.

Figure 32: Ideal functionality for verifying semi-honest protocol

F.0.2 The verification protocol. Using $\mathcal{F}_{\mathsf{CheatIdentify}}$, we next provide the protocol for verification of m multiplication triples with sublinear communication complexity in the number of multiplication triples. A multiplication triple is a shared triple $[\mathsf{x}], [\mathsf{y}], [\mathsf{z}]$ such that $\mathsf{z} = \mathsf{x} \cdot \mathsf{y}$. The ideal functionality for the same appears in Fig. 32. When verification fails, the functionality either obtains a pair of conflicting parties, one of which is guaranteed to be corrupt, from the adversary; or it identifies this pair by itself. In the latter case, the functionality obtains the inputs, randomness and views of honest parties when computing some incorrect multiplication triple, and uses this information to identify a pair of conflicting parties.

The protocol for $\mathcal{F}_{\mathsf{Verify}}$. To compute the functionality, the parties take a random linear combination

$$\beta = \sum_{k=1}^{m} \theta_k \cdot (\mathsf{z}_k - \mathsf{x}_k \cdot \mathsf{y}_k)$$

where θ_k is randomly chosen by all the parties and want to check if $\beta = 0$. Since β is a degree-2 function of $\{(\mathsf{x}_k, \mathsf{y}_k, \mathsf{z}_k)_{k=1}^m\}$ which are $[\cdot]$-shared, parties can compute an additive sharing ($\langle\cdot\rangle$-sharing) of β. Using the $\langle\cdot\rangle$-shares, parties can reconstruct β and check for equality with 0. However, since $\langle\cdot\rangle$-sharing does not allow for robust reconstruction, the parties first $[\cdot]$-share their additive shares of $\psi = \sum_{k=1}^m \theta_k \cdot \mathsf{x}_k \cdot \mathsf{y}_k$. Let ψ^i denote the additive share of ψ held by P_i. The consistency check in the $[\cdot]$-sharing protocol ensures that all receive consistent $[\cdot]$-shares of ψ^i. In case of a failure, the dealer broadcasts the share for which pairwise inconsistency exits. Given $[\psi^i]$ for $i \in \{1, \ldots, 5\}$, parties can compute

$$[\beta] = \sum_{k=1}^{m} \theta_k \cdot [\mathsf{z}_k] - \sum_{i=1}^{5} [\psi^i]$$

and reconstruct β. It is, however, required to ensure that every party P_i shares the correct value ψ^i. Towards realizing this, the property of $[\cdot]$-sharing, which allows parties to locally convert from $[\mathsf{x}_k], [\mathsf{y}_k]$ to $[\vec{\mathbf{x}}_k^i], [\vec{\mathbf{y}}_k^i]$, where $\vec{\mathbf{x}}_k^i, \vec{\mathbf{y}}_k^i$ are the vector of $[\cdot]$-shares of $\mathsf{x}_k, \mathsf{y}_k$, held by P_i, respectively, is used. Parties now want to verify if

$$\forall i \in \{1, \ldots, 5\} : \sum_{k=1}^{m} \theta_k \cdot \left([\vec{\mathbf{x}}_k^i] \diamond [\vec{\mathbf{y}}_k^i] \right) - [\psi^i] = 0$$

