Scalable Multiparty Garbling

Gabrielle Beck[1], Aarushi Goel[2], Aditya Hegde[1], Abhishek Jain[1], Zhengzhong Jin[3],
Gabriel Kaptchuk[4]

[1]Johns Hopkins University, {becgabri, ahegde, abhishek}@cs.jhu.edu
[2]NTT Research, aarushi.goel@ntt-research.com
[3]Massachusetts Institute of Technology, zzjin@mit.edu
[4]Boston University, kaptchuk@bu.edu

Abstract

Multiparty garbling is the most popular approach for constant-round secure multiparty computation (MPC). Despite being the focus of significant research effort, instantiating prior approaches to multiparty garbling results in constant-round MPC that can not realistically accommodate large numbers of parties. In this work we present the first global-scale multiparty garbling protocol. The per-party communication complexity of our protocol *decreases* as the number of parties participating in the protocol increases—for the first time matching the asymptotic communication complexity of non-constant round MPC protocols. Our protocol achieves malicious security in the *honest-majority* setting and relies on the hardness of the Learning Party with Noise assumption.

Contents

1 Introduction

Secure multiparty computation (MPC) [Yao86, GMW87, CCD88, BGW88] is a class of cryptographic protocols that allows mutually distrusting parties to compute a function over hidden inputs. Since the eighties—when the first feasibility results were established—continuous progress has been made towards improving the efficiency of MPC protocols along various dimensions. Modern protocols can support large and complex computations [BGG⁺20, GHK⁺21, CGG⁺21] and when running on local area networks, even a large number of participants [CGH⁺18, WJS⁺19, GS20, GSY21, BGJK21, GPS21]. Such improvements have resulted in the creation of toolchains for MPC (e.g., [MNPS04, BNP08, DSZ15, MR18, HHNZ19]) and deployment of some real-world applications (e.g., [BGG19, QLJ⁺19]).

Global-Scale MPC. As enthusiasm for MPC grows, there will be an increased need for protocols that can support global-scale applications. For example, consider hundreds or thousands of parties distributed around the globe who wish to evaluate a complex functionality. In such a deployment, network latency between parties becomes a key determinant of the protocol runtime (see, e.g. [WRK17b] for a discussion). High latency is particularly detrimental for MPC protocols where the parties jointly compute circuits in a gate-by-gate fashion, as these protocols require communication rounds proportional to the *depth of the circuit*.

Constant round MPC protocols [Yao86, BMR90, DI05, GGJS12, LPSY15, KOS16, BLO17, NST17, HSS17, WRK17b, HOSS18b, HOSS18a, BCO⁺21] are more appropriate for high latency settings, as the number of times parties must communicate is independent of the circuit size. There are two well-studied approaches to constant-round MPC—one based on fully-homomorphic encryption [Gen09] and another based on garbled circuits [Yao86, BMR90]. The latter has been studied more extensively because of its better potential for efficient solutions (despite incurring asymptotically worse communication).

The second approach follows a template first proposed by Beaver, Micali, and Rogaway (BMR) [BMR90]: the parties first execute a *garbling phase*, in which they jointly compute a garbled circuit of the desired functionality within an MPC protocol. The garbling phase is then followed by an *output evaluation phase*, in which the parties exchange inputs and evaluate the garbled circuit. Since garbled gates can be computed in parallel, the resulting protocol has constant rounds. Throughout this work we refer to the process of jointly computing the garbled circuit as *multiparty garbling*.

Barriers to Efficient, Scalable Multiparty Garbling. Although multiparty garbling is a well-studied approach for constant-round MPC, existing proposals cannot realistically be used to perform global-scale computations. Asymptotically efficient constructions of the BMR template can be obtained by making non-black-box use of cryptography used during garbling [BMR90], but representing the cryptography as circuitry introduces prohibitive overheads.

The best known black-box multiparty garbling protocols, on the other hand, require per-gate total communication (and computation) that is *quadratic in the number of parties* [DI05, LPSY15, BLO17, HSS17, WRK17b, HOSS18b, HOSS18a, BCO⁺21], meaning these protocols scale poorly with the number of parties. As is common in the literature on efficient MPC, these works split the garbling phase into a circuit independent *pre-processing phase* and a circuit dependent *garbling phase*. Recent works [BLO17, BCO⁺21] have demonstrated methods that reduce the complexity of the circuit-dependent garbling phase and output evaluation phase to be linear in the number of parties, but still require a pre-processing phase with quadratic complexity.

Thus, overall, the quadratic barrier stands for efficient multiparty garbling—in both the honest and dishonest majority settings. This poses a major barrier for global-scale computations; the combination of the quadratic dependence on the number of parties and the fact that garbling inevitably increases the circuit size by a multiplicative factor dependent on the security parameter results in impractical solutions, even for moderately-sized circuits. We therefore ask the following question:

Does there exist an efficient and scalable multiparty garbling protocol?

We answer this question in the affirmative, taking a significant step towards efficient, global-scale, constant-round MPC.

1.1 Our Contributions

We present a new *scalable* constant-round multiparty garbling protocol for boolean circuits in the honest-majority setting, where the total per-party communication complexity (see below) in our protocol *decreases* as the number of parties increase. To produce this protocol, we combined several recent advances in efficient MPC and carefully compose them using bespoke subprotocols. In more detail, our protocol has the following features:

- **Communication Complexity:** The total communication complexity of the protocol is independent of the number of parties (i.e., is $O(|\mathsf{C}|)$,[1] where C is the circuit being computed), meaning that the per-party communication actually *decreases* as the number of parties increases. Similar to prior constant round MPC protocols [BMR90, DI05, LPSY15, BLO17, NST17, HSS17, WRK17b, HOSS18b, HOSS18a, BCO$^+$21], our protocol utilizes a single round of broadcast to reconstruct the circuit to all parties, but otherwise runs over point-to-point channels.[2]

- **Computation Complexity:** The per-party computation complexity of the protocol is independent of the number of parties (i.e., the total computation complexity of the protocol is $O(n|\mathsf{C}|)$). This computational complexity is inherent in the constant-round BMR template, as each party needs to *evaluate* the garbled circuit.

- **Security and Assumptions:** Our protocol achieves *malicious security* against $t < \frac{n-2\ell+1}{2}$ corrupt parties, where $\ell \geq 1$ is a tunable parameter induced by the use of packed secret sharing[3]. The security of our construction relies on the Learning Parity with Noise over Large Fields (LPN) assumption. The use of LPN in our construction demonstrates that “less-powerful” assumptions (i.e., ones that are not known to imply FHE) are sufficient for designing efficient and scalable constant round MPC.

Our protocol is the first *constant-round MPC* to asymptotically match the best known communication complexity of concretely efficient, gate-by-gate MPC protocols without using fully-homomorphic encryption.[4] As such, our protocol demonstrates a path towards practical MPC in high-latency settings, where gate-by-gate protocols typically struggle.

Evaluation and Analysis. To estimate the efficiency of our protocol, we do a series of simulations to calculate the communication cost. We focus on communication costs since the setting of our work is high latency networks, where computation is cheap and end devices' throughput is assumed to be fundamentally limited by the bandwidth of the shortest link along their path. We provide detailed analysis for both the pre-processing phases and garbling phases of our protocol and compare against constant round protocols with worse asymptotic communication complexity [WRK17b, BCO$^+$21]. We find that our protocol has the smallest communication complexity during the garbling phase for $n \geq 250$ parties. Additionally, we show that for large numbers of parties, our protocol offers significant benefits over each of the prior works we consider, when seen holistically: our protocol's communication complexity during the pre-processing phrase is dramatically less that that of Ben-Efraim et al. [BCO$^+$21] (although our online communication

[1] The $O(\cdot)$ notation suppresses linear terms in the security parameter and other logarithmic terms (independent of circuit size).

[2] We note that because the broadcast channel is used only in the final round, this broadcast channel is particularly well suited to implementation via a website, where parties can post their shares and then download the garbled circuit at a later time.

[3] ℓ corresponds to the number of secrets packed into one share. We refer the reader to Section 3.2 for more details.

[4] Up to a security parameter factor, introduced from encrypting each gate.

complexity is roughly the same), and the communication complexity of the garbling phase is dramatically less than Wang et al. [WRK17b] (although our pre-processing communication complexity is roughly the same). As such, our protocol allows us to avoid the trade-offs in efficiency between pre-processing and online that appears in prior work.

1.2 Related Work

There is a long history of pushing towards $O(|\mathrm{C})|)$ MPC [DIK10, DIK+08, GIP15], that has recently resulted in *linear round* (ie. communication rounds linear in the depth of the circuit) MPC protocols with $O(|\mathrm{C})|)$ communication complexity that are concretely efficient [BGJK21, GSY21, GPS21, GPS22]. Our work builds on some of the techniques proposed in these works, but applies them to the constant-round setting. Most relevant to our protocol, we use the share transformation protocol proposed by Goyal et al. [GPS22] (see Section 2.6) as a subprotocol in order to achieve our result.

There are two popular templates for achieving constant round MPC. The first relies on multiparty variants of fully homomorphic encryption [MSs11, AJL+12, GLS15, MW16, BHP17]. While improving the efficiency of FHE is an active area of research, this approach currently remains very far from practical. The second template, first proposed by Beaver, Micali and Rogaway (BMR) [BMR90] relies on the observation that garbling a circuit [Yao86] can be performed in constant depth. In our work, we focus on this second approach: the problem of multiparty garbling.

The BMR approach has been the subject of significant research and has recently lead to asymptotically efficient constructions with garbled circuit specifications that can be evaluated quickly in practice [DI05, GGJS12, LPSY15, LSS16, BLO16, BLO17, NST17, HSS17, WRK17b, HOSS18b, HOSS18a, HSS20, BOSS20, BCO+21]. While the original approach required non-black-box use of cryptography, Damgård and Ishai [DI05] proposed a black-box technique for multi party garbling, paving the way towards more efficient constructions.

Of particular interest, Ben-Efraim, Lindell, and Omri [BLO17] showed how to leverage LWE to garble a circuit with an evaluation complexity of $O(n|C|)$ per-party, improving on the prior $O\left(n^2|C|\right)$ per-party complexity of Lindell et al. [LPSY15]. Ben-Efraim et al. [BCO+21] then further optimized the output evaluation phase to require only $O(|C|)$ per-party local computation (after reconstruction) using an LPN based encryption scheme. Additionally, their protocol features a online garbling phase with total communication complexity $O(n|C|)$, but their circuit independent preprocessing phase still has total communication complexity $O\left(n^2|C|\right)$. They achieve this result by reducing the size of the garbled tables to be constant in the number of parties. Their scheme uses an LPN-based encryption scheme that is both key-homomorphic and message-homomorphic, further demonstrating linearly homomorphic cryptographic primitives can produce concretely efficient protocols [BDOZ11, DPSZ12, KPR18, OSV20, CHI+20, CCD+20].

Finally we note that it is possible to garble arithmetic functionalities [AIK11, BMR16], at a high cost. Ben-Ephraim et al. [Ben18] and Makri et al. [MW19] study the feasibility of computing such function within a MPC protocol.

1.3 Future Directions

Our current protocol relies on the security of LPN. It would be interesting to see if a similar result—multiparty garbling with $O(|\mathrm{C}|)$ communication complexity—can be achieved with weaker assumptions, namely just one-way functions or information theoretic techniques for circuits in NC^1.

2 Technical Overview

We now give an overview of our techniques for scalable multiparty garbling.

2.1 Background and Barriers to Scalable Multiparty Garbling

BMR Constant Round Protocol Template. In their seminal work, Beaver, Micali, and Rogaway (BMR) [BMR90] outline a template for constructing constant round MPC. The parties first perform a *garbling phase* by taking a generic (*i.e.* non-constant round) MPC protocol and using it to compute a garbled circuit of the functionality—rather than computing the function itself. The parties then initiate an *output evaluation phase*, in which they locally evaluate the garbled circuit to recover the function output. Because the garbling procedure is not inherently sequential, the tables can all be computed in parallel. Since computing each garbled table can be done in a constant number of rounds (by using an appropriate encryption scheme), the resulting protocol is itself constant round. Since the introduction of the BMR template, a long sequence of works (e.g., [DI05, LPSY15, BLO17, NST17, HSS17, WRK17b, HOSS18b, HOSS18a, BCO+21]) have investigated the efficiency of running protocols within the BMR template, leading to several improvements.

Black-Box Use of Cryptography. Beaver, Micali, and Rogaway's initial protocol proposed making non-black box use of the garbling algorithm of the garbled circuit scheme. For a gate g with input wires a, b and output wire c computing the function $f : \{0, 1\}^2 \to \{0, 1\}$, the row of the garbled table corresponding to inputs $\alpha, \beta \in \{0, 1\}$ is computed as

$$ct_{\alpha,\beta} = \mathsf{PRF}_{\mathsf{k}_{a,\alpha}}(g) \oplus \mathsf{PRF}_{\mathsf{k}_{b,\beta}}(g) \oplus \mathsf{k}_{c,f(\alpha,\beta)}$$

where $\mathsf{k}_{a,\alpha}$ is a random key/label associated with the wire a and the value $\alpha \in \{0, 1\}$. While conceptually simple, the explicit circuit representation of PRF will be massive, resulting in an inefficient concrete construction. To obtain an efficient *black-box* solution, the PRF must somehow be evaluated *outside* the MPC protocol without compromising privacy or correctness.

Damgård and Ishai [DI05] devised an intuitive way to accomplish this goal: each party $\mathsf{P}_m \in \{\mathsf{P}_1, \ldots, \mathsf{P}_n\}$ independently samples a pair of labels $(\mathsf{k}^m_{c,0}, \mathsf{k}^m_{c,1})$ for each wire c in the circuit. The parties then combine these independent labels into a single label containing n keys within the MPC protocol as

$$(\mathsf{k}^1_{c,0}\| \ldots \|\mathsf{k}^n_{c,0}, \mathsf{k}^1_{c,1}\| \ldots \|\mathsf{k}^n_{c,1}).$$

The parties then encrypt the combined labels by feeding locally expanded PRFs into the MPC. Thus the garbled table for gate g with input wires a, b and output wire c implementing the function f is computed as follows for inputs $\alpha, \beta \in \{0, 1\}$ and party index $j \in [n]$: [5]

$$ct^j_{\alpha,\beta} = \bigoplus_{m=1}^{n} \mathsf{PRF}_{\mathsf{k}^m_{a,\alpha}}(g\|j) \oplus \bigoplus_{m=1}^{n} \mathsf{PRF}_{\mathsf{k}^m_{b,\beta}}(g\|j) \oplus \mathsf{k}^j_{c,f(\alpha,\beta)}$$

In other words, the garbled table for each gate consists of $4n$ ciphertexts, where each ciphertext is computed using n keys. Subsequent to the initial presentation of this protocol [DI05], Lindell et al. [LPSY15] showed that this approach can be extended to the malicious security case by incorporating simple local checks performed by the parties during the output evaluation phase.

The Main Bottleneck. Unfortunately, the above black-box approach comes at a price. Since *each party contributes to encrypting every other party's key*, the circuit representation of garbling each gate is of size $O(n^2)$.[6] Thus, when running the garbling phase, the overall communication complexity will be at least

[5] In the technical overview, we don't explicitly discuss how rows in the garbled table are permuted. We do this using standard point-and-permute [BMR90] techniques by sampling random bit-masks for every wire in the circuit.

[6] Throughout the technical overview, we will generally omit the security parameter from our asymptotic notation, as our focus is the dependence on the number of parties.

quadratic in n (which clearly doesn't scale well as n grows)—no matter the efficiency of the MPC protocol used. Thus, reducing the size of the garbled tables is a necessary condition for scalable multiparty garbling.

Roadmap. We use the following roadmap to achieve our result:

- *Generic Honest Majority Template.* We first identify a template for scalable multiparty garbling in the honest majority setting. We rely on a notion that we call *doubly-linear homomorphic* pseudorandom functions (DLH-PRFs), which require that any linear operation on the PRF output corresponds to the same operation on the keys. This is a variant of linearly key-homomorphic PRFs proposed by Boneh et al. [BLMR13] and used in the recent works by Ben-Efraim et al. [BLO17, BCO+21] in order to optimize the efficiency of the output evaluation phase. Our main observation is that the use of DLH-PRFs can be successfully "married" with packed secret-sharing (and other) techniques including ones from a recent work by Goyal et al. [GPS22] to compute garbled circuits with $O(|\mathsf{C}|)$ total communication.

- *Instantiation.* Since there are no known constructions of DLH-PRFs, we instantiate our template with a random function based on the learning parity with noise (LPN) assumption over large fields. This leads to unique challenges in the honest majority setting. In particular, we custom design efficient sub-protocols that enable distributed generation of the cryptographic material used in this random function. Finally, we show how to augment the above approach using known techniques [DPSZ12, CGH+18, BGJK21, GPS22] to achieve malicious security.

2.2 An Honest Majority Template for Scalable Multiparty Garbling

Reducing the Number of Ciphertexts. Motivated by the desire to optimize the output evaluation phase of multiparty garbling, Ben-Efraim et al. [BLO17] demonstrate a method in the dishonest majority setting for computing garbled tables with a *constant* number of ciphertexts without relying on non-black box use of cryptography. At the heart of their approach is a PRF that has key homomorphism [BLMR13]. If each party has a share of the wire key, encryption can be computed by (1) parties locally evaluating the PRF on their shares of the key, and (2) homomorphically combining the PRF outputs.

We build upon this approach to achieve better efficiency guarantees in the honest majority setting. When switching to the honest majority setting, we leverage *threshold* secret sharing instead of additive secret sharing. This fundamentally changes what it means for the parties to hold shares of the key, and therefore what it means to "combine" the output of the PRF. Following the general approach of Ben-Efraim et al., we want parties to be able to locally compute *shares* of a single ciphertext, such that the ciphertext can be reconstructed during the output evaluation phase. Specifically, let us imagine that for each gate g computing the function f, the parties hold secret shares of the keys corresponding to the input wires a, b and output wire c. We want each party P_m to locally compute its share of the row $\alpha, \beta \in \{0, 1\}$ as

$$[ct_{\alpha,\beta}]_m = \mathsf{PRF}_{[\mathsf{k}_{a,\alpha}]_m}(g) \oplus \mathsf{PRF}_{[\mathsf{k}_{b,\beta}]_m}(g) \oplus [\mathsf{k}_{c,f(\alpha,\beta)}]_m.$$

Recall that for this to be a valid secret sharing, the shares must reconstruct to a unique value when run through the Recon function of the secret sharing scheme. This function computes a *linear* function of the shares, where the coefficients are Lagrange coefficients. Because the reconstruction algorithm is linear (and therefore distributes over the linear operation $\oplus$), it follows that this will only be a valid secret sharing if $\mathsf{PRF}_{[\mathsf{k}_{a,\alpha}]_m}(g)$, $\mathsf{PRF}_{[\mathsf{k}_{b,\beta}]_m}(g)$ and $[\mathsf{k}^m_{c,g(\alpha,\beta)}]_m$ are each valid secret sharings of the values $\mathsf{PRF}_{\mathsf{k}_{a,\alpha}}(g)$, $\mathsf{PRF}_{\mathsf{k}_{b,\beta}}(g)$ and $\mathsf{k}_{b,\beta}$ respectively. That is,

$$\mathsf{PRF}_{[\mathsf{k}_{a,\alpha}]_m}(g) = [\mathsf{PRF}_{\mathsf{k}_{a,\alpha}}(g)]_m \quad \text{and} \quad \mathsf{PRF}_{[\mathsf{k}_{b,\beta}]_m}(g) = [\mathsf{PRF}_{\mathsf{k}_{b,\beta}}(g)]_m$$

This observation leads us directly to the property we require of our PRF, which we call *doubly-linear homomorphic* PRF. A PRF is *doubly-linear homomorphic* if for all linear functions L and keys k, the output of the PRF on k can also be obtained by applying L to the PRF output of any linear combination that produces k (*i.*e $L(\mathsf{PRF}_{k_1}(x), \mathsf{PRF}_{k_2}(x)) = \mathsf{PRF}_{L(k_1,k_2)}(x)$). We call it *doubly-linear* because we apply the same linear function (namely Recon) on both the keys and the PRF outputs. We note that this is a subtly different property than that required in the dishonest majority setting.

Achieving Malicious Security. As observed in prior work, achieving malicious security in multiparty garbling involves protecting against two types of attacks: (1) malicious adversaries manipulating the MPC protocol used to compute the garbled circuit, and (2) malicious adversaries injecting errors into the garbled circuit by using *inconsistent inputs*.

Towards addressing the first type of attack, we note that a long history of active research on honest majority malicious security compilers [GIP+14, GIP15, LN17, CGH+18, DOS18, NV18, FL19, GSZ20] has significantly reduced the overhead of malicious security. Adapting these techniques to our setting requires some care, which we discuss in more detail in Section 2.7.

Let us now discuss defense against the second type of attacks. We note that in the honest majority regime, when using a doubly-linear homomorphic PRF, it is not actually possible for the malicious players to manipulate the value of the PRF output directly, as the PRF output within a threshold secret sharing is uniquely defined by the honest parties' shares. Indeed, this is much more of an immediate problem in the *dishonest majority* setting, where additive shares are more commonly used. However, a subtle new potential attack avenue that rises from our instantiation of doubly-linear homomorphic PRFs with a random function. Fortunately, we observe that our approach for handling the first type of attacks can also be used to counter this attack. We defer more details on this to later in this section.

MPC with $O(|\mathsf{C}|)$ Communication. The final step in our template involves using an MPC protocol for computing the garbled tables with communication overhead independent of the number of parties. Such MPC protocols with $O(|\mathsf{C}|)$ total communication have been the subject of significant research efforts [DIK+08, DIK10, GIP15, GSY21, BGJK21, GPS21, GPS22]. All of these protocols rely heavily on *threshold packed secret sharing schemes* [FY92], a "Single-Instruction-Multiple-Data" (SIMD) version of threshold secret sharing schemes [Sha79]. By operating on $O(n)$ elements at a time and using efficient multiplication protocols (e.g. [DN07]), these protocols are able to achieve total communication complexity independent of the number of parties. In particular, we rely on the techniques from a recent work by Goyal et al. [GPS22], which is an efficient, non-constant round MPC for general circuits. We defer an overview of this protocol to later in this section. Next, we discuss the additional sub-protocols that we design in order to instantiate our multiparty garbling template with $O(|\mathsf{C}|)$ total communication and how we instantiate DLH-PRF.

2.3 Secret Sharing Bits/Masks

There is one significant element of the BMR template that we have so far not addressed: the rows of each garbled table must be permuted so that its position in the table reveals nothing about its value. In practice, most prior works do this by sampling a random "mask" bit $\lambda_c \in \{0, 1\}$ associated with each wire c. These bits are then used to select the order of the permutation of the table within the MPC. As such, this requires us to generate shares of random bits in a larger field[7].

We choose to work in a Galois Field of characteristic 2.[8] Efficient honest majority techniques are known for generating secret shares of an unknown random value in the field. However, these techniques don't

[7] Recall that protocol based on a polynomial-based secret sharing scheme requires that the field have at least $n + 1$ points, where n is the number of parties. Moreover, the [GPS22] protocol requires working in a field of size $O(|\mathsf{C}|)$.

[8] There is another reason for choosing to work in this field, which will become clear when we discussion our particular instantiation of DLH-PRF.

extend naturally for generating shares of uniform random binary values embedded in a bigger field $\mathbb{F}$. Techniques used in the dishonest majority setting [LPSY15, DPSZ12] for sampling shares of random bits are also not helpful, since they require $O(n)$ communication (for sharing each bit) in the honest majority setting. As such, we develop a replacement protocol that allows us to generate secret sharings of bits in the field $\mathbb{F}$, with a similar efficiency.

We start by recalling the standard technique [DN07, BTH08] used for generating shares of random values in the field in *batches*, using a Vandermonde matrix of size $n \times (n-t)$. Specifically, each party secret shares a random value in the field, and then each party locally multiplies the shares that it receives from other parties with the Vandermonde matrix. Since every square sub-matrix of size $(n-t) \times (n-t)$ of a Vandermonde matrix is invertible and honest parties are expected to secret share truly random values, the result is that the parties obtain $O(n)$ secret shares of random, independent values. Overall, with $O(n^2)$ communication and computation, using the above approach, parties are able to generate $O(n)$ random sharings.

To generate shares of random bits, it is not sufficient to require the parties to simply start by secret sharing random bits instead of random values in $\mathbb{F}$. If the Vandermonde matrix contains elements in $\mathbb{F}$ (as is the case in prior works), even if the parties start with shares of bits, the shares obtained after multiplying input shares with this matrix will be of elements in $\mathbb{F}$ rather than that of bits. To address this issue, we observe that the generator matrix of any binary linear error correcting code (denoted by **binM**) is a *super-invertible matrix over* $\mathbb{F}_2$. The parties can now start by simply secret sharing random bits and when they multiply these shares with **binM**, the resulting shares will be of independent, random bits. This allows us to generate $O(n)$ random bit sharings with $O(n^2)$ communication and computation.

Finally, we note that the same observation can also be used to generate *packed secret shares* of random bit-vectors. Each party simply sends packed secret sharing of vectors of random bits to the other parties. Each party then multiplies the received shares with the super-invertible bit matrix **binM**. This results in $O(n)$ packed secret sharings (containing n elements in each vector) with $O(n^2)$ communication and computation. A careful reader may have observed that this protocol yields shares of bits only if the parties originally start with sharings of bits (which cannot be guaranteed in the presence of malicious adversaries). As such, this protocol is only secure against a semi-honest adversary. We discuss malicious security for this protocol in Section 2.7.

2.4 Instantiating Key Homomorphic PRFs

The generic template above assumes the existence of a doubly-linear homomorphic pseudorandom function. To our knowledge, PRFs with this property are currently not known. In fact, even linearly key homomorphic PRFs are presently only known based on the Decisional Diffie-Hellman assumption in the random oracle model [BLMR13]. Ben Efraim et al. [BLO17, BCO+21] devised a way around the lack of key-homomorphic PRFs by using ring LWE and LPN (over boolean field) based random functions instead of a pseudo-random function. The function that they use is of the form $\mathbf{k} \cdot \mathbf{A} + \mathbf{e}$, where $\mathbf{k}$ is the key vector, $\mathbf{A}$ is a public matrix and $\mathbf{e}$ is the random error vector. We note that a similar function can also be based on field LWE and LPN over large fields.

Recall that while not a PRF, encryption using a random function of the above form still to some extent captures the doubly linear homomorphic property that we wanted, albeit with a caveat. Since, $\mathbf{A}$ is public, computing the random function $(\mathbf{k} \cdot \mathbf{A}) + \mathbf{e}$ requires linear operations over both the key vector $\mathbf{k}$ and the error vector $\mathbf{e}$. This essentially implies that if the parties hold shares of $\mathbf{k}$, $\mathbf{e}$ and the message $\mathbf{m}$, they can locally compute shares of the corresponding ciphertext without interaction. We note that, depending on the maximum fan-out $\mathsf{fanout}_{\mathsf{max}}$ across all gates in a given circuit, the number of unique $\mathbf{A}$ matrices that we require in general is $8 \times \mathsf{fanout}_{\mathsf{max}}$ [BLO17]. Since these are public matrices, we can generate them *a priori*.

Compatibility between the random function that we use and known efficient techniques for honest majority MPC will dictate the overall efficiency and scalability of our multiparty garbling scheme. In instantiating this template, we will use use the straight-forward equivalent of this random function based on LPN over a large field (similar assumptions have been used in several recent works [BCGI18, DGN+17, JLS21]). To justify these choices, we briefly discuss the alternative assumptions that we could use—LWE and boolean LPN—and demonstrate why LPN over large fields is the most appropriate choice for our application.

LWE vs LPN. Several prior works prefer LPN over LWE for efficiency reasons. Unlike LPN, LWE is known to imply FHE and is believed to be a "more-powerful" assumption than LPN. As a result, in general, parameter sizes in LWE tend to be larger than the ones required in LPN. Moreover, the matrix $\mathbf{A}$ in LPN is the generator matrix corresponding to a probabilistic code generation algorithm. It is possible to choose matrices, where each column contains a small (constant) number of random non-zero coordinates, without weakening the security of LPN [Ale03, ADI+17]. Using such a matrix, computing $\mathbf{k} \cdot \mathbf{A}$ for any vector $\mathbf{k}$ can be done in time linear in the length of $\mathbf{k}$. On the other hand, to the best of our knowledge, no such optimizations are known in LWE and hence computing $\mathbf{k} \cdot \mathbf{A}$ requires time quadratic in the length of $\mathbf{k}$. As such, we can believe that LPN poses a more fruitful direction for instantiating our template.

LPN over a boolean field vs. LPN over a larger field. The LPN-based encryption of a message $\mathbf{m}$ requires encoding $\mathbf{m}$ using a linear error correcting code $\mathsf{ECC.Enc}$, and adding the result to the output of the random function $\mathbf{k} \cdot \mathbf{A} + \mathbf{e}$. As such, the size of an LPN ciphertext depends on both (1) the efficiency of existing $\mathsf{ECC.Enc}$, and (2) the best known attacks on the LPN assumption. LPN over large fields outperforms LPN over boolean fields in both criteria: (1) $\mathsf{ECC.Enc}$'s in larger fields tend to have better rates than binary $\mathsf{ECC.Enc}$, and (2) in the large field setting, there exist variants of the LPN assumption (see [BCGI18, EKM17] for a detailed discussion) where the best known attack remains the same as in the boolean regime.

As a result, LPN over large fields provides equivalent levels of security with smaller parameter sizes. We note that this tradeoff is not always absolute: while LPN over larger fields might admit shorter vectors $\mathbf{k}$, $\mathbf{e}$ and a smaller matrix $\mathbf{A}$, representing each element requires multiple bits, which could result in the total representation that is larger than the equivalent construction from LPN over a boolean field.

We observe that in our setting it is still less efficient to use LPN over a boolean field because the parties run an MPC protocol to generate and use the cryptographic key material. Recall that since we rely on techniques from [GPS22], we need to work in a field of size $O(|\mathsf{C}|)$. Thus, the parties will have to use a larger field *irrespective of our choice of the cryptographic assumption.* As such, LPN over a boolean field becomes *wasteful* in the context of our protocol—each bit will be represented in a large field anyhow—undermining the potential advantage of working with LPN over a boolean fields.

2.5 Sub-Protocol for Generating Errors

The errors in our LPN-based encryption are sampled from a Bernoulli distribution over $\mathbb{F}$, i.e., every element of the error vector is a random non-zero element in $\mathbb{F}$ with probability $\frac{1}{p}$ and zero with probability $1 - \frac{1}{p}$, where p is derived from the parameter choices. While efficient distributed protocols for generating shares of uniform random values in the field are known due to Damgård et al. [DN07] and Beerliova-Trubiniova et al. [BTH08], to our knowledge, no such protocols are known for generating shares of values from this *biased distribution.*

To generate shares of biased bits, we use the following observation. Let us assume that p is a power of 2, i.e., of the form $p = 2^{\tau_{\mathsf{lpn}}}$. It is now easy to see that the product of τ_{lpn} random bits will be 1 with probability $1/p$ and 0 with probability $(p-1)/p$. To implement this idea, the parties can use our random bit sharing protocol to sample shares of τ_{lpn} random bits and then multiply them to get a sharing of the appropriately-biased bit. If τ_{lpn} is constant, these multiplications can be done in a constant number of rounds. Moreover, to ensure that our total communication is $O(|\mathsf{C}|)$, we generate these shares in packed secret sharing form. We choose our LPN parameters to ensure that p is a power of 2.

We note that this does not affect our other parameters because we can choose the Reed-Solomon codes properly to correct a constant fraction of errors. For LPN security, the LPN instance is more secure when the noise rate is larger, and constant noise rate was referred to as *high noise* LPN in the literature [Döt15]. Given the above above sub-protocols, we finally discuss how to integrate them with techniques introduced in [GPS22] and our multiparty garbling template.

2.6 Integrating with Goyal et al.'s share transformation protocol [GPS22]

As discussed earlier, packed secret-sharing scheme (PSS) is a polynomial-based linear secret sharing scheme that allows sharing a vector of secrets $\mathbf{v} = \{v_1, \ldots, v_\ell\}$, where $\ell \in O(n)$. Essentially, the dealer samples a random polynomial q of appropriate degree, such that for each $j \in [\ell]$, $q(\mathsf{slot}_j^{\mathsf{def}}) = v_j$ and each party P_i (for $i \in [n]$) gets a share $q(p_i)$ (where p_i is a publicly known field element that is unique to party P_i). Most existing $O(|\mathsf{C}|)$ MPC protocols use the same set of slots/points $\mathsf{slot}_1^{\mathsf{def}}, \ldots, \mathsf{slot}_\ell^{\mathsf{def}}$ in the polynomial for embedding secrets in all packed secret sharings used throughout the protocol. Using PSS, it is possible to evaluate a block of $O(n)$ gates at the same multiplicative depth in the circuit, in one shot.

Goyal et al.'s protocol [GPS22] is a non-constant round, gate-by-gate evaluation style of protocol that slightly deviates from this approach. In this protocol, a unique slot/field element $\mathsf{slot}_g^{\mathsf{C}}$ is assigned for every gate g in the circuit and the following invariant is maintained throughout the protocol: let $g_1, \ldots, g_\ell$ be a block of gates that are evaluated simultaneously using PSS and let $\mathbf{z} = \{z_{g_1}, \ldots, z_{g_\ell}\}$ be output of these gates. Upon evaluating these gates, parties obtain a packed secret sharing of $\mathbf{z}$, where each z_{g_j} is embedded at the slot associated with gate g_j. Borrowing notion from [GPS22], we use $[\mathbf{z}\,|\mathsf{pos}]$ to denote such a sharing, where $\mathsf{pos} = \{\mathsf{slot}_{g_1}^{\mathsf{C}}, \ldots, \mathsf{slot}_{g_\ell}^{\mathsf{C}}\}$.

Evaluating a Block of Gates. We now explain in more detail how a block of gates are evaluated in [GPS22] using PSS. Let d denote the degree of the PSS. Let $g_1, \ldots, g_\ell$ be a block of multiplication or addition gates that we wish to evaluate and let $\mathbf{l} = \{l_{g_1}, \ldots, l_{g_\ell}\}$ be the set of left inputs to these gates. Further, let us assume that that for each $j \in [\ell]$, l_{g_j} was the output of some gate h_j. Given the above invariant, this means that for each $j \in [\ell]$, there must exist some degree-d packed secret sharing of the form $[\mathbf{z}_j\,|\mathsf{pos}_j]_{\mathsf{d}}$, where $\mathbf{z}_j = \{\ldots, l_{g_j}, \ldots\}$ and $\mathsf{pos}_j = \{\ldots, \mathsf{slot}_{h_j}^{\mathsf{C}}, \ldots\}$. The next steps are as follows:

1. *Bringing all left inputs to the same PSS:* In order to evaluate $g_1, \ldots, g_\ell$ simultaneously, the first step is to bring all left inputs $l_{g_1}, \ldots, l_{g_\ell}$ in the same PSS. This can be done by allowing the parties to locally multiply each $[\mathbf{z}_j\,|\mathsf{pos}_j]_{\mathsf{d}}$ with a degree-$(\ell-1)$ PSS of a unit vector $\mathbf{e}_j$ (i.e., where only the j-th term is 1 and all other terms are 0) of the form $[\mathbf{e}_j\,|\mathsf{pos}^h]_{\ell-1}$, where $\mathsf{pos}^h = \{\mathsf{slot}_{h_1}^{\mathsf{C}}, \ldots, \mathsf{slot}_{h_\ell}^{\mathsf{C}}\}$. The resulting degree-$(\mathsf{d} + \ell - 1)$ sharing will be such that the value stored at position $\mathsf{slot}_{h_j}^{\mathsf{C}}$ is l_{g_j} and the values stored at other positions in pos^h are all 0. Adding all of these multiplied shares will result in shares of the form $[\mathbf{l}\,|\mathsf{pos}^h]_{\mathsf{d}+\ell-1} = \sum_{j\in[\ell]} [\mathbf{z}_j\,|\mathsf{pos}_j]_{\mathsf{d}} \cdot [\mathbf{e}_j\,|\mathsf{pos}^h]_{\ell-1}$.

2. *Transforming to a PSS at default slots:* We now want to transform the above sharing $[\mathbf{l}\,|\mathsf{pos}^h]_{\mathsf{d}+\ell-1}$ into a sharing of the form $[\mathbf{l}\,|\mathsf{pos}_{\mathsf{def}}]_{\mathsf{d}}$, where $\mathsf{pos}_{\mathsf{def}} = \{\mathsf{slot}_1^{\mathsf{def}}, \ldots \mathsf{slot}_\ell^{\mathsf{def}}\}$ are some default slots used throughout the protocol that are independent from the ones associated the gates. We will discuss how this transformation is done shortly.