Letting $[c^i] = [\psi^i]$, $[\vec{\mathbf{a}}_k^i] = \theta_k \cdot [\vec{\mathbf{x}}_k^i]$ and $[\vec{\mathbf{b}}_k^i] = [\vec{\mathbf{y}}_k^i]$, one needs to verify that $[c^i] - \sum_{k=1}^m [\vec{\mathbf{a}}_k^i] \diamond [\vec{\mathbf{b}}_k^i] = 0$ for $i \in \{1, \ldots, 5\}$. This can be verified using $\mathcal{F}_{\mathsf{CheatIdentify}}$. In case of a reject, $\mathcal{F}_{\mathsf{CheatIdentify}}$ outputs a pair of conflicting parties. Otherwise, parties proceed with reconstructing β. If reconstruction fails due to inconsistency, pairwise consistency check of $[\cdot]$-sharing is used to identify a pair of conflicting parties, where the consistency check is carried out over a broadcast channel. Finally, if $\beta \neq 0$, then it implies that no one cheated in the verification protocol (with high probability), and one of the multiplication triples is incorrect. Parties localize the fault by running a binary search on the multiplication triples to identify a triple where $\mathsf{z}_k \neq \mathsf{x}_k \cdot \mathsf{y}_k$. In each search step, the verification protocol is carried out on half the number of triples until one incorrect triple is identified. Finally, parties check the execution of the multiplication protocol for this triple to find a pair of disputing parties. This is done by invoking a functionality $\mathcal{F}_{\mathsf{miniMPC}}$ which takes the inputs, randomness and view of parties in the multiplication protocol as input and outputs the pair of parties for which the incoming and outgoing messages do not match. The protocol appears in Fig. 33.

Cheating probability over finite fields. Assume that there is an incorrect triple. If the adversary does not cheat in the verification protocol, then there will at most $\log m$ executions. In each execution, the probability that the test will pass is $\frac{1}{\mathbb{F}}$ which happens when the random linear combination outputs a value 0. Thus, the overall cheating probability is bounded by $\log m \cdot \frac{1}{\mathbb{F}}$.

Communication cost. Protocol Π_{Verify} is recursive. In the jth step, parties secret share one element each, reconstruct one element, and call $\mathcal{F}_{\mathsf{CheatIdentify}}$ for every party over a set of triples of size $m/2^j$. Thus, the total communication cost in the jth step is

$$5 \cdot 2 + 7 + 5 \cdot \left(6(\log(m/2^j) - 1) + 17\right)$$
$$= 97 + 30 \cdot \log(m/2^j) \text{ elements.}$$

Protocol $\Pi_{\mathsf{Verify}}\left(\mathcal{P}, \{([\mathsf{x}_k], [\mathsf{y}_k], [\mathsf{z}_k])\}_{k=1}^{m}\right)$

(1) Parties generate random values $\theta_1, \ldots, \theta_m \in \mathbb{F}$.
(2) Parties locally compute

$$\langle \psi \rangle = \left\langle \sum_{k=1}^{m} \theta_k \cdot \mathsf{x}_k \cdot \mathsf{y}_k \right\rangle = \sum_{k=1}^{m} \theta_k \cdot ([\mathsf{x}_k] \diamond [\mathsf{y}_k])$$