All the above steps are repeated for all the right input wire values $\mathbf{r} = \{r_{g_1}, \ldots, r_{g_l}\}$ to obtain a sharing of the form $[\mathbf{r}\,|\mathsf{pos}_{\mathsf{def}}]_{\mathsf{d}}$. If $g_1, \ldots, g_\ell$ were a block of addition gates, the parties can simply add their respective shares in $[\mathbf{l}\,|\mathsf{pos}_{\mathsf{def}}]_{\mathsf{d}}$ and $[\mathbf{r}\,|\mathsf{pos}_{\mathsf{def}}]_{\mathsf{d}}$ to obtain a sharing $[\mathbf{z}\,|\mathsf{pos}_{\mathsf{def}}]_{\mathsf{d}}$, where $\mathbf{z} = \{(l_{g_1}+r_{g_1}), \ldots, (l_{g_\ell}+r_{g_\ell})\}$. If

$g_1, \ldots, g_\ell$ were a block of multiplication gates, the parties can use existing multiplication protocols [DIK10] for computing packed shares $[\mathbf{z}\,|\mathsf{pos}_{\mathsf{def}}]_{\mathsf{d}}$ of the multiplied values, i.e., $\mathbf{z} = \{(l_{g_1} \cdot r_{g_1}), \ldots, (l_{g_\ell} \cdot r_{g_\ell})\}$.

Finally, in order to comply with the invariant, the last step in their protocol is to transform $[\mathbf{z}\,|\mathsf{pos}_{\mathsf{def}}]_{\mathsf{d}}$ into $[\mathbf{z}\,|\mathsf{pos}]_{\mathsf{d}}$, where $\mathsf{pos} = \{\mathsf{slot}^{\mathsf{C}}_{g_1}, \ldots, \mathsf{slot}^{\mathsf{C}}_{g_\ell}\}$ are the positions associated with gates $g_1, \ldots, g_\ell$. Next, we discuss how this transformation is done.

Share Transformation. Notice that in the above approach, we need to switch between sharings of the form $[\mathbf{x}\,|\mathsf{pos}_1]_{d_1}$ and $[\mathbf{x}\,|\mathsf{pos}_2]_{d_2}$. This can be done easily if the parties have access to secret sharings of random vectors of form $[\mathbf{r}\,|\mathsf{pos}_1]_{n-1}$ and $[\mathbf{r}\,|\mathsf{pos}_2]_{d_2}$. Indeed, given such sharings, the parties can do the following: (1) locally compute $[\mathbf{x}+\mathbf{r}\,|\mathsf{pos}_1]_{n-1}$, (2) reconstruct $\mathbf{x}+\mathbf{r}$, (3) compute $[\mathbf{x}+\mathbf{r}\,|\mathsf{pos}_2]_{d_2}$ and finally, (4) subtract the random sharing to get $[\mathbf{x}\,|\mathsf{pos}_2]_{d_2}$.

Prior approaches for generating such correlated random sharings $[\mathbf{r}\,|\mathsf{pos}_1]$ and $[\mathbf{r}\,|\mathsf{pos}_2]$ required $O(n^2)$ communication. Goyal et al. [GPS22] propose a novel idea that enables efficient generation of such correlated randomness with $O(n)$ communication. We defer details about how this is done to the technical sections.

Key Generation and Garbling. For our multiparty garbling scheme, we also want to enable the parties to generate a secret sharing of random keys. In order to do this with $O(|\mathsf{C}|)$ total communication, we generate PSS of a random vector of keys. As discussed in Section 2.3, this can be done quite efficiently using known techniques [DN07, BTH08]. However, since shares of random values are generated in “batches” using this technique, when used for generating PSS, the secrets in the resulting packed shares are always stored at the same slots. While generating we ensure that these slots are always the default positions. This is also the case when we sample random sharings of bit masks and computing shares of the error vectors. When computing the ciphertext, we use the above share transformation protocol to move the above PSS of keys/masks/errors to another PSS where these values are all stored at the different slots associated with the gates/wires that they correspond to. We can now easily compute the garbling functionality using these values as input, and by relying on techniques from [GPS22].

2.7 Malicious Security

To ensure malicious security of our above approach, we need to thwart the following type of attacks:

- *Attack Type I:* The malicious parties can cause the ciphertexts to decrypt to an incorrect value by influencing error generation.
- *Attack Type II:* Any other potential attacks during the garbling phase (including at the time of key/mask generation), we need to ensure that the MPC protocol used for all the other computations in the garbling phase is also secure against malicious adversaries.

We first discuss how to handle the second type of attacks. Genkin et al. [GIP+14, GIP15] observed that most semi-honest, secret sharing (and packed secret sharing) based MPC protocols are also *private against malicious adversaries* until parties reconstruct the output shares. To add full-malicious security to such a protocol, the parties simply need to verify that the non-linear operations (i.e., non-scalar multiplications) in the circuit were honestly computed before reconstructing the output.

A recent line of works have showed how to incorporate these malicious security checks *efficiently* in the honest majority setting [DPSZ12, DKL+13, CGH+18, FL19]. The most popular kind of check is one where the parties sample a random sharing of a global MAC key (say kmac), which is essentially a random element in $\mathbb{F}$. Throughout the protocol, the parties perform every computation twice to maintain the following invariant: for every intermediate value z in the computation for which the parties hold a secret sharing (or packed secret sharing), they also hold a sharing of $(\mathsf{kmac} \cdot z)$. At the end of the parties compute a random linear combination of all the intermediate values and also compute a linear combination of all the MAC'ed intermediate values and essentially check whether the outcome of the second combination is kmac times that of the first combination. When working on a large field (i.e., exponential in the security parameter), it suffices to use a single MAC key. For smaller fields, the above check needs to be repeated for different MAC keys (to ensure negligible failure probability).

Goyal et al. [GPS22] demonstrate how the above check seamlessly extends to their protocol and techniques. We rely on similar observations to ensure malicious security of most of our garbling protocol. Besides error generation, the only other *custom* sub-protocol that we use is the random bit sharing protocol for generating shares of masks. While this sub-protocol is already private against malicious adversaries (which follows from the observation of Genkin et al. [GIP+14, GIP15]), to ensure security of our garbling protocol, we also need to ensure it actually outputs valid shares of bits and not any other field element. Indeed, if the adversary deviates from the protocol description, it could cause the parties to output (potentially invalid sharings) of any random field elements. The standard technique for checking if any given element b is 0 or 1, is to simply check if $b^2 \overset{?}{=} b$. We use the same idea. Upon receiving PSS of bits from the random bit sharing protocol, the parties multiply this sharing with itself (correctness of this multiplication checked using the above MAC based check) and at the end, we collective check if the above condition (i.e., $b^2 = b$) is met for all bits that were generated, in a single shot.

To counter the first type of attack (i.e., one that originates from incorrect error generation), we recall that the two main steps in our error generation sub-protocol are: (1) generating packed shares of random bits and (2) multiplying these bits. It is easy to see that security and correctness of both these steps can be ensured using the above ideas.

2.8 Protocol Summary

To summarize all of the ideas presented thus far, before proceeding to the technical sections, we present a high level overview of our multiparty garbling with reduced number of ciphertexts and for achieving malicious security. As discussed above, for malicious security at each step of the computation, the parties have a sharing of the actual intermediate values as well as a sharing of its MAC'ed counterpart. In the following summary, we collectively refer to them as "authenticated sharing" (and we assume that computing on authenticated shares yields authenticated shares of the outcome).

Circuit Independent Preprocessing:

(1) *MAC Key:* The parties compute a packed secret sharing of a vector in which each element of the vector is exactly a global IT-MAC key kmac.

(2) *Key-Shares:* The parties generate packed secret sharings of at least $2|\mathsf{C}|$ wire keys (2 for each wire in the circuit) using a randomness generation sub-protocol. We also multiply these keys with kmac to obtain authenticated packed sharing of the keys.

(3) *Mask-Shares:* The parties generate packed secret shares of bit-masks that will be used for permuting the rows of garbled tables. They authenticate these packed sharings of the masks. Moreover, as discussed above, the parties also compute authenticated sharings of the square of these masks.

(4) *LPN Errors:* The parties generate and authenticate packed secret sharings of LPN errors. They also compute authenticated sharings of the square of bits that were used to generate these errors.

Online Garbling: For each chunk of ℓ gates, parties using the above packed shares of keys and masks to compute the following using [GPS22]:

(1) Use Share transformation from [GPS22] to transform authenticated packed shares of keys, masks and errors to authenticated packed sharings at positions associated with the respective gates.

(2) *Computing Plaintexts:* Compute on the above transformed authenticated shares to permute keys associated with the outgoing wires of the gates according to the functionality of the gates and the masks, to obtain 4 messages – one for each row of the gate table using techniques from [GPS22].

(3) *Computing Ciphertext:* Encrypt each wire value using authenticated shares of keys and errors.

(4) *Malicious Security Check:* Check if all the above intermediate authenticated shares are consistent and if the outputs of random bit sharing sub-protocol were indeed bits.

Reconstructing the Gabrled and Evaluation Phase:

(1) *Input Keys:* Exchange inputs to the garbled circuit and obtain the relevant input wire keys.

(2) *Reconstructing GC:* Reconstruct the garbled tables to all parties.

(3) *Output Masks:* The parties also reconstruct the masks associated with the output wires.

(4) *Evaluation:* Using the reconstructed input wire keys, the parties locally evaluate the reconstructed garbled circuit.

3 Preliminaries

We consider the *client-server* model for multi-party computation where a set of n_c clients provide inputs to the functionality and receive outputs, while a set of n servers participate in the computation but do not have any inputs or outputs. A party can have different roles in the computation and the client-server model corresponds to the standard model if each party plays the role of a single client and a single server. We denote the set of servers by $\mathcal{P} = \{\mathsf{P}_1, \ldots, \mathsf{P}_n\}$ and the set of clients by $\mathcal{P}_c = \left\{\mathsf{P}^c_1, \ldots, \mathsf{P}^c_{n_c}\right\}$. For simplicity, we refer to the servers as parties in our protocols. The party P_1 will occasionally play a special role as the "leader" in our protocols; this assignment is arbitrary and could even change throughout the protocol. We assume that the parties have access to both point-to-point private and authenticated synchronous channels and a public synchronous broadcast channel, each of which has "unit" cost.

We consider security against a static adversary $\mathcal{A}$ that corrupts t servers and t_c clients. Throughout this work, we use packed Shamir shares [FY92] and we use ℓ to denote the packing constant, which is the number of secrets packed in a single share. We discuss packed secret sharing in more detail below. The default degree of our packed secret sharing scheme is $\mathrm{d} = t + \ell - 1$ and the total number of parties is $n = 2t + 2\ell - 1$.

We use $\mathbf{x}$ to denote a vector and $(\mathbf{x})_i$ to denote the i-th element in the vector. We use $[a, b]$ where $a \leq b$ to denote the set of integers $\{a, a+1, \ldots, b\}$. In this work, we construct an MPC protocol that evaluates functions that can be represented as a boolean circuit C with AND and XOR gates.[9] We abuse notation and use $\mathsf{C} : \{0,1\}^{W_{\mathsf{inp}}} \longrightarrow \{0,1\}^{W_{\mathsf{out}}}$ to also denote the function that the circuit C represents. The circuit C has W_{inp} input wires numbered $[1, W_{\mathsf{inp}}]$ and $\mathsf{client}(w)$ gives the index of the client providing the input value for wire w for each $w \in [1, W_{\mathsf{inp}}]$. C has G gates each with two inputs and one output with arbitrary fan-out. The gates are topologically ordered from $[1, G]$ and we use $\mathsf{left}(g)$ and $\mathsf{right}(g)$ to denote the index

[9] This only captures deterministic functions. However, a randomized function can be transformed into a deterministic one by asking every party to provide a random input and then XORing the inputs provided by all parties.

of the left and right input wires to gate g. We number the output wire of the g-th gate as $W_{\mathsf{inp}} + g$. Thus, there are a total of $W = W_{\mathsf{inp}} + G$ wires in C. Note that since the gates are topologically ordered, we have $\mathsf{left}(g) < W_{\mathsf{inp}} + g$ and $\mathsf{right}(g) < W_{\mathsf{inp}} + g$. The set of circuit output wires are denoted by $\mathcal{W}_{\mathsf{out}}$ such that $|\mathcal{W}_{\mathsf{out}}| = W_{\mathsf{out}}$. In our protocols, computation is carried out over a finite field $\mathbb{F}$ of size $|\mathbb{F}| \geq \ell + W + n$.

3.1 Security Model

We define secure multiparty computation in the real/ideal paradigm [Gol04]. Informally, a protocol is considered secure if whatever an adversary can do in the real execution of the protocol, can be done also in an ideal computation, in which an uncorrupted trusted party assists the computation. In this work, we consider security against a static adversary $\mathcal{A}$ that corrupts clients and servers at the onset of the protocol and we aim to construct protocols that achieve unanimous abort where $\mathcal{A}$ can cause honest parties to abort after learning the output of corrupt parties.

Real World. The real world execution of a protocol Π begins with an adversary $\mathcal{A}$ selecting an arbitrary subset of servers $C \subset \mathcal{P}$ of size t and clients $C_{\mathsf{c}} \subset \mathcal{P}_{\mathsf{c}}$ of size t_{c} to corrupt. The servers and clients then engage in an execution of a real n-party protocol Π. Throughout the execution of Π, the adversary $\mathcal{A}$ sends all messages on behalf of the corrupt clients and servers, and may follow an arbitrary polynomial-time strategy. In contrast, the honest clients and servers follow the instructions of Π. At the conclusion of the protocol, all honest clients and servers terminate with an output obtained in the computations. Malicious clients may output an arbitrary PPT function of the view of $\mathcal{A}$. This joint execution of Π under $(\mathcal{A}, C, C_{\mathsf{c}})$ in the real model, on input vector $\mathbf{x}$, auxiliary input z and security parameter κ_c, denoted by $\mathsf{REAL}^{C,C_{\mathsf{c}}}_{\Pi,\mathcal{A}(z)}(1^{\kappa_c}, \mathbf{x})$, is defined as the output of honest clients and servers, and the view of $\mathcal{A}(z)$ resulting from this protocol interaction.

Ideal World. In the ideal world, clients and servers interact with a trusted third party $\mathcal{F}$ that receives inputs from the clients, carries out the computation locally and hands the output back to the clients. The ideal world adversary Sim can directly interact with $\mathcal{F}$ to send inputs and receive outputs of corrupt clients, as well as ask $\mathcal{F}$ to abort.

Sending inputs to trusted party: Each $\mathsf{P}^{\mathsf{c}}_i \notin C_{\mathsf{c}}$, sends its input x_i to the trusted party $\mathcal{F}$. For each $\mathsf{P}^{\mathsf{c}}_i \in C$, the adversary Sim may send to the trusted party any arbitrary input. Let x'_i be the value actually sent as $\mathsf{P}^{\mathsf{c}}_i$'s input.

Early abort: The adversary Sim can abort the computation by sending an `abort` message to $\mathcal{F}$. In case of such an abort, $\mathcal{F}$ sends $\perp$ to all parties and halts.

Trusted party answers adversary: The trusted party computes $(y_1, \ldots, y_n) = f(x'_1, \ldots, x'_n)$ and sends y_i to each $\mathsf{P}^{\mathsf{c}}_i \in \mathcal{P}_{\mathsf{c}}$.

Late abort: The adversary Sim can abort the computation (after seeing the outputs of corrupt parties) by sending an `abort` message to $\mathcal{F}$. In case of such an abort, $\mathcal{F}$ sends $\perp$ to all honest parties and halts. Otherwise, the Sim sends a `continue` message to $\mathcal{F}$.

Trusted party answers remaining parties: The trusted party sends y_i to each honest $\mathsf{P}^{\mathsf{c}}_i \in \mathcal{P}_{\mathsf{c}}$.

Outputs: Honest parties always output the message received from the trusted party and the corrupt parties output nothing. The adversary Sim outputs an arbitrary function of the initial inputs x_i where $\mathsf{P}^{\mathsf{c}}_i \in C_{\mathsf{c}}$, the messages received by the corrupt parties from the trusted party and its auxiliary input.

The joint execution of $\mathcal{F}$ under (Sim, C, C_c), in the ideal model, on input vector $\mathbf{x}$, auxiliary input z to Sim and security parameter κ_c, denoted by $\mathsf{IDEAL}^{C,C_c}_{\mathcal{F},\mathsf{Sim}(z)}(1^{\kappa_c}, \mathbf{x})$ is defined as the output of honest clients and servers, and the view of Sim.

Having defined the real and ideal models, we say a protocol Π t-securely realizes $\mathcal{F}$ if for every PPT adversary $\mathcal{A}$, there exists a PPT simulator Sim in the ideal model such that for every $C \subset \mathcal{P}$ of size at most t and every $C_c \subseteq \mathcal{P}_c$, it holds that

$$\left\{\mathsf{REAL}^{C,C_c}_{\Pi,\mathcal{A}(z)}(1^{\kappa_c}, \mathbf{x})\right\}_{\mathbf{x},z,\kappa_c} \overset{c}{\approx} \left\{\mathsf{IDEAL}^{C,C_c}_{\mathcal{F},\mathsf{Sim}(z)}(1^{\kappa_c}, \mathbf{x})\right\}_{\mathbf{x},z,\kappa_c}$$

where $(\mathbf{x}, z) \in (\{0,1\}^*)^{n+1}$ and $\kappa_c \in \mathbb{N}$.

As discussed before, clients only participate in our protocol to provide inputs and receive outputs. The actual computation is carried out by the servers. As observed in prior works [GIP15, GPS22], an advantage of working in the client-server model is that it suffices to consider adversaries that corrupt exactly t servers. For any t-secure protocol Π, security against an adversary $\mathcal{A}'$ that corrupts $t' < t$ servers can be reduced to security against an adversary $\mathcal{A}$ that corrupts exactly t servers. Intuitively, this works by constructing $\mathcal{A}$ to corrupt $t - t'$ servers in addition to the t' servers that $\mathcal{A}'$ corrupts. $\mathcal{A}$ then runs the t' servers with the same strategy as $\mathcal{A}'$ while the remaining $t - t'$ servers follows the steps of the protocol. Since servers don't provide inputs, any protocol secure against $\mathcal{A}$ is secure against $\mathcal{A}'$.

3.2 Secret Sharing

In this work we use the packed Shamir secret sharing scheme introduced by Franklin and Yung [FY92]. This is a generalization of the Shamir secret sharing scheme [Sha79] where each share corresponds to ℓ secrets. Let $\mathbb{F}$ be a finite field and let $\{p_1, \ldots, p_n\}$, and $\mathsf{pos} = (\mathsf{slot}_1, \ldots, \mathsf{slot}_\ell)$ be $n + \ell$ distinct points in $\mathbb{F}$. A degree-d packed Shamir sharing of a secret $\mathbf{x} = (x_1, \ldots, x_\ell)$ in $\mathbb{F}^\ell$ is a vector $[\mathbf{x}]_d = (s_1, \ldots, s_n)$ in $\mathbb{F}^n$ which satisfies that there exists a polynomial $p(\cdot) \in \mathbb{F}[X]$ of degree at most d such that for each $i \in [1, n]$, $p(p_i) = s_i$ and for each $i \in [1, \ell]$, $p(\mathsf{slot}_i) = x_i$. Thus, pos is the set of values which evaluate to the secret $\mathbf{x}$ under $p(\cdot)$ and $\{p_i\}_{i=1}^n$ is the set of values which evaluate to the elements of the sharing. The i-th element s_i in $[\mathbf{x}]_d$ is held by the party P_i. Reconstructing a degree-d sharing is equivalent to reconstructing the underlying polynomial $p(\cdot)$ which requires $d + 1$ shares and can be done through Lagrange interpolation.

We adopt a similar approach to [GPS22], and use different sets of points pos for the secrets. We use $[\mathbf{x}\,|\mathsf{pos}]_d$ to denote a degree-d packed Shamir sharing with the secret $\mathbf{x}$ stored at positions pos. Specifically, in our protocol, we require $\ell + W$ distinct elements in $\mathbb{F}$ denoted by $\left\{\mathsf{slot}^{\mathsf{def}}_1, \ldots, \mathsf{slot}^{\mathsf{def}}_\ell, \mathsf{slot}^{\mathsf{C}}_1, \ldots, \mathsf{slot}^{\mathsf{C}}_W\right\}$ where we use $\mathsf{pos}_{\mathsf{def}} = (\mathsf{slot}^{\mathsf{def}}_i)_{i=1}^{\ell}$ as the default positions for storing secrets while $\mathsf{slot}^{\mathsf{C}}_w$ is used to store the secret associated with wire w in C. We define the following procedures for sharing and reconstructing secrets where $\mathsf{pos} = (\mathsf{slot}_1, \ldots, \mathsf{slot}_\ell)$ is an ordered set of positions and $\mathcal{I} \subset [1, n]$.

- $\mathsf{share}(d, \mathbf{x}; \mathsf{pos}, \mathbf{r})$: This procedure computes a degree-d packed Shamir sharing $[\mathbf{x}\,|\mathsf{pos}]_d$ of $\mathbf{x} \in \mathbb{F}^\ell$ over a random polynomial $p(\cdot)$ of degree at most d using the uniformly random vector $\mathbf{r} \in \mathbb{F}^{d+1-\ell}$ subject to the constraints $p(\mathsf{slot}_i) = (\mathbf{x})_i$ for every $i \in [1, \ell]$. For brevity, we use $\mathsf{share}(d, \mathbf{x})$ to denote $\mathsf{share}(d, \mathbf{x}; \mathsf{pos}_{\mathsf{def}}, \mathbf{r})$.

- $\mathsf{share}(d, \mathbf{x}, \{s_i\}_{i \in \mathcal{I}}; \mathsf{pos}, \mathbf{r})$: This procedure computes a degree-d packed Shamir sharing $[\mathbf{x}\,|\mathsf{pos}]_d$ of $\mathbf{x} \in \mathbb{F}^\ell$ over a random polynomial $p(\cdot)$ of degree at most d using the uniformly random vector $\mathbf{r} \in \mathbb{F}^{d+1-\ell-|\mathcal{I}|}$ subject to the constraints $p(\mathsf{slot}_i) = (\mathbf{x})_i$ for every $i \in [1, \ell]$ and $p(p_i) = s_i$ for every $i \in \mathcal{I}$. For brevity, we use $\mathsf{share}(d, \mathbf{x}, \{s_i\}_{i \in \mathcal{I}})$ to denote $\mathsf{share}(d, \mathbf{x}, \{s_i\}_{i \in \mathcal{I}}; \mathsf{pos}_{\mathsf{def}}, \mathbf{r})$. Informally, we say that a degree-d packed Shamir sharing $[\mathbf{x}\,|\mathsf{pos}]_d$ of the secret $\mathbf{x}$ is computed to be consistent with the shares $\{s_i\}_{i \in \mathcal{I}}$.

- $\mathsf{rec}(d, \mathsf{pos}, [\mathbf{x}\,|\mathsf{pos}]_d)$: This procedure reconstructs the secrets $\mathbf{x}$ stored at positions pos in the degree-d packed Shamir sharing $[\mathbf{x}\,|\mathsf{pos}]_d$. If the inputs do not form a valid degree-d packed Shamir sharing, the procedure outputs $\perp$. Informally, we say that a degree-d packed Shamir sharing $[\mathbf{x}\,|\mathsf{pos}]_d$ is reconstructed to compute the secrets $\mathbf{x}$.

As discussed earlier, our protocols are run by a set of n servers among which at most t may be corrupted by a static malicious adversary. For the majority of the computation, parties use degree-d packed Shamir sharings where $\mathrm{d} = t + \ell - 1$. Note that a degree-d sharing requires at least $t + \ell$ shares to reconstruct the secret. This implies that the shares of the t corrupt parties are independent of the secret in this case. Moreover, the shares of honest parties are completely determined by the secrets and the shares of corrupt parties since the latter define ℓ and t points on the underlying polynomial respectively. We denote a degree-d packed Shamir sharing $[\mathbf{x}\,|\mathsf{pos}]_\mathrm{d}$ as $[\mathbf{x}\,|\mathsf{pos}]$ for an arbitrary set of positions pos and as $[\mathbf{x}]$ when $\mathsf{pos} = \mathsf{pos}_{\mathsf{def}}$, for brevity.

Note that each party $\mathsf{P}_i \in \mathcal{P}$ has the share $([\mathbf{x}\,|\mathsf{pos}]_d)_i$ in the sharing $[\mathbf{x}\,|\mathsf{pos}]_d$. Co-ordinate wise operations on sharings correspond to parties locally performing the same operation on their shares i.e., the vector is simply distributed across parties. We now recall a few properties of packed Shamir sharing which follow directly from the computation on the underlying polynomials.

- Linear Homomorphism: For all $d \geq \ell - 1$, $\mathbf{x}, \mathbf{y} \in \mathbb{F}^\ell$ we have $[\mathbf{x} + \mathbf{y}\,|\mathsf{pos}]_d = [\mathbf{x}\,|\mathsf{pos}]_d + [\mathbf{y}\,|\mathsf{pos}]_d$.
- Multiplicative: Let $\mathbf{x} \cdot \mathbf{y}$ denote the co-ordinate wise multiplication operation between $\mathbf{x}$ and $\mathbf{y}$. For all $\mathbf{x}, \mathbf{y} \in \mathbb{F}^\ell$ and $d_1, d_2 \geq \ell - 1$ such that $d_1 + d_2 < n$, we have $[\mathbf{x} \cdot \mathbf{y}\,|\mathsf{pos}]_{d_1+d_2} = [\mathbf{x}\,|\mathsf{pos}]_{d_1} \cdot [\mathbf{y}\,|\mathsf{pos}]_{d_2}$.

The multiplicative property implies that degree-d packed Shamir sharings are *multiplication friendly*. This means that for all $\mathbf{x}, \mathbf{c} \in \mathbb{F}^\ell$, all parties can locally compute $[\mathbf{c} \cdot \mathbf{x}\,|\mathsf{pos}]_{\mathrm{d}+\ell-1}$ from a degree-d packed Shamir sharing $[\mathbf{x}\,|\mathsf{pos}]$ and the vector $\mathbf{c}$. To do so, parties first locally compute a degree-$(\ell - 1)$ packed Shamir sharing $[\mathbf{c}\,|\mathsf{pos}]_{\ell-1}$ and then use the multiplicative property of packed Shamir sharing to compute $[\mathbf{c} \cdot \mathbf{x}\,|\mathsf{pos}]_{\mathrm{d}+\ell-1} = [\mathbf{c}\,|\mathsf{pos}]_{\ell-1} \cdot [\mathbf{x}\,|\mathsf{pos}]$. For ease of exposition, we denote this as $[\mathbf{c} \cdot \mathbf{x}\,|\mathsf{pos}]_{\mathrm{d}+\ell-1} = \mathbf{c} \cdot [\mathbf{x}\,|\mathsf{pos}]$.

As in [GPS22], we use this multiplication friendliness to compute a degree-$(\mathrm{d} + \ell - 1)$ packed sharing where each secret might come from a different degree-d sharing with the only constraint being that the positions of the secrets are distinct. Specifically, let $\mathsf{pos}, \mathsf{pos}_1, \ldots, \mathsf{pos}_\ell$ be a set of positions such that $\mathsf{pos} = (\mathsf{slot}_1, \ldots, \mathsf{slot}_\ell)$ with the constraint that $\mathsf{slot}_i \in \mathsf{pos}_i$ for each $i \in [1, \ell]$. Consider the procedure $\mathsf{select}([\mathbf{x}_1\,|\mathsf{pos}_1], \ldots, [\mathbf{x}_\ell\,|\mathsf{pos}_\ell], \mathsf{pos})$ which computes the following

$$[\mathbf{x}\,|\mathsf{pos}]_{\mathrm{d}+\ell-1} = \sum_{i=1}^{\ell} [\mathbf{e}_i\,|\mathsf{pos}]_{\ell-1} \cdot [\mathbf{x}_i\,|\mathsf{pos}_i]$$

where $\mathbf{e}_i$ is a vector with $(\mathbf{e}_i)_i = 1$ and $(\mathbf{e}_i)_j = 0$ for all $j \in [1, \ell] \setminus \{i\}$. Thus, the secret $\mathbf{x}$ in the output sharing $[\mathbf{x}\,|\mathsf{pos}]_{\mathrm{d}+\ell-1}$ is such that $(\mathbf{x})_i$ is the secret stored at slot_i in $[\mathbf{x}_i\,|\mathsf{pos}_i]$.

We use information theoretic MACs for verifying the correctness of computation in the MPC protocol and since we work over a small field $\mathbb{F}$ (in contrast to a large field with order exponential in the security parameter) we use $L_{\mathsf{mac}} \geq \kappa_s / \log|\mathbb{F}|$ number of information theoretic MACs of the form $[\mathsf{kmac}_i \cdot \mathbf{x}\,|\mathsf{pos}]_d$ for each $i \in [1, L_{\mathsf{mac}}]$ for a given sharing $[\mathbf{x}\,|\mathsf{pos}]_d$ to ensure negligible probability of the verification going through despite malicious behavior. Here, kmac_i corresponds to the MAC key for the i-th MAC and is a uniformly random value in $\mathbb{F}$. We denote such authenticated sharings by $[\![\mathbf{x}\,|\mathsf{pos}]\!]_d$ and operations on these sharings translate to performing the same operation on $[\mathbf{x}\,|\mathsf{pos}]_d$ and $[\mathsf{kmac}_i \cdot \mathbf{x}\,|\mathsf{pos}]_d$ for every $i \in [1, L_{\mathsf{mac}}]$.

Additive Errors to Shares and Secrets. Let $\mathcal{A}$ denote the adversary, C denote the set of corrupt parties, and $\mathcal{H}$ be the set of honest parties. As discussed earlier, most of our sub-protocols are semi-honest secure but guarantee perfect privacy against a malicious adversary. However, $\mathcal{A}$ can carry out attacks that affect the correctness of the sub-protocols. As discussed in [GPS21], the deviation of a fully malicious adversary can be classified into the following two kinds of attacks.

- The adversary can distribute an inconsistent degree-d packed Shamir sharing.
- The adversary can add additive errors to the secrets of the output sharing.

Specifically, we have $n - t = t + 2\ell - 1$ honest parties and the degree of the packed Sharing is $\mathrm{d} = t + \ell - 1$. Since only $\mathrm{d} + 1$ shares are required to determine the complete sharing, $\mathcal{A}$ can carry out an attack such that the shares of all honest parties no longer lie on a polynomial of degree d. Note that this is not an issue when $n = 2t + 1$ and computation is done using degree-t sharings since the shares of *all* honest parties define the complete sharing. In such a case, honest parties having incorrect shares would would only lead to an additive error on the secret.

We use the same approach as [GPS21] and capture the attacks carried out by a malicious adversary by accounting for additive errors to the secret and sharings. Let $\mathcal{H}_{\mathcal{H}}$ be any arbitrary fixed subset of $\mathcal{H}$ of size $\mathrm{d} + 1$, and $\mathcal{H}_C = \mathcal{H} \setminus \mathcal{H}_{\mathcal{H}}$. Since $|\mathcal{H}_{\mathcal{H}}| = \mathrm{d} + 1$, the shares in $[\mathbf{x}\,|\mathrm{pos}]$ corresponding to parties in $\mathcal{H}_{\mathcal{H}}$ completely determine a degree-d sharing, which we denote by $[\mathbf{x}\,|\mathrm{pos}]^{\mathcal{H}}$. Observe that if $\mathcal{A}$ introduces errors to the shares of parties in $\mathcal{H}_{\mathcal{H}}$, it translates to an additive error on the secret, which we capture in our functionalities through the vector $\boldsymbol{\delta}_x \in \mathbb{F}^{\ell}$ for a secret $\mathbf{x}$. Note that $[\mathbf{x}\,|\mathrm{pos}]^{\mathcal{H}}$ is by definition a valid degree-d packed Shamir sharing while $[\mathbf{x}\,|\mathrm{pos}]$ can be inconsistent. We will maintain the invariant that $\mathcal{A}$ can learn the shares of corrupt parties in $[\mathbf{x}\,|\mathrm{pos}]^{\mathcal{H}}$ and the difference $\boldsymbol{\Delta}_x = [\mathbf{x}\,|\mathrm{pos}] - [\mathbf{x}\,|\mathrm{pos}]^{\mathcal{H}}$. Since the shares held by honest parties in $\mathcal{H}_{\mathcal{H}}$ for $[\mathbf{x}\,|\mathrm{pos}]$ are equal to those in $[\mathbf{x}\,|\mathrm{pos}]^{\mathcal{H}}$ by definition, it follows that the elements in $\boldsymbol{\Delta}_x$ corresponding to parties in $\mathcal{H}_{\mathcal{H}}$ are 0. Moreover, since we maintain the invariant that $\mathcal{A}$ learns the shares of corrupt parties in $[\mathbf{x}\,|\mathrm{pos}]^{\mathcal{H}}$, we assume that corrupt parties hold the correct shares in $[\mathbf{x}\,|\mathrm{pos}]$ and only communicate incorrect values during the computation. Thus, $\boldsymbol{\Delta}_x$ possibly has non-zero elements corresponding to parties in $\mathcal{H}_C$. Note that the complete sharings $[\mathbf{x}]$ and $[\mathbf{x}]^{\mathcal{H}}$ can be computed only from the shares of honest parties.

3.3 Error Correcting Codes

Let Q, L, d, q be integers. An $[Q, L, d]_q$ error correcting code is a pair of algorithms $\mathbf{ECC} = (\mathsf{Enc}, \mathsf{Dec})$, where the encoding algorithm Enc takes a message $\mathbf{m} \in [1, q]^L$ as input, and outputs a codeword in $[1, q]^Q$. The decoding algorithm Dec takes a potentially corrupted codeword as input, and recovers the message. The distance of the code is the minimum Hamming distance between any two different codewords. For an error correcting code with distance d, it can correct at most $\lfloor (d-1)/2 \rfloor$ Hamming errors. We now discuss two properties of error correcting codes required in the construction of our protocol. Specifically, Theorem 1 shows that the generator matrix of a binary linear error correcting code is a binary super-invertible matrix [DN07], which is in turn used in the construction of our Π_{bitrand} protocol (Section 5.4).

Theorem 1. Let $C = \{\mathbf{c} \mid \mathbf{c} = \mathbf{G} \cdot \mathbf{m}\} \subseteq \mathbb{F}^Q$ be an $[Q, L, d]$-binary linear code with generating matrix $\mathbf{G} \in \mathbb{F}^{Q \times L}$, then any sub-matrices consisted of $(Q - d + 1)$-rows of $\mathbf{G}$ is full rank.

Proof. We prove by contradiction. Suppose the theorem is not true, then there exists a sub-matrix $\mathbf{G}'$ of $(Q-d+1)$-rows of $\mathbf{G}$, and $\mathbf{G}'$ is not full rank. In that case, there exists an $\mathbf{m} \neq 0$ such that $\mathbf{G}' \cdot \mathbf{m} = 0$. Since $\mathbf{G}'$ is a sub-matrix of $\mathbf{G}$, the codeword of $\mathbf{m}$ has at least $Q - d + 1$ zero entries. Hence, the distance of the code is less than $d - 1$, and we reach a contradiction. □

Theorem 2. Let C be a code with parameters $[Q, L, d]_q$ and C' be another code with parameters $[Q', L', d']_{q'}$, where $q = 2^{L'}$, then the concatenated code $C \circ C'$ has parameters $[Q \cdot Q', L \cdot L']_{q'}$ with distance at least $d \cdot d'$.

3.4 LPN Assumption and LPN Based Encryption

Our protocols rely on the Learning Parity with Noise (LPN) assumption over large fields that has also been used in a number of prior works [IPS09, AAB15, ADI+17, BCGI18, JLS21]. Here, we recall a variation of the assumption stated by Boyle et al. [BCGI18]. Let $\mathsf{Ber}_{\tau_{\mathsf{lpn}}}$ denote the Bernoulli distribution over the field $\mathbb{F}$ obtained by sampling a uniformly random element of $\mathbb{F}$ with probability $2^{-\tau_{\mathsf{lpn}}}$, and 0 with probability $1 - 2^{-\tau_{\mathsf{lpn}}}$. Then, the LPN assumption over large fields assumes that it is hard to distinguish the tuple $(\mathbf{A}, \mathbf{A} \cdot \mathbf{s} + \mathbf{e})$ and a random tuple, where $\mathbf{A}, \mathbf{s}$ are sampled from the uniform distribution and $\mathbf{e}$ is a noise vector with small Hamming weight sampled from $\mathsf{Ber}_{\tau_{\mathsf{lpn}}}$.

Definition 1 (LPN Assumption)**.** Let $\mathbf{C}$ be a probabilistic code generation algorithm such that $\mathbf{C}(L_{\mathsf{lpn}}, Q_{\mathsf{lpn}}, \mathbb{F})$ outputs $A \in \mathbb{F}^{L_{\mathsf{lpn}} \times Q_{\mathsf{lpn}}}$. For dimension $L_{\mathsf{lpn}} = L_{\mathsf{lpn}}(\kappa_c)$, number of queries (or block length) $Q_{\mathsf{lpn}} = Q_{\mathsf{lpn}}(\kappa_c)$, and constant noise rate, the $\mathsf{LPN}(L_{\mathsf{lpn}}, Q_{\mathsf{lpn}}, \tau_{\mathsf{lpn}})$ assumption with respect to $\mathbf{C}$ states that for any polynomial-time non-uniform adversary $\mathcal{A}$, it holds that

$$\Pr\Big[\mathbb{F} \leftarrow \mathcal{A}(1^{\kappa_c}), \mathbf{A} \leftarrow \mathbf{C}(L_{\mathsf{lpn}}, Q_{\mathsf{lpn}}, \mathbb{F}), \mathbf{e} \leftarrow \mathsf{Ber}_{\tau_{\mathsf{lpn}}}^{Q_{\mathsf{lpn}}}, \mathbf{s} \leftarrow \mathbb{F}^{L_{\mathsf{lpn}}}, \mathbf{b} \leftarrow \mathbf{s} \cdot \mathbf{A} + \mathbf{e} : \mathcal{A}(\mathbf{A}, \mathbf{b}) = 1\Big]$$

$$\overset{c}{\approx}$$

$$\Pr\Big[\mathbb{F} \leftarrow \mathcal{A}(1^{\kappa_c}), \mathbf{A} \leftarrow \mathbf{C}(L_{\mathsf{lpn}}, Q_{\mathsf{lpn}}, \mathbb{F}), \mathbf{b} \leftarrow \mathbb{F}^{Q_{\mathsf{lpn}}} : \mathcal{A}(\mathbf{A}, \mathbf{b}) = 1\Big].$$

LPN-based Encryption. We now describe a symmetric key CPA-secure encryption scheme from the LPN assumption. We use a $[Q_{\mathsf{ecc}}, L_{\mathsf{ecc}}, d]_2$ error-correcting code in our construction. Let the message space be $\mathcal{M} = \mathbb{F}^{L_{\mathsf{ecc}}}$.