(3) Let the $\langle\cdot\rangle$-share of ψ held by P_i be ψ^i. Each party P_i shares ψ^i among other parties.
(4) For each $i \in \{1, 2, \ldots, 5\}$:
   - Parties locally convert $[\mathsf{x}_k], [\mathsf{y}_k]$ to $\left[\vec{\mathsf{x}}^{\mathbf{i}}_{\mathbf{k}}\right], \left[\vec{\mathsf{y}}^{\mathbf{i}}_{\mathbf{k}}\right]$ for each $k \in \{1, 2, \ldots, m\}$.
   - Parties define $\left[c^i\right] = \left[\psi^i\right]$, $\left[\vec{\mathsf{a}}^{\mathbf{i}}_{\mathbf{k}}\right] = \theta_k \cdot \left[\vec{\mathsf{x}}^{\mathbf{i}}_{\mathbf{k}}\right]$ and $\left[\vec{\mathsf{b}}^{\mathbf{i}}_{\mathbf{k}}\right] = \theta_k \cdot \left[\vec{\mathsf{y}}^{\mathbf{i}}_{\mathbf{k}}\right]$.
   - Parties send $\left[c^i\right]$ and $\left(\left[\vec{\mathsf{a}}^{\mathbf{i}}_{\mathbf{k}}\right], \left[\vec{\mathsf{b}}^{\mathbf{i}}_{\mathbf{k}}\right]\right)_{k=1}^{m}$ to $\mathcal{F}_{\mathsf{CheatIdentify}}$.
   - If parties receive reject, (i, j) from $\mathcal{F}_{\mathsf{CheatIdentify}}$, then they output it and halt.
(5) If parties received accept from $\mathcal{F}_{\mathsf{CheatIdentify}}$ in all five invocations, they proceed to the next step.
(6) Parties locally compute $[\beta] = \sum_{k=1}^{m} \theta_k \cdot [\mathsf{z}_k] - \sum_{i=1}^{5} \left[\psi^i\right]$.
(7) Parties robustly reconstruct β by sending their shares via broadcast.
   - If parties see inconsistent shares, they output reject, (i, j), where P_i, P_j is the first pair of parties for which pair-wise inconsistency exists.
   - If $\beta = 0$, parties output accept.
   - If $\beta \neq 0$, parties perform a fault localization procedure to identify the first incorrect triple by running a binary search on the input triples. For this search, parties run the above protocol on two half-sized sets of input triples, and proceed as follows.
     - If parties output accept in both executions, they output accept and halt.
     - If any execution ends with parties holding a pair of conflicting parties (i, j), parties output reject, (i, j) and halt.
     - If $\beta \neq 0$ in both the executions, they continue the search on one of the sets.
     - If $\beta \neq 0$ in one of the executions, they continue the search on the set for which $\beta \neq 0$.

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

Figure 33: Realizing $\mathcal{F}_{\mathsf{Verify}}$

In the worst case, there are $\log m$ steps, and the total cost is

$$97 \cdot \log m + 30 \cdot \sum_{j=1}^{\log m} \log(m/2^j)$$

Since $\sum_{j=1}^{\log m} \log(m/2^j) \leq \log m \cdot \log \sqrt{m}$, the total communication cost is

$$97 \cdot \log m + 30 \cdot \log m \cdot \log \sqrt{m} \text{ elements.} \quad (6)$$

Note that while working over extended rings, the cost gets multiplied by a factor d, which is the degree of the extension.

Theorem F.2. *Protocol Π_{Verify} (Fig. 33) securely computes $\mathcal{F}_{\mathsf{Verify}}$ over field $\mathbb{F}$ in the $(1,1)$-FaF model with error $\log m \cdot \frac{1}{\mathbb{F}}$ in the $(\mathcal{F}_{\mathsf{miniMPC}}, \mathcal{F}_{\mathsf{CheatIdentify}})$-hybrid model in the 5PC setting.*

Proof. Let $\mathcal{S}_{\mathcal{A}}$ be the ideal world malicious simulator, $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ be the ideal world semi-honest simulator and let $\mathcal{A}$ be the real world malicious adversary and $\mathcal{A}_{\mathcal{H}}$ be the real world semi-honest adversary. $\mathcal{S}_{\mathcal{A}}$ is invoked by $\mathcal{F}_{\mathsf{Verify}}$ which sends it the corrupted party's shares of $(\mathsf{x}_k, \mathsf{y}_k, \mathsf{z}_k)_{k=1}^{m}$ and out $\in \{\text{accept}, \text{reject}\}$ and $\mathsf{d}_k = \mathsf{z}_k - \mathsf{x}_k \cdot \mathsf{y}_k$ for $k \in \{1, 2, \ldots, m\}$. Further, $\mathcal{F}_{\mathsf{Verify}}$ sends $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ the shares for $P_{\mathcal{H}}$, which is the semi-honest party.