- $\mathsf{LPN.Keygen}(1^{\kappa_c})$: Sample $\mathbf{s} \leftarrow \mathbb{F}^{L_{\mathsf{lpn}}}$ uniformly at random.
- $\mathsf{LPN.Enc}(\mathbf{x}, \mathbf{s})$: To encrypt a message $\mathbf{x}$ under the key $\mathbf{s}$, first sample $\mathbf{A} \leftarrow \mathbb{F}^{L_{\mathsf{lpn}} \times Q_{\mathsf{ecc}}}$ uniformly at random and sample $\boldsymbol{\epsilon} \leftarrow \mathsf{Ber}_{\tau_{\mathsf{lpn}}}^{Q_{\mathsf{ecc}}}$. Output $(\mathbf{A}, \mathbf{s} \cdot \mathbf{A} + \boldsymbol{\epsilon} + \mathsf{ECC.Enc}(\mathbf{x}))$ as the ciphertext.
- $\mathsf{LPN.Dec}((\mathbf{A}, \mathbf{c}), \mathbf{s})$: Decrypt the ciphertext $\mathbf{c}$ using $\mathbf{s}$ by computing $\mathsf{ECC.Dec}(\mathbf{c} - \mathbf{s} \cdot \mathbf{A})$.

Note that security follows from the fact that $(\mathbf{A}, \mathbf{s} \cdot \mathbf{A} + \boldsymbol{\epsilon} + \mathsf{ECC.Enc}(\mathbf{x}))$ is computationally indistinguishable from $(\mathbf{A}, \mathbf{r} + \mathsf{ECC.Enc}(\mathbf{x}))$ where $\mathbf{r} \leftarrow \mathbb{F}^{Q_{\mathsf{ecc}}}$ is sampled uniformly at random. Correctness holds because $\mathbf{c} - \mathbf{s} \cdot \mathbf{A} = \boldsymbol{\epsilon} + \mathsf{ECC.Enc}(\mathbf{x})$ and if the Hamming weight of the noise $\boldsymbol{\epsilon}$ is no greater than $\lfloor (d-1)/2 \rfloor$, then the error correcting code can decrypt the message correctly.

4 LPN Based Garbling Scheme

In this section, we present the LPN encryption based garbling scheme used in our MPC protocol. At a high level, parties run the garbling algorithm defined below in a distributed manner in the MPC protocol. The resulting garbled circuit is then evaluated locally by each party to compute the output.

We use the standard garbled circuit scheme, as used in prior works on multi-party garbling [LPSY15, BLO17, WRK17b, HSS17, BCO+21], where evaluation of the garbled circuit corresponds to a masked evaluation of the circuit. Specifically, the garbling scheme samples a random bit λ_w for every wire w. The evaluation algorithm then uses the garbled circuit to carry out the computation on masked values i.e., it maintains the invariant that only the masked value $\rho_w = \lambda_w + v_w$ is revealed to parties for each wire w, where v_w denotes the actual value on the wire w during a plaintext evaluation of the circuit. Given this evaluation invariant, the garbling algorithm constructs the garbled circuit such that knowing the masked values $\rho_{\mathsf{left}(g)}$ and $\rho_{\mathsf{right}(g)}$ on the input wires of a gate g suffices to compute the masked value $\rho_{W_{\mathsf{inp}}+g}$ on the output wire of the gate. This is done by sampling two labels k^0_w and k^1_w for each wire w corresponding to when ρ_w is 0 and 1 respectively and encrypting the label on the output wire under the labels on the input wires in a way that ensures that $k^{\rho_a}_a$ and $k^{\rho_b}_b$ can be used to obtain $k^{\rho_c}_c$ where a and b are input wires to the gate and c is the output wire. We note that the rows of the garbled table for each gate, consisting of the encryptions of the output label, are implicitly permuted due to the randomness of λ_a and λ_b.

We now proceed to describe the garbling scheme GC = $(\mathsf{MaskGen}, \mathsf{Garble}, \mathsf{Encode}, \mathsf{Eval})$.

- GC.MaskGen(C): This is a randomized algorithm that samples the mask $\lambda_w \leftarrow \{0,1\}$ uniformly at random for each wire $w \in [1, W]$ and outputs $\{\lambda_w\}_{w=1}^W$.

- GC.Garble$(\mathsf{C}, \{\lambda_w\}_{w=1}^W, 1^{\kappa_c})$: This is a randomized algorithm that computes the garbled circuit using the masks output by GC.MaskGen. It works as follows:

  1. It obtains the labels $\mathbf{k}^b_w \leftarrow \mathsf{LPN.Keygen}(1^{\kappa_c})$ by running the key generation algorithm for LPN encryption for every $w \in [1, W]$ and $b \in \{0,1\}$.

  2. For each $\alpha, \beta \in \{0,1\}$, and $g \in [1, G]$, it computes the $(2\alpha + \beta)$-th row of the garbled table for the g-th gate as
  $$\mathbf{ctx}^{\alpha,\beta}_g = \mathsf{LPN.Enc}(\mathbf{k}^b_w \| b, \mathbf{k}^\alpha_{\mathsf{left}(g)} + \mathbf{k}^\beta_{\mathsf{right}(g)})$$
  where $b = g(\lambda_{\mathsf{left}(g)} + \alpha, \lambda_{\mathsf{right}(g)} + \beta) + \lambda_{W_{\mathsf{inp}}+g}$ and $g(\cdot,\cdot)$ is the function being computed by the g-th gate.

  3. The output is the garbled circuit $\mathcal{G}$, the input encoding information $\mathsf{aux}_{\mathsf{enc}}$, and the output decoding information $\mathsf{aux}_{\mathsf{dec}}$ where
  $$\mathcal{G} = \left\{\mathbf{ctx}^{0,0}_g, \mathbf{ctx}^{0,1}_g, \mathbf{ctx}^{1,0}_g, \mathbf{ctx}^{1,1}_g\right\}_{g=1}^W$$
  $$\mathsf{aux}_{\mathsf{enc}} = \left\{\mathbf{k}^0_w, \mathbf{k}^1_w, \lambda_w\right\}_{w=1}^{W_{\mathsf{inp}}}, \quad \mathsf{aux}_{\mathsf{dec}} = \{\lambda_w\}_{w \in \mathcal{W}_{\mathsf{out}}}.$$

- GC.Encode$(\mathbf{x}, \mathsf{aux}_{\mathsf{enc}})$: This is a deterministic algorithm that takes the input and input encoding information and outputs the input encoding $\mathbf{X}$ such that the w-th element in $\mathbf{X}$ is the masked value of the input and the label corresponding to the masked value i.e., $(\mathbf{X})_w = (\mathbf{k}^{\lambda_w + x_w}_w, \lambda_w + x_w)$ where $x_w = (\mathbf{x})_w$ for each $w \in [1, W_{\mathsf{inp}}]$.

- GC.Eval$(\mathbf{X}, \mathcal{G}, \mathsf{aux}_{\mathsf{dec}})$: This is a deterministic algorithm that takes the input encoding $\mathbf{X}$, the garbled circuit $\mathcal{G}$, and the output decoding information $\mathsf{aux}_{\mathsf{dec}}$ and computes $\mathsf{C}(\mathbf{x})$. It first parses $\mathbf{X}$ such that $(\mathbf{X})_w = (\mathbf{k}^{\rho_w}_w, \rho_w)$ for each $w \in [1, W_{\mathsf{inp}}]$. It then iterates over each $g \in [1, G]$ and in each iteration decrypts a row of the garbled table by computing
$$(\mathbf{k}^{\rho_w}_w, \rho_w) = \mathsf{LPN.Dec}(\mathbf{ctx}^{\alpha,\beta}_g, \mathbf{k}^\alpha_{\mathsf{left}(g)} + \mathbf{k}^\beta_{\mathsf{right}(g)})$$
where $\alpha = \rho_{\mathsf{left}(g)}$, $\beta = \rho_{\mathsf{right}(g)}$, and $w = W_{\mathsf{inp}} + g$. It then uses $\mathsf{aux}_{\mathsf{dec}}$ to compute the output $v_w = \lambda_w + \rho_w$ for each $w \in \mathcal{W}_{\mathsf{out}}$.

We now proceed to define the properties of the garbling scheme. For ease of exposition, we split the circuit input $\mathbf{x} \in \{0,1\}^{W_{\mathsf{inp}}}$ into two parts, $\mathbf{y}$ and $\mathbf{z}$ respectively. We denote the input wires corresponding to $\mathbf{y}$ by $\mathcal{W}_y$ and those corresponding to $\mathbf{z}$ as $\mathcal{W}_z$. Note that, $\mathcal{W}_y$ and $\mathcal{W}_z$ is a partition of $[1, W_{\mathsf{inp}}]$ and that the division of the circuit inputs into $\mathbf{y}$ and $\mathbf{z}$ is arbitrary and without loss of generality. The garbling scheme satisfies the *correctness* and *security* properties defined as follows.

$\mathsf{GCSim}^{\mathsf{D}}_{\mathsf{GC},\mathsf{Sim}_{\mathsf{gc}}}(b)$

Challenger D

$b \leftarrow \{0,1\}$ $\xleftarrow{\quad \mathbf{y} \quad}$

$\{\lambda_w\}_{w=1}^{W} \leftarrow \mathsf{GC.MaskGen}(\mathsf{C}, 1^{\kappa_c})$

$\boldsymbol{\lambda}_z = \{\lambda_w\}_{w \in \mathcal{W}_z}$

$\boldsymbol{\rho}_y = \{\lambda_w + y_w\}_{w \in \mathcal{W}_y}$

$\boldsymbol{\lambda}'_z, \boldsymbol{\rho}'_y, \mathsf{state} \leftarrow \mathsf{Sim}_{\mathsf{gc}}(\mathsf{C}, \mathcal{W}_y, 1^{\kappa_c})$

$\xrightarrow[\boldsymbol{\lambda}'_z, \boldsymbol{\rho}'_y \text{ if } b = 1]{\boldsymbol{\lambda}_z, \boldsymbol{\rho}_y \text{ if } b = 0}$

$\xleftarrow{\quad \mathbf{z} \quad}$

$\mathcal{G}, \mathsf{aux}_{\mathsf{enc}}, \mathsf{aux}_{\mathsf{dec}} \leftarrow \mathsf{GC.Garble}(\mathsf{C}, \{\lambda_w\}_{w=1}^{W}, 1^{\kappa_c})$

$\mathbf{X} := \mathsf{GC.Encode}(\mathbf{y} \| \mathbf{z}, \mathsf{aux}_{\mathsf{enc}})$

$\mathcal{G}', \mathbf{X}', \mathsf{aux}'_{\mathsf{dec}} \leftarrow \mathsf{Sim}_{\mathsf{gc}}(\mathsf{C}(\mathbf{y}, \mathbf{z}), \mathbf{z}, \mathsf{state})$

$\xrightarrow[(\mathcal{G}', \mathbf{X}', \mathsf{aux}'_{\mathsf{dec}}) \text{ if } b = 1]{(\mathcal{G}, \mathbf{X}, \mathsf{aux}_{\mathsf{dec}}) \text{ if } b = 0}$

$\xleftarrow{\quad b' \quad}$

Figure 1: Security Game for the garbling scheme.

- **Correctness:** For every circuit C and inputs $\mathbf{x} \in \{0,1\}^{W_{\mathsf{inp}}}$, the garbling scheme GC satisfies

$$\Pr[\mathsf{GC.Eval}(\mathsf{GC.Encode}(\mathbf{x}, \mathsf{aux}_{\mathsf{enc}}), \mathcal{G}, \mathsf{aux}_{\mathsf{dec}}) = \mathsf{C}(\mathbf{x})] = 1$$

where the probability is over the randomness in

$$(\mathcal{G}, \mathsf{aux}_{\mathsf{enc}}, \mathsf{aux}_{\mathsf{dec}}) \leftarrow \mathsf{GC.Garble}(\mathsf{C}, \mathsf{GC.MaskGen}(\mathsf{C}), 1^{\kappa_c}).$$

Proof Sketch. Correctness of GC follows from the correctness of the standard point-and-permute based garbling. Specifically, evaluation maintains the invariant that for each wire w, the masked value $\rho_w = \lambda_w + v_w$ where v_w is the value on the wire during a plaintext evaluation of the circuit on the given inputs. Let the invariant be true on the input wires to the g-th gate and let $\alpha = \rho_{\mathsf{left}(g)}$, $\beta = \rho_{\mathsf{right}(g)}$, and $w = W_{\mathsf{inp}} + g$. During evaluation, $\mathsf{LPN.Dec}(\mathbf{ctx}_g^{\alpha,\beta}, \mathbf{k}^{\alpha}_{\mathsf{left}(g)} + \mathbf{k}^{\beta}_{\mathsf{right}(g)})$ essentially outputs

$$\begin{aligned}\rho_w &= g(\lambda_{\mathsf{left}(g)} + \alpha, \lambda_{\mathsf{right}(g)} + \beta) + \lambda_w \\ &= g\big(\rho_{\mathsf{left}(g)} + v_{\mathsf{left}(g)} + \rho_{\mathsf{left}(g)}, \rho_{\mathsf{right}(g)} + v_{\mathsf{right}(g)} + \rho_{\mathsf{right}(g)}\big) + \lambda_w \\ &= g\big(v_{\mathsf{left}(g)}, v_{\mathsf{right}(g)}\big) + \lambda_w \\ &= v_w + \lambda_w.\end{aligned}$$

The correctness of GC then follows from the invariant and the fact that the output is computed in GC.Eval as $v_w = \lambda_w + \rho_w$ for each $w \in \mathcal{W}_{\mathsf{out}}$. □

– **Security:** For every circuit C and PPT distinguisher D, there exists a PPT algorithm $\mathsf{Sim}_{\mathsf{gc}}$ such that the garbling scheme GC satisfies

$$\left|\Pr\left[\mathsf{GCSim}^{\mathsf{D}}_{\mathsf{GC},\mathsf{Sim}_{\mathsf{gc}}}(0)=1\right]-\Pr\left[\mathsf{GCSim}^{\mathsf{D}}_{\mathsf{GC},\mathsf{Sim}_{\mathsf{gc}}}(1)=1\right]\right|\leq \mathsf{negl}(\kappa_c)$$

where $\mathsf{GCSim}_{\mathsf{GC},\mathsf{Sim}_{\mathsf{gc}}}(b)$ is defined in Figure 1 and $\mathsf{GCSim}_{\mathsf{GC},\mathsf{Sim}_{\mathsf{gc}}}(b)=1$ if $b'=b$.

Proof Sketch. The proof is similar to that of standard point-and-permute garbling. When run for the first time, $\mathsf{Sim}_{\mathsf{gc}}$ outputs a uniformly random bit for every $\{\lambda_w\}_{w\in\mathcal{W}_z}$ and $\{\rho_w\}_{w\in\mathcal{W}_y}$. The state output by the simulator consists of $\boldsymbol{\lambda}'_z$, $\boldsymbol{\rho}'_y$, C, and $\mathbf{y}$. The simulator when run again with $\mathbf{z}$ and the previously output state first computes the masked value $\rho_w = v_w + \lambda_w$ for each $w\in\mathcal{W}_z$ where v_w corresponds to the input to wire w determined by $\mathbf{z}$. It also samples ρ_w uniformly at random from $\{0,1\}$ for each $w\in[W_{\mathsf{inp}},W]$. Given the masked values ρ_w for all wires w in the circuit, $\mathsf{Sim}_{\mathsf{gc}}$ can determine the *active path* i.e., the ciphertext that is decrypted in the garbled table for each gate. It then samples keys $\mathbf{k}_w \leftarrow \mathsf{LPN.Keygen}(1^{\kappa_c})$ for every wire w in the circuit and computes the garbled circuit for gate g by setting the ciphertext on the active path to an encryption of $\mathbf{k}_{W_{\mathsf{inp}}+g}$ and the remaining ciphertexts in the garbled table to an encryption of $\mathbf{0}$ under the key $\mathbf{k}_{\mathsf{left}(g)}+\mathbf{k}_{\mathsf{right}(g)}$. Finally, it computes $\mathsf{aux}_{\mathsf{dec}} = \{\lambda_w = \rho_w + v_w\}_{w\in\mathcal{W}_{\mathsf{out}}}$ where v_w is the output value determined by $C(\mathbf{y},\mathbf{z})$.

We now argue indistinguishability of the simulated view. $\lambda_w\in\{0,1\}$ is uniformly random in the garbling scheme and not received by D for $w\in\mathcal{W}_y$. Similarly, λ_w for internal wires w of the circuit is uniformly random and not part of D's view which implies that ρ_w sampled uniformly at random by the simulator is indistinguishable to the corresponding masked value computed during the evaluation of the garbled circuit produced by the garbling scheme. Moreover, it is easy to see that $\mathsf{aux}_{\mathsf{dec}}$ computed by the simulator decodes the output labels to the correct output and is indistinguishable from the decoding information output by the garbling scheme. The only difference then between the simulated view and that of the garbling scheme is that $\mathsf{Sim}_{\mathsf{gc}}$ sets the inactive ciphertexts to encryptions of $\mathbf{0}$. However, this is indistinguishable from an encryption computed by the garbling scheme and can be argued by a standard hybrid argument where indistinguishability is reduced to the security of the underlying LPN-based encryption scheme. □

We note that the above security requirement is slightly different from the standard security of garbling schemes [BHR12]. Specifically, we require the simulator to output $\boldsymbol{\lambda}_z$ and $\boldsymbol{\rho}_y$ without using the output of the circuit $\mathsf{C}(\mathbf{y},\mathbf{z})$. The simulator then takes the circuit output and computes the garbled circuit, encoded input and decoding information. The modified security guarantee is required to prove the security of our MPC protocol. However, to the best of our knowledge, existing point and permute garbling schemes are secure with respect to the above definition and prior works on multi-party garbling [LPSY15, BLO17, WRK17b, HSS17, BCO+21] require similar modifications to the security definition of a standalone garbling scheme for use in their MPC protocols. In our MPC protocol, the adversary learns the masks $\boldsymbol{\lambda}_z$ for the inputs provided by corrupt clients and the masked inputs of honest clients $\boldsymbol{\rho}_y$ before it receives the garbled circuit or the encoding of the inputs.