Random $\theta_1, \ldots, \theta_m \in \mathbb{F}$ are generated. $\mathcal{S}_{\mathcal{A}}$ plays the role of $\mathcal{F}_{\mathsf{CheatIdentify}}$ and $\mathcal{F}_{\mathsf{miniMPC}}$. Similar to the proof of Theorem F.1, $\mathcal{S}_{\mathcal{A}}$ chooses random shares for corrupted party for each ψ^j, where P_j is honest and hands these to $\mathcal{A}$. Then, $\mathcal{S}_{\mathcal{A}}$ receives the honest parties' shares for ψ^i, where P_i is the maliciously corrupt party. If the shares dealt by $\mathcal{A}$ are inconsistent, then the consistency check takes care of this. The presence of honest majority enables $\mathcal{S}_{\mathcal{A}}$ to use the honest parties' shares to compute ψ^i for the corrupt P_i and its shares. Thus, for each $i \in \{1, 2, \ldots, 5\}$, $\mathcal{S}_{\mathcal{A}}$ can simulate $\mathcal{F}_{\mathsf{CheatIdentify}}$, handing accept or reject to $\mathcal{A}$, accordingly. If the output is reject for any $i \in \{1, 2, \ldots, 5\}$, then $\mathcal{A}$ sends index of a party P_j to $\mathcal{S}_{\mathcal{A}}$, which together with P_i forms a disputed pair of parties. Then, $\mathcal{S}_{\mathcal{A}}$ sends reject, (i, j) to $\mathcal{F}_{\mathsf{Verify}}$, outputs whatever $\mathcal{A}$ outputs and halts.

If the simulation has not ended with a reject, then it means that all ψ^i's are correct. Thus, $\mathcal{S}_{\mathcal{A}}$ can compute $\beta = \sum_{k=1}^{m} \theta_k \cdot (\mathsf{z}_k - \mathsf{x}_k \cdot \mathsf{y}_k) = \sum_{k=1}^{m} \theta_k \cdot \mathsf{d}_k$ and choose random shares for the honest parties, given the value of β and the corrupted party's shares (known to $\mathcal{S}_{\mathcal{A}}$). Using these shares, $\mathcal{S}_{\mathcal{A}}$ simulates the reconstruction procedure. Consider the following cases.

- If $\mathcal{A}$ sent incorrect shares, causing the opening of β to fail, then $\mathcal{S}_{\mathcal{A}}$ takes the first pair of parties P_i, P_j for which pair-wise inconsistency occurred, and sends reject, (i, j) to $\mathcal{F}_{\mathsf{Verify}}$, outputs whatever $\mathcal{A}$ outputs and halts.
- If $\beta = 0$: if out = reject (honest parties output accept in this case), $\mathcal{S}_{\mathcal{A}}$ outputs fail and halts; if out = accept, $\mathcal{S}_{\mathcal{A}}$ sends accept to $\mathcal{F}_{\mathsf{Verify}}$, outputs whatever $\mathcal{A}$ outputs and halts.
- If $\beta \neq 0$, simulation proceeds to the binary search, where $\mathcal{S}_{\mathcal{A}}$ simulates each steps as described so far. If a pair of disputed parties is located, then it is sent to $\mathcal{F}_{\mathsf{Verify}}$. If honest parties output accept, then $\mathcal{S}_{\mathcal{A}}$ outputs fail (here it must hold that out = reject, since otherwise the simulation would not have reached the binary search phase). If parties found an incorrect triple $\mathsf{x}_{\bar{k}}, \mathsf{y}_{\bar{k}}, \mathsf{z}_{\bar{k}}$ such that $\mathsf{z}_{\bar{k}} \neq \mathsf{x}_{\bar{k}} \cdot \mathsf{y}_{\bar{k}}$ without identifying a disputed pair, then $\mathcal{S}_{\mathcal{A}}$ asks $\mathcal{F}_{\mathsf{Verify}}$ to find such a pair by sending it $\bar{k}$. Upon receiving (i, j) from $\mathcal{F}_{\mathsf{Verify}}$, $\mathcal{S}_{\mathcal{A}}$ simulates $\mathcal{F}_{\mathsf{miniMPC}}$, handing (i, j) to $\mathcal{A}$. Finally, $\mathcal{S}_{\mathcal{A}}$ outputs whatever $\mathcal{A}$ outputs. Note that an event where the $\bar{k}^{\text{th}}$ triple is correct is not possible, because in this case β must be equal to 0.

$\mathcal{A}$'s view consists of (i) random shares of β^j for each honest party P_j, (ii) message sent by $\mathcal{F}_{\mathsf{CheatIdentify}}$, (iii) the revealed β, and (iv) message from $\mathcal{F}_{\mathsf{miniMPC}}$. The argument for identical distribution of $\mathcal{A}$'s view in (i), (ii), (iii) follows from the proof of Theorem F.1. For (iv), since $\mathcal{S}_{\mathcal{A}}$ receives a pair of parties with conflicting views in the computation of the $\bar{k}^{\text{th}}$ triple from $\mathcal{F}_{\mathsf{Verify}}$, it can simulate

the role of $\mathcal{F}_{\text{miniMPC}}$ perfectly. Thus, the only difference between the simulation and real-execution is the event where $\mathcal{S}_{\mathcal{A}}$ outputs fail. This happens when $\exists k \in \{1,2,\ldots,m\} : \mathsf{d}_k \neq 0$ (which is why out = reject) but the parties eventually output accept. This occurs when $\beta = 0$ in one of binary search steps. Since there are $\log m$ steps and $\Pr[\beta = 0] = \frac{1}{\mathbb{F}}$ in each step, we have that $\Pr[\text{fail}] \leq \frac{\log m}{\mathbb{F}}$, which is the error in the simulation.

Following this, $\mathcal{S}_{\mathcal{A}}$ sends its view to $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ to simulate the view for $\mathcal{A}_{\mathcal{H}}$. $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ chooses random shares for corrupted party for each ψ^j, where P_j is honest and hands these to $\mathcal{A}_{\mathcal{H}}$. Then, $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ receives the honest parties' shares for $\psi^{\mathcal{H}}$, where $P_{\mathcal{H}}$ is the semi-honest party. The presence of honest majority enables $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ to use the honest parties' shares to compute $\psi^{\mathcal{H}}$ for $P_{\mathcal{H}}$ and its shares. Thus, for each $i \in \{1,2,\ldots,5\}$, $\mathcal{S}_{\mathcal{A}}$ simulates $\mathcal{F}_{\text{CheatIdentify}}$, handing accept or reject to $\mathcal{A}_{\mathcal{H}}$ according to $\mathcal{S}_{\mathcal{A}}$'s view. If the output is reject for any $i \in \{1,2,\ldots,5\}$, then $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ sends reject, (i,j) to $\mathcal{A}_{\mathcal{H}}$, as present in $\mathcal{S}_{\mathcal{A}}$'s view. $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ simulates the reconstruction procedure for β using shares received from $\mathcal{A}_{\mathcal{H}}$. Now, depending on the view received from $\mathcal{S}_{\mathcal{A}}$, $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ sends (i,j) or accept to $\mathcal{A}_{\mathcal{H}}$. The argument for indisntinguishability of the views of $\mathcal{A}_{\mathcal{H}}$ in real and ideal world follows similar to the argument for $\mathcal{A}$. □

Similar to $\Pi_{\text{CheatIdentify}}$, the protocol Π_{Verify} can also be extended to work over the ring $\mathbb{Z}_{2^\ell}$ (see F.0.1).

F.0.3 The main protocol. We now provide details of the main protocol for computing the multiplication triples in the preprocessing phase. The ideal functionality for the same appears in Fig. 34. We remark that operating in the preprocessing model, we can generate a large number of multiplication triples at the same time which also helps in amortizing the cost due to verification. The main protocol begins with executing a semi-honest 5PC protocol, followed by a verification phase to check the correctness of the multiplication triples generated during the semi-honest execution. Verification completes with it either being a success or outputting a pair of conflicting parties (in which case a semi-honest 3PC is executed as described earlier). The protocol appears in Fig. 35.