5 Standard Sub-protocols

In this section, we describe the functionalities and sub-protocols used in the construction of our main MPC protocol. In some of these sub- protocols, we use sets $\mathcal{S}_{\mathsf{cons}}$, $\mathcal{S}_{\mathsf{mac}}$, and $\mathcal{S}_{\mathsf{zero}}$ for bookkeeping purposes in order to collect sharings that need to later (in the main protocol) be verified for consistency, correctness of computation, and to ensure that the underlying secret is a zero vector respectively.[10] We assume $\ell = O(n)$

[10] Looking ahead, this will become clearer from context as we describe these sub-protocols and functionalities in more detail.

and discuss the asymptotic cost of securely realizing each functionality.

5.1 Sharing Random Vectors

We now describe a protocol Π_{rand} (Protocol 1) for computing a batch of random packed Shamir sharings. In the protocol, each party first deals a locally sampled random packed sharing and parties then extract uniformly random packed sharings unknown to the adversary using the super-invertible matrix (e.g., Vandermonde Matrix) $\mathbf{M}_{n-t,n}$. The functionality $\mathcal{F}_{\text{rand}}$ realized by this protocol is presented in Functionality 1. Since the corrupt parties are not guaranteed to deal consistent sharings, we model $\mathcal{F}_{\text{rand}}$ to allow Sim to specify the additive errors to the shares of honest parties. As discussed in Section 3.2, it suffices to consider additive errors to the shares of honest parties in $\mathcal{H}_C$ to model attacks which render the shares of honest parties inconsistent. Note that $\mathcal{F}_{\text{rand}}$ does not receive any additive errors to the secrets from Sim since the secrets corresponding to the output sharings are uniformly distributed. Looking ahead, we ensure that all parties did indeed deal consistent sharings through the consistency check subprotocol $\Pi_{\text{check-cons}}$ described in Section 5.10.

Complexity Analysis. The total communication complexity of Π_{rand} is $O(n^2)$ field elements. However, since each run of the protocol yields $O(n-t)$ random packed sharings, the amortized cost of generating a single packed sharing is $O(n)$ field elements. Similarly the amortized computation complexity of the protocol per packed sharing is $O(n^2)$. The round complexity of the protocol is 1.

Lemma 1. Π_{rand} (Protocol 1) t-securely realizes $n-t$ invocations of $\mathcal{F}_{\text{rand}}$ (Functionality 1).

Proof. Let $\mathcal{A}$ denote the adversary and C be the set of corrupt parties. Let $\mathcal{H}$ be the set of honest parties, $\mathcal{H}_{\mathcal{H}}$ be a fixed subset of $\mathcal{H}$ of size $\mathsf{d}+1$, and $\mathcal{H}_C = \mathcal{H} \setminus \mathcal{H}_{\mathcal{H}}$. We construct a simulator Sim to simulate the behavior of honest parties.

The simulator Sim works as follows.

- Sim sends uniformly random values as the shares of corrupt parties in $[\mathbf{u}_i]$ for every $\mathsf{P}_i \in \mathcal{H}$ and receives the shares of honest parties in $[\mathbf{u}_j]$ for every $\mathsf{P}_j \in C$ from $\mathcal{A}$.
- It then computes the whole sharing $[\mathbf{u}_j]$ and $[\mathbf{u}_j]^{\mathcal{H}}$ and computes $\Delta_{u_j} = [\mathbf{u}_j] - [\mathbf{u}_j]^{\mathcal{H}}$ for every $\mathsf{P}_j \in C$. It sets $\Delta_{u_j} = \mathbf{0}$ for each $\mathsf{P}_j \in \mathcal{H}$.
- Note that Sim now knows the shares of corrupt parties in $[\mathbf{u}_i]$ for every $\mathsf{P}_i \in \mathcal{P}$. It can thus compute the shares of the corrupt party in $[\mathbf{r}_j]$ for each $j \in [1, n-t]$ by multiplying the shares of the corrupt parties in $[\mathbf{u}_i]$ with $\mathbf{M}_{n-t,n}$. It sends the shares of the corrupt parties in $[\mathbf{r}_j]$ to the j-th invocation of $\mathcal{F}_{\text{rand}}$ for every $j \in [1, n-t]$.
- Sim computes
$$(\Delta_{r_1}, \ldots, \Delta_{r_{n-t}}) = \mathbf{M}_{n-t,n}(\Delta_{u_1}, \ldots, \Delta_{u_n})$$

Functionality 1: $\mathcal{F}_{\text{rand}}$

1. $\mathcal{F}_{\text{rand}}$ receives the shares of the corrupt parties $\{s_i\}_{i\in C}$ from Sim. It then samples $\mathbf{r} \leftarrow \mathbb{F}^{\ell}$ uniformly at random and computes $[\mathbf{r}]^{\mathcal{H}} = \mathsf{share}(\mathsf{d}, \mathbf{r}, \{s_i\}_{i\in C})$.
2. $\mathcal{F}_{\text{rand}}$ receives a vector $\Delta \in \mathbb{F}^n$ from Sim such that for all $\mathsf{P}_i \notin \mathcal{H}_C$, $(\Delta)_i = 0$. It then computes $[\mathbf{r}] = [\mathbf{r}]^{\mathcal{H}} + \Delta$.
3. $\mathcal{F}_{\text{rand}}$ distributes the shares $[\mathbf{r}]$ to honest parties.

Protocol 1: Π_{rand}

1. Every party $\mathsf{P}_i \in \mathcal{P}$ locally samples $\mathbf{u}_i \leftarrow \mathbb{F}^\ell$ uniformly at random, computes $[\mathbf{u}_i] \leftarrow \mathsf{share}(\mathsf{d}, \mathbf{u}_i)$, and then sends the j-th share $([\mathbf{u}_i])_j$ to P_j for each $j \in [1, n]$.
2. Having received their shares in $[\mathbf{u}_1], \ldots, [\mathbf{u}_n]$, parties locally compute
$$[\mathbf{r}_1], \ldots, [\mathbf{r}_{n-t}] = \mathbf{M}_{n-t,n} \cdot ([\mathbf{u}_1], \ldots, [\mathbf{u}_n])$$
and output $[\mathbf{r}_1], \ldots, [\mathbf{r}_{n-t}]$.

and sends Δ_{r_j} to the j-th invocation of $\mathcal{F}_{\mathsf{rand}}$ for every $j \in [1, n-t]$.

We now show that $\mathcal{A}$'s view in the real world is identically distributed to its view in the simulation. $\mathcal{A}$'s view in the real world consists of shares of corrupt parties in a random degree-d packed Shamir sharings dealt by honest parties. However, these shares of the corrupt parties are uniformly distributed. Since Sim sends uniformly random values to $\mathcal{A}$, it follows that the view of $\mathcal{A}$ in the simulation is identically distributed to that in the real world.

We now show that the shares output by honest parties is identically distributed in the real and ideal worlds conditioned on the view of $\mathcal{A}$.

— Let $\mathbf{M}^{\mathcal{H}}_{n-t,n-t}$ and $\mathbf{M}^{C}_{n-t,t}$ be the sub-matrix of $\mathbf{M}_{n-t,n}$ containing columns with indices in $\mathcal{H}$ and C respectively. We have

$$\begin{aligned}(\mathbf{r}_1, \ldots, \mathbf{r}_{\mathbf{n-t}}) &= \mathbf{M}_{n-t,n}(\mathbf{u}_1, \ldots, \mathbf{u}_n) \\ &= \mathbf{M}^{\mathcal{H}}_{n-t,n-t}(\mathbf{u}_i)_{\mathsf{P}_i \in \mathcal{H}} + \mathbf{M}^{C}_{n-t,t}(\mathbf{u}_i)_{\mathsf{P}_i \in C}.\end{aligned}$$

From the property of super-invertible matrices $\mathbf{M}^{\mathcal{H}}_{n-t,n-t}$ is invertible and it follows that given $\mathbf{u}_i$ shared by each corrupt party $\mathsf{P}_i \in C$, there is a one-one linear mapping between the output $\mathbf{r}_1, \ldots, \mathbf{r}_{n-t}$ and secrets shared by honest parties $\mathbf{u}_i$ where $\mathsf{P}_i \in \mathcal{H}$. Since the honest parties share uniformly random secrets, it follows that the output secrets are uniformly random too. In the ideal world, $\mathcal{F}_{\mathsf{rand}}$ samples $\mathbf{r} \leftarrow \mathbb{F}^\ell$ uniformly at random as the secret. Thus, the secrets are identically distributed in the real and ideal worlds.

— Note that $\mathcal{A}$'s view completely determines the shares of the corrupt parties in $[\mathbf{u}_i]$ for each $i \in [1, n]$ since it consists of the shares of corrupt parties in sharings dealt by honest parties and shares of honest parties in sharings dealt by corrupt parties. Specifically, the latter suffices to compute the whole sharing. Moreover, since Sim honestly follows the steps of the protocol and computes the shares of corrupt parties in the output sharings, it follows that the shares of corrupt parties are identical in the real and ideal worlds.

— In the real world, honest parties receive their shares in $[\mathbf{u}_j] = [\mathbf{u}_j]^{\mathcal{H}} + \Delta_{u_j}$ for every corrupt party $\mathsf{P}_j \in \mathcal{P}$ and they compute their output shares as

$$\begin{aligned}[\mathbf{r}_1], \ldots, [\mathbf{r}_{n-t}] &= \mathbf{M}_{n-t,n} \cdot ([\mathbf{u}_1], \ldots, [\mathbf{u}_n]) \\ &= \mathbf{M}^{\mathcal{H}}_{n-t,n-t}([\mathbf{u}_i]^{\mathcal{H}})_{\mathsf{P}_i \in \mathcal{H}} + \mathbf{M}^{C}_{n-t,t}([\mathbf{u}_i]^{\mathcal{H}} + \Delta_{u_i})_{\mathsf{P}_i \in C} \\ &= \mathbf{M}^{\mathcal{H}}_{n-t,n-t}([\mathbf{u}_i]^{\mathcal{H}})_{\mathsf{P}_i \in \mathcal{H}} + \mathbf{M}^{C}_{n-t,t}([\mathbf{u}_i]^{\mathcal{H}})_{\mathsf{P}_i \in C} + \mathbf{M}^{C}_{n-t,t}(\Delta_{u_i})_{\mathsf{P}_i \in C} \\ &= ([\mathbf{r}_1]^{\mathcal{H}}, \ldots, [\mathbf{r}_{n-t}]^{\mathcal{H}}) + \mathbf{M}^{C}_{n-t,t}(\Delta_{u_i})_{\mathsf{P}_i \in C}.\end{aligned}$$

Functionality 2: $\mathcal{F}_{\text{coin}}$

1. $\mathcal{F}_{\text{coin}}$ samples $r \leftarrow \mathbb{F}$ uniformly at random.
2. $\mathcal{F}_{\text{coin}}$ sends r to Sim.
   - If Sim sends `continue`, $\mathcal{F}_{\text{coin}}$ sends r to honest parties.
   - Else, $\mathcal{F}_{\text{coin}}$ sends `abort` to honest parties.

Protocol 2: Π_{coin}

1. Parties invoke $\mathcal{F}_{\text{rand}}$ and receive a random sharing $[\mathbf{r}]$.
2. Parties send their share in $[\mathbf{r}]$ to every other party and each party $\mathsf{P}_i \in \mathcal{P}$ checks if all shares form a valid degree-d packed Shamir sharing. If not, P_i aborts. Else, P_i reconstructs and outputs $\mathbf{r}$.

Since Sim computes the additive errors to the shares of the honest parties in the same way, it follows that the additive errors to the shares are identical in the real and ideal world.

The degree-d sharing $[\mathbf{r}]$ distributed by $\mathcal{F}_{\text{rand}}$ is completely determined by the secret $\mathbf{r}$, t shares of the corrupt parties and the additive errors to the shares of honest parties in $\mathcal{H}_C$. Since the latter are identically distributed in the real and ideal worlds, it follows that the shares of the honest parties are identically distributed in the real and ideal worlds.

Thus, the joint distribution consisting of the view of $\mathcal{A}$ and the output of honest parties is identically distributed in the real and ideal worlds which proves the security of Π_{rand}. □

5.2 Common Coin

We now describe a protocol Π_{coin} (Protocol 2) for publicly sampling random values from the field $\mathbb{F}$. In the protocol, parties invoke $\mathcal{F}_{\text{rand}}$ to receive a random packed sharing and then reconstruct it to compute a uniformly random vector. Each element in the vector then corresponds to a publicly sampled random value. Thus, an instance of the protocol generates a batch of ℓ random field elements. The functionality $\mathcal{F}_{\text{coin}}$ realized by this protocol is presented in Functionality 2. Note that the Π_{coin} realizes $\mathcal{F}_{\text{coin}}$ with abort since corrupt parties might send incorrect shares during reconstruction at which point honest parties abort.

Complexity Analysis. The total communication complexity of this protocol is $O(n^2)$ field elements. However, since each run of the protocol yields $O(\ell)$ random values, the amortized cost is $O(n)$ field elements per sample since $\ell = O(n)$. Similarly, the amortized computation complexity of the protocol for publicly sampling a single random value is $O(n)$. The round complexity of the protocol is 1 assuming the random sharing has been generated beforehand. Note that the communication complexity of realizing $\mathcal{F}_{\text{coin}}$ can be significantly improved in practice by using Π_{coin} to generate a PRF key and then using the PRF to sample pseudorandom values for parallel invocations of $\mathcal{F}_{\text{coin}}$.

Lemma 2. Π_{coin} (Protocol 2) t-securely realizes ℓ invocations of $\mathcal{F}_{\text{coin}}$ (Functionality 2) in the $\mathcal{F}_{\text{rand}}$-hybrid model.

Proof. Let $\mathcal{A}$ denote the adversary and C be the set of corrupt parties. Let $\mathcal{H}$ be the set of honest parties. We construct a simulator Sim to simulate the behavior of honest parties as follows.

Functionality 3: $\mathcal{F}_{\text{zero}}$

1. $\mathcal{F}_{\text{zero}}$ receives the shares of the corrupt parties $\{s_i\}_{i \in C}$ and computes $[o]_{n-1} = \text{share}(n-1, \mathbf{0}, \{s_i\}_{i \in C})$.
2. $\mathcal{F}_{\text{zero}}$ distributes the shares $[o]_{n-1}$ to honest parties.

Protocol 3: Π_{zero}

1. Every party $\mathsf{P}_i \in \mathcal{P}$ locally computes a zero sharing $[\mathbf{u}_i]_{n-1} \leftarrow \text{share}(n-1, (0)^{\ell})$, and then sends the j-th share $([\mathbf{u}_i])_j$ to P_j for each $j \in [1, n]$.
2. Having received their shares in $[\mathbf{u}_1], \ldots, [\mathbf{u}_n]$, parties locally compute

$$[\mathbf{o}_1], \ldots, [\mathbf{o}_{n-t}] = \mathbf{M}_{n-t,n} \cdot ([\mathbf{u}_1], \ldots, [\mathbf{u}_n])$$

and output $[\mathbf{o}_1], \ldots, [\mathbf{o}_{n-t}]$.

— Sim honestly emulates $\mathcal{F}_{\text{rand}}$ and receives the shares of the corrupt parties and the additive errors to the shares $\mathbf{\Delta}_r$ from $\mathcal{A}$. It also receives $\mathbf{r} \in \mathbb{F}^{\ell}$, where $(\mathbf{r})_i$ is received from the i-th invocation of $\mathcal{F}_{\text{coin}}$.

— It computes the sharing $[\mathbf{r}]$ such that the shares of the corrupt parties are the same as those received from $\mathcal{A}$. It then sends the shares of honest parties in $[\mathbf{r}]$ to $\mathcal{A}$ and receives the shares of the corrupt parties from $\mathcal{A}$. If $\mathcal{A}$ sent incorrect values, Sim sends `abort` to all invocations of $\mathcal{F}_{\text{coin}}$ else it sends `continue`.

It is easy to see that view of $\mathcal{A}$ in the real and ideal worlds are identically distributed since it interacts with $\mathcal{F}_{\text{rand}}$ and receives honest parties shares in a random degree-d sharing in both cases.

Note that honest parties abort in the real world when corrupt parties send incorrect shares. Sim sends `abort` to $\mathcal{F}_{\text{coin}}$ in this case in the ideal world. On the other hand, if honest parties are able reconstruct the secret they output a random vector $\mathbf{r}$. It is easy to see that the same is the case in the ideal world since the simulator constructs the sharing $[\mathbf{r}]$ based on the value received from $\mathcal{F}_{\text{coin}}$. It follows that the output of honest parties is identically distributed in the real and ideal worlds conditioned on the view of $\mathcal{A}$.

Thus, from the above argument, it follows that Π_{coin} t-securely realizes ℓ invocations of $\mathcal{F}_{\text{coin}}$. □

5.3 Sharing Zero Vectors

In this section, we describe the protocol Π_{zero} (Protocol 3) to generate degree-$(n-1)$ packed Shamir sharings of the zero vector. Shares of honest parties in a degree-$(n-1)$ packed sharing are uniformly random to the adversary despite knowing the secrets and the shares of the corrupt parties and are used to re-randomize sharings in our MPC protocol (e.g., when parties locally compute non-linear operations on their shares) in turn ensuring privacy. In the protocol, parties first compute and deal degree-$(n-1)$ packed sharings of zero and then use a super-invertible matrix $\mathbf{M}_{n-t,n}$ to compute a set of $n-t$ zero sharings which are uniformly random to the adversary. The functionality $\mathcal{F}_{\text{zero}}$ realized by this protocol is described in Functionality 3. Note that since the output of Π_{zero} is a degree-$(n-1)$ sharing, $\mathcal{F}_{\text{zero}}$ doesn't receive additive errors to either shares or secrets from Sim i.e., shares of all honest parties are trivially consistent and there always exists shares of corrupt parties such that the underlying secret is a zero vector.

Complexity Analysis. The total communication complexity of this protocol is $O(n^2)$ field elements which implies that the amortized communication complexity to generate a single packed zero sharing is

$O(n)$ field elements. Similarly, the amortized computation complexity is $O(n^2)$ per zero sharing. The round complexity of the protocol is 1.

Lemma 3. Π_{zero} (Protocol 3) t-securely realizes $n - t$ invocations of $\mathcal{F}_{\text{zero}}$ (Functionality 3).

Proof. Let $\mathcal{A}$ denote the adversary and C be the set of corrupt parties. Let $\mathcal{H}$ be the set of honest parties. We construct a simulator Sim to simulate the behavior of honest parties as follows.

- Sim samples uniformly random values as the shares of the corrupt parties in $[\mathbf{u}_i]_{n-1}$ dealt by honest parties $\mathsf{P}_i \in \mathcal{H}$. It sends these shares to $\mathcal{A}$ and receives the shares of honest parties in $[\mathbf{u}_i]_{n-1}$ dealt by corrupt parties for each $\mathsf{P}_i \in C$.
- Sim then computes the shares of corrupt parties in $[\mathbf{u}_i]_{n-1}$ for each $\mathsf{P}_i \in C$ by sampling a random degree-$(n-1)$ packed Shamir sharing of $\mathbf{0}$ such that the shares of the honest parties are consistent with those received from $\mathcal{A}$. Thus, Sim now has the corrupt parties shares in $[\mathbf{u}_i]_{n-1}$ for each $i \in [1, n]$.
- It then computes the shares of the corrupt parties in the output sharing $[\mathbf{o}_i]_{n-1}$ for each $i \in [1, n-t]$ by multiplying the shares of corrupt parties in $\{[\mathbf{u}_i]_{n-1}\}_{i=1}^{n}$ with $\mathbf{M}_{n-t,n}$. It sends the shares of corrupt parties in $[\mathbf{o}_i]_{n-1}$ to the i-th invocation of $\mathcal{F}_{\text{zero}}$ for each $i \in [1, n-t]$.

Since the shares of corrupt parties are uniformly distributed in the real and ideal worlds it follows that the view of $\mathcal{A}$ is identically distributed in both worlds.

We now show that the output of honest parties is identically distributed in the real and ideal worlds conditioned on the view of $\mathcal{A}$. Note that the $\mathcal{A}$'s view fixes the shares of honest parties in $\{[\mathbf{u}_i]_{n-1}\}_{\mathsf{P}_i \in C}$ and the shares of corrupt parties $\{[\mathbf{u}_i]_{n-1}\}_{\mathsf{P}_i \in \mathcal{H}}$. Let $\mathbf{M}^{\mathcal{H}}_{n-t,n-t}$ and $\mathbf{M}^{C}_{n-t,t}$ be the sub-matrix of $\mathbf{M}_{n-t,n}$ containing columns with indices in $\mathcal{H}$ and C respectively. For an honest party P_i, we have

$$\begin{aligned}(([\mathbf{o}_1]_{n-1})_i, \ldots, ([\mathbf{o}_{n-t}]_{n-1})_i) &= \mathbf{M}_{n-t,n}(([\mathbf{u}_1]_{n-1})_i, \ldots, ([\mathbf{u}_n]_{n-1})_i) \\ &= \mathbf{M}^{\mathcal{H}}_{n-t,n-t}(([\mathbf{u}_j]_{n-1})_i)_{\mathsf{P}_j \in \mathcal{H}} + \mathbf{M}^{C}_{n-t,t}(([\mathbf{u}_j]_{n-1})_i)_{\mathsf{P}_j \in C}\end{aligned}$$

and since from the property of super-invertible matrices $\mathbf{M}_{n-t,n-t}$ is invertible, given the shares of P_i in the sharings dealt by corrupt parties, there exists a one-to-one linear mapping between the shares of P_i in sharings dealt by honest parties and the output shares. Since, there always exists shares of honest parties for $\{[\mathbf{u}_i]_{n-1}\}_{\mathsf{P}_i \in \mathcal{H}}$ consistent with $\mathcal{A}$'s view and Sim computes the shares of the corrupt parties as in the protocol it follows that the shares of honest parties are consistent in the real and ideal worlds.

Thus, the joint distribution consisting of the view of $\mathcal{A}$ and the output of honest parties is identically distributed in the real and ideal worlds proving the security of Π_{zero}. □

5.4 Sharing Random Bit Vectors

In this section, we describe the protocol Π_{bitrand} (Protocol 4) for generating packed sharings of random bit vectors over $\mathbb{F}$. To achieve low communication complexity, each run of the protocol needs to generate a batch of packed sharings. While a super-invertible matrix [DN07] can serve as a linear randomness extractor when sampling random values in $\mathbb{F}$, we need a way to sample a random bit while working over a non-binary field with characteristic 2. To this end, we show in Theorem 1 that the generator matrix of any binary error correcting code is a super-invertible matrix over $\mathbb{F}_2$. We discuss how to generate such a super-invertible matrix $\mathbf{binM}_{k,n}$ with optimal parameters in Section 7.3. Specifically, we show that there exists $\mathbf{binM}_{k,n}$ such that $k = O(n)$. Given the existence of a binary super-invertible matrix with appropriate parameters, Π_{bitrand} works similar to Π_{rand} (Protocol 1) and extracts sharings of uniformly random bit vectors from sharings dealt by individual parties.

Functionality 4: $\mathcal{F}_{\mathsf{bitrand}}$

1. $\mathcal{F}_{\mathsf{bitrand}}$ receives the shares of the corrupt parties $\{s_i\}_{i \in C}$ and a vector $\boldsymbol{\delta} \in \mathbb{F}^{\ell}$. It then samples $\mathbf{b} \leftarrow \{0,1\}^{\ell}$ uniformly at random and computes $[b]^{\mathcal{H}} = \mathsf{share}(\mathsf{d}, \mathbf{b} + \boldsymbol{\delta}, \{s_i\}_{i \in C})$.
2. $\mathcal{F}_{\mathsf{bitrand}}$ receives a vector $\boldsymbol{\Delta} \in \mathbb{F}^n$ from Sim such that for all $\mathsf{P}_i \notin \mathcal{H}_C$, $(\boldsymbol{\Delta})_i = 0$ and computes $[b] = [b]^{\mathcal{H}} + \boldsymbol{\Delta}$.
3. $\mathcal{F}_{\mathsf{bitrand}}$ distributes the shares $[b]$ to honest parties.

Protocol 4: Π_{bitrand}

1. Every party $\mathsf{P}_i \in \mathcal{P}$ locally samples a vector of bits $\mathbf{u}_i \leftarrow \{0,1\}^{\ell}$ uniformly at random, computes $[\mathbf{u}_i] \leftarrow \mathsf{share}(\mathsf{d}, \mathbb{F}(\mathbf{u}_i))$, and then sends the j-th share $([\mathbf{u}_i])_j$ to P_j for each $j \in [1, n]$.
2. Having received their shares in $[\mathbf{u}_1], \ldots, [\mathbf{u}_n]$, parties locally compute

$$[\mathbf{b}_1], \ldots, [\mathbf{b}_k] = \mathbf{binM}_{k,n} \cdot ([\mathbf{u}_1], \ldots, [\mathbf{u}_n])$$

and output $[\mathbf{b}_1], \ldots, [\mathbf{b}_k]$.

The functionality $\mathcal{F}_{\mathsf{bitrand}}$ realized by Π_{bitrand} is presented in Functionality 4. Note that $\mathcal{F}_{\mathsf{bitrand}}$ receives the additive error to the secrets as well as additive error to the shares from Sim, which capture all attacks an adversary can run in the real world, as discussed in Section 3.2. Looking ahead, we ensure that the output sharings are consistent and that the underlying secrets are bit vectors using $\Pi_{\mathsf{check\text{-}cons}}$ and $\Pi_{\mathsf{check\text{-}bit}}$ respectively.

Complexity Analysis. The total communication complexity of this protocol is $O(n^2)$. However, when using a $k \times n$ binary super-invertible matrix where $k = O(n)$, each run of the protocol yields k packed sharings which implies that the amortized communication cost of generating a single packed sharing is $O(n^2/k)$ which is equal to $O(n)$. Similarly, the amortized computation complexity of the protocol is $O(n^2)$. The round complexity of the protocol is 1 round.

Lemma 4. Π_{bitrand} (Protocol 4) securely realizes k invocations of $\mathcal{F}_{\mathsf{bitrand}}$ (Functionality 4) against a static malicious adversary that controls t parties.

Proof. Let $\mathcal{A}$ denote the adversary and C be the set of corrupt parties. Let $\mathcal{H}$ be the set of honest parties, $\mathcal{H}_{\mathcal{H}}$ be a fixed subset of $\mathcal{H}$ of size $\mathsf{d}+1$, and $\mathcal{H}_C = \mathcal{H} \setminus \mathcal{H}_{\mathcal{H}}$. We construct a simulator Sim to simulate the behavior of honest parties.

- Sim sends uniformly random values as the shares of corrupt parties in $[\mathbf{u}_i]$ for every $\mathsf{P}_i \in \mathcal{H}$ and receives the shares of honest parties in $[\mathbf{u}_j]$ for every $\mathsf{P}_j \in C$ from $\mathcal{A}$.
- It then computes the whole sharing $[\mathbf{u}_i]$ and $[\mathbf{u}_i]^{\mathcal{H}}$ and computes $\boldsymbol{\Delta}_{u_i} = [\mathbf{u}_i] - [\mathbf{u}_i]^{\mathcal{H}}$ for every $\mathsf{P}_i \in C$. It reconstructs the secret $\mathbf{u}_i$ from $[\mathbf{u}_i]^{\mathcal{H}}$ and sets $\boldsymbol{\delta}_{u_i} = \mathbf{u}_i$ if $\mathbf{u}_i$ is not a valid bit vector else it sets $\boldsymbol{\delta}_{u_i} = \mathbf{0}$, for every $\mathsf{P}_i \in C$. It sets $\boldsymbol{\Delta}_{u_i} = \mathbf{0}$ and $\boldsymbol{\delta}_{u_i} = \mathbf{0}$ for each $\mathsf{P}_i \in \mathcal{H}$.
- Note that Sim now knows the shares of corrupt parties for $[\mathbf{u}_i]$ for every $\mathsf{P}_i \in \mathcal{P}$. It can thus compute the shares of the corrupt party in $[\mathbf{b}_j]$ for each $j \in [1, k]$ by multiplying the shares of the corrupt parties

in $[\mathbf{u}_i]$ with $\mathbf{binM}_{k,n}$. It sends the shares of the corrupt parties in $[\mathbf{b}_j]$ to the j-th invocation of $\mathcal{F}_{\mathsf{bitrand}}$ for every $j \in [1, k]$.

— Sim computes

$$(\Delta_{b_1}, \ldots, \Delta_{b_k}) = \mathbf{binM}_{k,n}(\Delta_{u_1}, \ldots, \Delta_{u_n})$$
$$(\boldsymbol{\delta}_{b_1}, \ldots, \boldsymbol{\delta}_{b_k}) = \mathbf{binM}_{k,n}(\boldsymbol{\delta}_{u_1}, \ldots, \boldsymbol{\delta}_{u_n})$$

and sends $\boldsymbol{\delta}_{b_j}$ and Δ_{b_j} to the j-th invocation of $\mathcal{F}_{\mathsf{bitrand}}$ for every $j \in [1, k]$.

Since shares of corrupt parties in the sharings dealt by honest parties are uniformly distributed in the real world, it follows that the view of $\mathcal{A}$ in the real and ideal worlds are identically distributed.

We now show that the output of honest parties is identically distributed in the real and ideal worlds conditioned on the view of $\mathcal{A}$.

— Let $\mathbf{binM}^{\mathcal{H}}_{k,k}$ be a sub-matrix of $\mathbf{binM}_{k,n}$ containing columns with indices in an arbitrary k-subset $\mathcal{I}$ of $\mathcal{H}$. Let $\mathbf{binM}'_{k,n-k}$ be the sub-matrix of $\mathbf{binM}_{k,n}$ containing the remaining columns. We have

$$\begin{aligned}(\mathbf{r_1}, \ldots, \mathbf{r_{n-t}}) &= \mathbf{binM}_{k,n}(\mathbf{u}_1, \ldots, \mathbf{u}_n)\\ &= \mathbf{binM}^{\mathcal{H}}_{k,k}(\mathbf{u}_i)_{i\in\mathcal{I}} + \mathbf{binM}'_{k,n-k}(\mathbf{u}_i)_{i\in[1,n]\setminus\mathcal{I}}.\end{aligned}$$

From the property of super-invertible matrices $\mathbf{binM}^{\mathcal{H}}_{k,k}$ is invertible and it follows that given $\mathbf{u}_i$ shared by P_i where $i \in [1, n] \setminus \mathcal{I}$, there is a one-one linear mapping (over $\mathbb{F}_2$) between the output $\mathbf{b}_1, \ldots, \mathbf{b}_k$ and secrets shared by a k-subset of honest parties $\mathbf{u}_i$ where $i \in \mathcal{I}$. Since the honest parties share uniformly random bit vectors, it follows that the output secrets in the real world are uniformly random bit vectors too assuming $(\mathbf{u_i})_{\mathsf{P}_i\in\mathcal{C}}$ are bit vectors. However, this might not be the case since corrupt parties might not deal a sharing of bit vectors. In such a case note that Sim computes exactly the additive error $\mathbf{binM}^{\mathcal{C}}_{k,t}(\mathbf{u}_i)_{\mathsf{P}_i\in\mathcal{C}}$ for the output secrets and sends it to $\mathcal{F}_{\mathsf{bitrand}}$. Since $\mathcal{F}_{\mathsf{bitrand}}$ samples a random bit vector $\mathbf{b}$ and computes the sharing of $\mathbf{b} + \boldsymbol{\delta}_b$, it follows that the secrets are identically distributed in the real and ideal worlds.

— Note that $\mathcal{A}$'s view completely decides the shares of the corrupt parties in $[\mathbf{u}_i]$ for each $i \in [1, n]$ since it consists of the shares of corrupt parties in sharings dealt by honest parties and shares of honest parties in sharings dealt by corrupt parties. Specifically, the latter suffices to compute the whole sharing. Moreover, Sim honestly follows the steps of the protocol using the shares of the corrupt parties. It thus follows that the shares of corrupt parties are identical in the real and ideal worlds.

— Let $\mathbf{binM}^{\mathcal{H}}_{k,n-t}$ and $\mathbf{binM}^{\mathcal{C}}_{k,t}$ be the sub-matrix of $\mathbf{binM}_{k,n}$ containing columns with indices in $\mathcal{H}$ and $\mathcal{C}$ respectively. In the real world, honest parties receive their shares in $[\mathbf{u}_j] = [\mathbf{u}_j]^{\mathcal{H}} + \Delta_{u_j}$ for every corrupt party $\mathsf{P}_j \in \mathcal{P}$ and they compute their output shares as

$$\begin{aligned}[\mathbf{b}_1], \ldots, [\mathbf{b}_k] &= \mathbf{binM}_{k,n} \cdot ([\mathbf{u}_1], \ldots, [\mathbf{u}_n])\\ &= \mathbf{binM}^{\mathcal{H}}_{k,n-t}([\mathbf{u}_i]^{\mathcal{H}})_{\mathsf{P}_i\in\mathcal{H}} + \mathbf{binM}^{\mathcal{C}}_{k,t}([\mathbf{u}_i]^{\mathcal{H}} + \Delta_{u_i})_{\mathsf{P}_i\in\mathcal{C}}\\ &= \mathbf{binM}^{\mathcal{H}}_{k,n-t}([\mathbf{u}_i]^{\mathcal{H}})_{\mathsf{P}_i\in\mathcal{H}} + \mathbf{binM}^{\mathcal{C}}_{k,t}([\mathbf{u}_i]^{\mathcal{H}})_{\mathsf{P}_i\in\mathcal{C}} + \mathbf{binM}^{\mathcal{C}}_{k,t}(\Delta_{u_i})_{\mathsf{P}_i\in\mathcal{C}}\\ &= ([\mathbf{b}_1]^{\mathcal{H}}, \ldots, [\mathbf{b}_k]^{\mathcal{H}}) + \mathbf{binM}^{\mathcal{C}}_{k,t}(\Delta_{u_i})_{\mathsf{P}_i\in\mathcal{C}}.\end{aligned}$$

Since Sim computes the additive errors to the shares of the honest parties in the same way, it follows that the additive errors to the shares are identical in the real and ideal world.

Functionality 5: $\mathcal{F}_{\mathsf{mult}}$

1. Let $[x]$ and $[y]$ be the input degree-d packed Shamir sharings. $\mathcal{F}_{\mathsf{mult}}$ takes the shares of honest parties in $[x]$ and $[y]$ as input.
2. $\mathcal{F}_{\mathsf{mult}}$ recovers the whole sharings $[x]$, $[x]^{\mathcal{H}}$, $[y]$, and $[y]^{\mathcal{H}}$. $\mathcal{F}_{\mathsf{mult}}$ computes $\Delta_x = [x] - [x]^{\mathcal{H}}$ and $\Delta_y = [y] - [y]^{\mathcal{H}}$. It then sends the shares of $[x]^{\mathcal{H}}$ and $[y]^{\mathcal{H}}$ for corrupt parties, and Δ_x, Δ_y to Sim.
3. $\mathcal{F}_{\mathsf{mult}}$ computes $[z]_{n-1} = [x] \cdot [y]$ and reconstructs the secrets $\mathbf{z}$.
4. $\mathcal{F}_{\mathsf{mult}}$ receives the shares of the corrupt parties $\{s_i\}_{i \in C}$ and a vector $\boldsymbol{\delta}_z \in \mathbb{F}^{\ell}$. It then computes $[z]^{\mathcal{H}} = \mathsf{share}(\mathsf{d}, \mathbf{z} + \boldsymbol{\delta}_z, \{s_i\}_{i \in C})$.
5. $\mathcal{F}_{\mathsf{mult}}$ receives a vector $\Delta_z \in \mathbb{F}^n$ from Sim such that for all $\mathsf{P}_i \notin \mathcal{H}_C$, $(\Delta_z)_i = 0$. It then computes $[z] = [z]^{\mathcal{H}} + \Delta_z$.
6. $\mathcal{F}_{\mathsf{mult}}$ distributes the shares $[z]$ to honest parties.

Protocol 5: $\Pi_{\mathsf{mult}}([\mathbf{x}], [\mathbf{y}])$

1. Parties invoke $\mathcal{F}_{\mathsf{rand}}$ and $\mathcal{F}_{\mathsf{zero}}$ to receive a random sharing $[\mathbf{r}]$ and a zero sharing $[\mathbf{o}]_{n-1}$ respectively and locally compute $[\mathbf{r}]_{n-1} = [\mathbf{r}] + [\mathbf{o}]_{n-1}$.
2. Parties locally compute $[\mathbf{u}]_{n-1} = [\mathbf{x}] \cdot [\mathbf{y}] + [\mathbf{r}]_{n-1}$ and send their shares in $[\mathbf{u}]_{n-1}$ to P_1.
3. P_1 receive the whole sharing $[\mathbf{u}]_{n-1}$, reconstructs the secrets $\mathbf{u}$ and computes $[\mathbf{u}] \longleftarrow \mathsf{share}(\mathsf{d}, \mathbf{u})$. It then sends the i-th share $([\mathbf{u}])_i$ to P_i for each $i \in [1, n]$.
4. Parties receive their share in $[\mathbf{u}]$ and output $[\mathbf{z}] = [\mathbf{u}] - [\mathbf{r}]$.