Functionality $\mathcal{F}_{\text{MulPre}}$

Let $\mathcal{S}_{\mathcal{A}}$ be an ideal world malicious adversary and $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ be the ideal world semi-honest adversary.

(1) $\mathcal{F}_{\text{MulPre}}$ interacts with the parties in $\mathcal{P}$ and the adversaries $\mathcal{S}_{\mathcal{A}}$, $\mathcal{S}_{\mathcal{A},\mathcal{H}}$. $\mathcal{F}_{\text{MulPre}}$ receives $[\cdot]$-shares of $\{(\mathsf{x}_k, \mathsf{y}_k)_{k=1}^m\}$ from honest parties.
(2) $\mathcal{F}_{\text{MulPre}}$ receives $([\mathsf{x}_k \cdot \mathsf{y}_k]_i)_{k=1}^m$ from $\mathcal{S}_{\mathcal{A}}$ where P_i is controlled by $\mathcal{S}_{\mathcal{A}}$. It also receives continue or (abort, j) from $\mathcal{S}_{\mathcal{A}}$. If received abort, $\mathcal{F}_{\text{MulPre}}$ sends (i,j) to all. Else, it does the following.
   - $\mathcal{F}_{\text{MulPre}}$ reconstructs $\mathsf{x}_k, \mathsf{y}_k$ using the honest parties' shares and computes $\mathsf{x}_k \cdot \mathsf{y}_k$ for $k \in \{1,\ldots,m\}$.
   - $\mathcal{F}_{\text{MulPre}}$ generates $[\mathsf{x}_k \cdot \mathsf{y}_k]$, for $k \in \{1,2,\ldots,m\}$, using $\mathsf{x}_k \cdot \mathsf{y}_k$ and $[\mathsf{x}_k \cdot \mathsf{y}_k]_i$ received from $\mathcal{S}_{\mathcal{A}}$.
   - $\mathcal{F}_{\text{MulPre}}$ sends $(\text{Output}, [\mathsf{x}_k \cdot \mathsf{y}_k]_s)$ to $P_s \in \mathcal{P}$.
(3) $\mathcal{S}_{\mathcal{A}}$ sends its view to $\mathcal{S}_{\mathcal{A},\mathcal{H}}$.

Figure 34: Ideal functionality for computing multiplication triples in the preprocessing

Protocol $\Pi_{\text{mulPre}}\left(\mathcal{P}, \{[\mathsf{x}_k], [\mathsf{y}_k]\}_{k=1}^m\right)$

(1) Parties generate $[\cdot]$-shares of random values $\mathsf{r}_1, \mathsf{r}_2, \ldots, \mathsf{r}_m$, non-interactively using their shared key setup. They locally convert $[\cdot]$-shares to $\langle\cdot\rangle$-shares.
(2) Parties locally compute $\langle \mathsf{x}_k \cdot \mathsf{y}_k - \mathsf{r}_k \rangle = [\mathsf{x}_k] \diamond [\mathsf{y}_k] - \langle \mathsf{r}_k \rangle$ for each $k \in \{1,2,\ldots,m\}$ and send it to P_1.
(3) P_1 reconstructs $\mathsf{x}_k \cdot \mathsf{y}_k - \mathsf{r}_k$ for each $k \in \{1,2,\ldots,m\}$ and generates $[\mathsf{x}_k \cdot \mathsf{y}_k - \mathsf{r}_k]$ using $\Pi_{\text{RSS-Sh}}$ (Fig. 11).
(4) Parties compute $[\mathsf{x}_k \cdot \mathsf{y}_k] = [\mathsf{x}_k \cdot \mathsf{y}_k - \mathsf{r}_k] + [\mathsf{r}_k]$ for $k \in \{1,2,\ldots,m\}$.
(5) Parties invoke $\Pi_{\text{Verify}}\left(\mathcal{P}, \{([\mathsf{x}_k], [\mathsf{y}_k], [\mathsf{x}_k \cdot \mathsf{y}_k])\}_{k=1}^m\right)$ to verify the correctness of the multiplication triples.
(6) If parties receive accept from Π_{Verify}, they proceed with the online phase. Else, parties obtain a pair of parties (P_i, P_j) to eliminate from Π_{Verify}.

Figure 35: $(1,1)$-FaF secure protocol for 5PC preprocessing phase of multiplication

Communication cost. The communication cost follows from the cost of the semi-honest protocol and the cost of the verification protocol. The semi-honest protocol requires communicating 6 ring elements. The cost due to the verification phase can be amortized by preprocessing a large number of multiplication triples. Concretely, for verifying 2^{25} multiplication triples, the cost for verification is only 0.003 ring elements for an extension degree $d = 46$ (see Table 4 of full (eprint) version of [12]). Table 7 summarizes the communication cost for various number of multiplication triples to be verified.

m	Cost (per party per multiplication)
2^{10}	22.1914
2^{20}	0.0696
2^{25}	0.0032
2^{30}	0.0001

Table 7: Cost of verification in terms of the number of ring elements communicated per party per multiplication, and 40 bits of statistical security. Here, m - #multiplication triples to be verified and degree of extension $d = 46$ to achieve statistical security of 2^{-40}.

Theorem F.3. *Protocol Π_{Pre} (Fig. 35) securely computes $\mathcal{F}_{\text{Pre}}$ over the field $\mathbb{F}$ or ring $\mathbb{Z}_{2^\ell}$ in the $(1,1)$-FaF model in the $\mathcal{F}_{\text{Verify}}$-hybrid model in the 5PC setting.*

Proof. Consider the case of a corrupt P_1. $\mathcal{S}_{\mathcal{A}}$ generates $[\cdot]$-shares for $\{\mathsf{x}_k, \mathsf{y}_k, \mathsf{r}_k\}_{k=1}^m$, and learns these values on clear. Step 3 of the protocol is simulated by sending random values to $\mathcal{A}$. $\mathcal{S}_{\mathcal{A}}$ also computes the secret $\mathsf{x}_k \cdot \mathsf{y}_k$ for $k \in \{1,2,\ldots,m\}$. If inconsistent shares are received in step 4 from $\mathcal{A}$, then $\mathcal{S}_{\mathcal{A}}$ detects the inconsistency, and the simulation outputs a pair of conflicting parties. Else, if the shares are consistent but the correct output is not received, $\mathcal{S}_{\mathcal{A}}$ computes the difference between these values and simulates $\mathcal{F}_{\text{Verify}}$. If cheating took place, then it sens reject and $\mathsf{d}_k \neq 0$ to $\mathcal{A}$. Then, it waits to receive from $\mathcal{A}$ either a pair of conflicting parties or a request to $\mathcal{F}_{\text{Verify}}$ to find such a pair. In the latter case, $\mathcal{S}_{\mathcal{A}}$ finds such a pair of conflicting parties by computing the messages that should have been sent by the corrupted party and compares it with

what was received from $\mathcal{A}$. Then, $\mathcal{S}_{\mathcal{A}}$ sends the obtained pair to $\mathcal{A}$. If no cheating took place, then $\mathcal{S}_{\mathcal{A}}$ sends accept to $\mathcal{A}$. Following this, $\mathcal{A}$ can decide to reject, in which case a pair of conflicting parties is sent as output. Observe that since $\mathcal{A}$'s view consists of random shares in both the worlds, the views are identical. Then, $\mathcal{S}_{\mathcal{A}}$ sends its view to $\mathcal{S}_{\mathcal{A},\mathcal{H}}$. Simulation by $\mathcal{S}_{\mathcal{A},\mathcal{H}}$ for a semi-honest party follows trivially as there are no messages to simulate other than those from P_1 which are already received as part of $\mathcal{A}$'s view.

Cases where other parties are corrupt can be simulated trivially. Simulation for semi-honest P_1 also follows. □