The degree-d sharing $[\mathbf{b}]$ distributed by $\mathcal{F}_{\mathsf{bitrand}}$ is completely determined by the ℓ secrets, t shares of the corrupt parties and the additive errors to the shares of honest parties in $\mathcal{H}_C$. Since the latter are identically distributed in the real and ideal worlds, it follows that the shares of the honest parties are identically distributed in the real and ideal worlds.

Thus, the joint distribution consisting of the view of $\mathcal{A}$ and the output of honest parties is identically distributed in the real and ideal worlds which proves the security of Π_{bitrand}. □

5.5 Multiplication

In this section, we first describe the protocol Π_{mult} (Protocol 5) to multiply two secret shared vectors and then describe $\Pi_{\mathsf{auth\text{-}mult}}$ (Protocol 6) to multiply two authenticated secret shared vectors (see Section 3.2).

The Π_{mult} protocol described here is identical to the one used in [DN07]. Specifically, parties locally compute a $2d = n - 1$ degree sharing of the product by multiplying their respective shares in the input packed sharings. The aim of the protocol is to then securely perform a degree reduction so that parties can output a degree-d packed sharing of the product. The degree reduction is carried out by reconstructing the degree-$(n-1)$ sharing towards P_1 after masking it with a random degree-$(n-1)$ packed sharing which ensures privacy of the secrets. P_1 then reconstructs the secret and deals a degree-d packed sharing of it. Parties then locally unmask and output a packed sharing of the product.

The functionality $\mathcal{F}_{\mathsf{mult}}$ that Π_{mult} realizes is presented in Functionality 5 and is identical to the one described in [GPS21]. Specifically, $\mathcal{F}_{\mathsf{mult}}$ reconstructs the product $\mathbf{z}$ from $[\mathbf{x}] \cdot [\mathbf{y}]$. This captures the situation

Protocol 6: $\Pi_{\text{auth-mult}}(\llbracket \mathbf{x} \rrbracket, \llbracket \mathbf{y} \rrbracket, \mathcal{S}_{\text{cons}}, \mathcal{S}_{\text{mac}})$

1. Parties invoke $\mathcal{F}_{\text{mult}}$ on inputs $[\mathbf{x}]$ and $[\mathbf{y}]$ to obtain the sharing of the product $[\mathbf{z}]$.
2. Parties invoke $\mathcal{F}_{\text{mult}}$ on inputs $[\mathsf{kmac}_i \cdot \mathbf{x}]$ and $[\mathbf{y}]$ to obtain the sharing $[\mathsf{kmac}_i \cdot z]$ for each $i \in [1, L_{\text{mac}}]$.
3. Parties set $\llbracket \mathbf{z} \rrbracket = ([\mathbf{z}], \{[\mathsf{kmac}_i \cdot \mathbf{z}]\}_{i=1}^{L_{\text{mac}}})$.
4. Parties set $\mathcal{S}_{\text{cons}} := \mathcal{S}_{\text{cons}} \cup \{[\mathbf{z}]\} \cup \{[\mathsf{kmac}_\mathsf{i} \cdot \mathbf{z}]\}_{i=1}^{L_{\text{mac}}}$ and $\mathcal{S}_{\text{mac}} := \mathcal{S}_{\text{mac}} \cup \{\llbracket \mathbf{z} \rrbracket\}$ and output $\llbracket \mathbf{z} \rrbracket$.

in Π_{mult} when the shares of honest parties are inconsistent and thus the locally computed degree-$(n-1)$ product sharing is of the form

$$[\mathbf{x}] \cdot [\mathbf{y}] + [\mathbf{r}]_{n-1} = [\mathbf{x}]^{\mathcal{H}} \cdot [\mathbf{y}]^{\mathcal{H}} + \Delta_x \cdot [\mathbf{y}]^{\mathcal{H}} + \Delta_y \cdot [\mathbf{x}]^{\mathcal{H}} + \Delta_x \cdot \Delta_y + [\mathbf{r}]_{n-1}.$$

As discussed in [GPS21], the terms involving Δ_x can introduce linear errors in $\mathbf{y}$ in the reconstructed secret and similarly, the terms involving Δ_y can introduce linear errors in $\mathbf{x}$. Thus, by reconstructing the product $\mathbf{z}$ from $[\mathbf{x}] \cdot [\mathbf{y}]$, $\mathcal{F}_{\text{mult}}$ computes the same secret as in the protocol with possible linear errors when Δ_x or Δ_y are not $\mathbf{0}$. Note that privacy is still guaranteed in this case since we use a random degree-$(n-1)$ packed sharing as the mask. However, if $\Delta_x = \Delta_y = \mathbf{0}$, then $[\mathbf{x}] = [\mathbf{x}]^{\mathcal{H}}$ and $[\mathbf{y}] = [\mathbf{y}]^{\mathcal{H}}$, and $\mathcal{F}_{\text{mult}}$ computes the correct product $\mathbf{z} = \mathbf{x} \cdot \mathbf{y}$. Finally, since P_1 might be corrupt, $\mathcal{F}_{\text{mult}}$ receives the additive errors to the secret and additive errors to the shares of the output sharing from Sim. Looking ahead, we first verify that the input and output sharings are consistent using $\Pi_{\text{check-cons}}$ (Protocol 13) which guarantees that $\Delta_x = \Delta_y = \Delta_z = \mathbf{0}$, and then verify that the additive error to the secret $\boldsymbol{\delta}_z = \mathbf{0}$ using $\Pi_{\text{check-mac}}$ (Protocol 14) which guarantees correctness of the multiplication.

The protocol $\Pi_{\text{auth-mult}}$ (Protocol 6) helps compute the multiplication of authenticated packed sharings required for running the MAC check. The sets $\mathcal{S}_{\text{cons}}$ and $\mathcal{S}_{\text{mac}}$ input to the protocol are used to collect shares that are later checked for consistency and additive errors using the $\Pi_{\text{check-cons}}$ and $\Pi_{\text{check-mac}}$ protocols respectively.

Complexity Analysis. The communication complexity of Π_{mult}, to multiply two input packed sharings, is $O(n)$ field elements while the computation complexity is $O(n^2)$. The round complexity of the protocol is 2 assuming the random sharings required in the protocol have been generated beforehand.

Lemma 5. Π_{mult} (Protocol 5) t-securely realizes $\mathcal{F}_{\text{mult}}$ (Functionality 5) in the $\{\mathcal{F}_{\text{rand}}, \mathcal{F}_{\text{zero}}\}$-hybrid model.

Proof. Let $\mathcal{A}$ denote the adversary and C be the set of corrupt parties. Let $\mathcal{H}$ be the set of honest parties, $\mathcal{H}_{\mathcal{H}}$ be a fixed subset of $\mathcal{H}$ of size $\mathsf{d}+1$, and $\mathcal{H}_C = \mathcal{H} \setminus \mathcal{H}_{\mathcal{H}}$. We construct a simulator Sim to simulate the behavior of honest parties as follows.

- Sim emulates $\mathcal{F}_{\text{rand}}$ and receives the shares of the corrupt parties in $[\mathbf{r}]$ and the additive error to the shares Δ_r from $\mathcal{A}$. Sim emulates $\mathcal{F}_{\text{zero}}$ and receives the shares of the corrupt parties in $[\mathbf{o}]_{n-1}$ from $\mathcal{A}$. It then computes the shares of the corrupt parties in $[\mathbf{r}]_{n-1}$ by adding the shares of corrupt parties in $[\mathbf{r}]$ and $[\mathbf{o}]_{n-1}$.
- Sim receives the shares of corrupt parties in $[\mathbf{x}]^{\mathcal{H}}$ and $[\mathbf{y}]^{\mathcal{H}}$ as well as the vectors Δ_x and Δ_y from $\mathcal{F}_{\text{mult}}$.
- Sim knows the shares of the corrupt parties in $[\mathbf{x}]$, $[\mathbf{y}]$, and $[\mathbf{r}]_{n-1}$ and can thus compute the shares of the corrupt parties in $[\mathbf{u}]_{n-1}$. It then samples $\mathbf{u} \leftarrow \mathbb{F}^{\ell}$ uniformly at random and computes the whole sharing $[\mathbf{u}]_{n-1}$ such that the shares of the corrupt parties are consistent. It then sends the shares of the honest parties in $[\mathbf{u}]_{n-1}$ to $\mathcal{A}$.

- Sim receives the shares of honest parties in $[\mathbf{u}]$ from $\mathcal{A}$ and computes the whole sharing $[\mathbf{u}]$ and $[\mathbf{u}]^{\mathcal{H}}$. It then computes the shares of corrupt parties in $[\mathbf{z}]$ using the shares of corrupt parties in $[\mathbf{u}]$ and $[\mathbf{r}]$ and sends it to $\mathcal{F}_{\mathsf{mult}}$.

- Sim reconstructs the secret $\bar{\mathbf{u}}$ from $[\mathbf{u}]$, computes $\boldsymbol{\delta}_z = \bar{\mathbf{u}} - \mathbf{u}$ and sends it to $\mathcal{F}_{\mathsf{mult}}$. It computes $\Delta_z = [\mathbf{u}] - [\mathbf{u}]^{\mathcal{H}} - \Delta_r$ and sends it to $\mathcal{F}_{\mathsf{mult}}$.

We now show that the view of $\mathcal{A}$ is identically distributed in the real and ideal worlds. Since Sim honestly emulates $\mathcal{F}_{\mathsf{rand}}$ and $\mathcal{F}_{\mathsf{zero}}$, the only difference in $\mathcal{A}$'s view is that it receives the shares of honest parties for a random degree-$(n-1)$ sharing rather than for $[\mathbf{u}]_{n-1}$ computed as in the protocol. However, since $\mathcal{F}_{\mathsf{rand}}$ samples $\mathbf{r}$ uniformly at random, $\mathbf{u}$ is also uniformly random in the real world. Moreover, since $[\mathbf{o}]_{n-1}$ and thus $[\mathbf{r}]_{n-1}$ are random degree-$(n-1)$ packed Shamir sharings, $[\mathbf{u}]_{n-1}$ is also a random degree-$(n-1)$ packed Shamir sharing given the secret $\mathbf{u}$ and the shares of the corrupt parties. It thus follows that $\mathcal{A}$'s view is identically distributed in both worlds.

We now show that the output of honest parties is identically distributed in both worlds conditioned on the view of the adversary.

- Note that $\mathcal{A}$'s view fixes the value of $\mathbf{u}$ and $\bar{\mathbf{u}}$ where $\bar{\mathbf{u}}$ is underlying secret for the shares sent by P_1. In the real world, we have $\mathbf{z} = \bar{\mathbf{u}} - \mathbf{r}$. In the ideal world, $\mathcal{F}_{\mathsf{mult}}$ reconstructs $\mathbf{z}$ from $[\mathbf{x}] \cdot [\mathbf{y}]$ which is equal to $\mathbf{u} - \mathbf{r}$. $\mathcal{F}_{\mathsf{mult}}$ then computes $[\mathbf{z}]^{\mathcal{H}}$ using $\mathbf{z} + \boldsymbol{\delta}_z$ as the secret where Sim computes $\boldsymbol{\delta}_z = \bar{\mathbf{u}} - \mathbf{u}$. Thus $\mathcal{F}_{\mathsf{mult}}$ computes $[\mathbf{z}]^{\mathcal{H}}$ over the secret $\mathbf{z} + \bar{\mathbf{u}} - \mathbf{u} = \mathbf{u} - \mathbf{r} + \bar{\mathbf{u}} - \mathbf{u} = \bar{\mathbf{u}} - \mathbf{r}$ which is the same as in the real world. Thus, the secrets for the output sharing are identically distributed in the real and ideal worlds.

- In the real world, the shares of corrupt parties in the output sharing correspond to $[\mathbf{z}] = [\mathbf{u}] - [\mathbf{r}]$. Note that the shares of corrupt parties in $[\mathbf{z}]$ are equal to those in $[\mathbf{z}]^{\mathcal{H}}$ by definition. In the ideal world, Sim honestly emulates $\mathcal{F}_{\mathsf{rand}}$ and receives the shares of the corrupt parties in $[\mathbf{r}]$ while it computes the shares of the corrupt parties in $[\mathbf{u}]$ from the shares of the honest parties sent by $\mathcal{A}$. It follows that the shares of corrupt parties are identical in the real and ideal worlds.

- In the real world, $[\mathbf{z}] = [\mathbf{u}] - [\mathbf{r}] = [\mathbf{u}]^{\mathcal{H}} + \Delta_u - [\mathbf{r}]^{\mathcal{H}} - \Delta_r = [\mathbf{z}]^{\mathcal{H}} + \Delta_u - \Delta_r$ and thus the additive errors to the output sharing is $\Delta_u - \Delta_r$. In the ideal world, Sim computes $\Delta_z = [\mathbf{u}] - [\mathbf{u}]^{\mathcal{H}} - \Delta_r = \Delta_u - \Delta_r$. Thus, the additive errors to the output shares are identically distributed in the real and ideal worlds.

The degree-d sharing $[\mathbf{z}]$ distributed by $\mathcal{F}_{\mathsf{mult}}$ is completely determined by the secrets $\mathbf{z}$, t shares of the corrupt parties and the additive errors to the shares of honest parties in $\mathcal{H}_C$. Since the latter are identically distributed in the real and ideal worlds, it follows that the shares of the honest parties are identically distributed in the real and ideal worlds.

Thus, the joint distribution consisting of the view of $\mathcal{A}$ and the output of honest parties is identically distributed in the real and ideal worlds which proves the security of Π_{mult}. □

5.6 Share Transformation

In this section, we present the protocol Π_{trans} (Protocol 8) that is used to transform secrets from one set of positions pos to another set of positions pos′ while applying a linear function f on the secrets. The protocol is identical to the one presented in [GPS22]. The $\Pi_{\mathsf{auth\text{-}trans}}$ protocol (Protocol 9) uses Π_{trans} to transform authenticated packed sharings from one set of positions to the other while applying a linear function. The $\Pi_{\mathsf{rand\text{-}sharing}}$ protocol (Protocol 7) allows parties to compute pre-processing data required to run Π_{trans}. Given pos, pos′, and a linear function f, $\Pi_{\mathsf{rand\text{-}sharing}}$ outputs a pair of packed sharings over uniformly random secrets; one where the secrets are shared over pos and the other where the secrets are shared over pos′ after applying f.

Functionality 6: $\mathcal{F}_{\text{rand-sharing}}$

1. $\mathcal{F}_{\text{rand-sharing}}$ receives the input pos, pos', and f from the honest parties.
2. $\mathcal{F}_{\text{rand-sharing}}$ samples $\mathbf{s} \leftarrow \mathbb{F}^{\ell}$ uniformly at random.
3. $\mathcal{F}_{\text{rand-sharing}}$ receives the shares of the corrupt parties $\left\{(v_i, v'_i)\right\}_{i \in C}$ and a vector $\boldsymbol{\delta} \in \mathbb{F}^{\ell}$. It then computes $[\mathbf{s}\,|\mathsf{pos}]_{n-1} \leftarrow \mathsf{share}(n-1, \mathbf{s}, \{v_i\}_{i \in C}; \mathsf{pos})$ and $[f(\mathbf{s})\,|\mathsf{pos}']^{\mathcal{H}} = \mathsf{share}(\mathsf{d}, f(\mathbf{s}) + \boldsymbol{\delta}, \left\{v'_i\right\}_{i \in C}; \mathsf{pos}')$.
4. $\mathcal{F}_{\text{rand-sharing}}$ receives a vector $\Delta \in \mathbb{F}^{n}$ from Sim such that for all $\mathsf{P}_i \notin \mathcal{H}_C$, $(\Delta)_i = 0$. It then computes $[f(\mathbf{s})\,|\mathsf{pos}'] = [f(\mathbf{s})\,|\mathsf{pos}']^{\mathcal{H}} + \Delta$.
5. $\mathcal{F}_{\text{rand-sharing}}$ distributes the shares of $[\mathbf{s}\,|\mathsf{pos}]_{n-1}$ and $[f(\mathbf{s})\,|\mathsf{pos}']$ to honest parties.

Let pos and pos' be ordered sets such that $\mathsf{pos}, \mathsf{pos}' \subset \left\{\mathsf{slot}^{\mathsf{C}}_1, \ldots, \mathsf{slot}^{\mathsf{C}}_W\right\} \cup \mathsf{pos}_{\mathsf{def}}$. The $\mathcal{F}_{\text{rand-sharing}}$ functionality (Functionality 6), realized by $\Pi_{\text{rand-sharing}}$, outputs a pair of sharings $\left([\mathbf{s}\,|\mathsf{pos}]_{n-1}, [f(\mathbf{s})\,|\mathsf{pos}']\right)$ when given pos, pos', and f as inputs, where $\mathbf{s}$ is a vector sampled uniformly at random. In Π_{trans}, parties invoke $\mathcal{F}_{\text{rand-sharing}}$ and use the degree-$(n-1)$ sharing to mask the input and reconstruct it towards P_1. P_1 then reconstructs the secret, locally applies the linear function f, and re-shares the transformed (masked) secrets over pos'. Parties can then locally unmask the secret using their share in $[f(\mathbf{s})\,|\mathsf{pos}']$.

The protocol $\Pi_{\text{auth-trans}}$ (Protocol 9) is used to transform secrets in authenticated sharings from one set of positions to the other while applying a linear function. The set $\mathcal{S}_{\text{cons}}$ input to the protocol is used to collect shares that are later checked for consistency using the $\Pi_{\text{check-cons}}$ protocol (Protocol 13).

The protocol $\Pi_{\text{rand-sharing}}$ (Protocol 7) realizes the functionality $\mathcal{F}_{\text{rand-sharing}}$. $\Pi_{\text{rand-sharing}}$ uses the fact that share is a linear function and can thus be computed over secret shares. Thus, parties first generate random sharings for the secrets and randomness used in the share algorithm and then compute share in a distributed manner using the secret shares. $\Pi_{\text{rand-sharing}}$ achieves low communication complexity by generating a batch of ℓ pairs of transformed random sharings in each run. Specifically, parties use packed sharings to compute ℓ parallel instances of share over different inputs $(\mathsf{pos}_i, \mathsf{pos}'_i, f_i)$ for each $i \in [1, \ell]$. However, this involves multiplying the packed shares with a vector of linear combiners corresponding to the ℓ parallel instances of share being computed. This is done by utilizing the multiplicative friendliness property described in Section 3.2, that enables multiplying a publicly known vector with a packed sharing.

Note that Π_{trans} provides perfect privacy against a malicious adversary but guarantees correctness only against a semi-honest adversary. Looking ahead, we verify that the output sharings dealt by P_1 are consistent using $\Pi_{\text{check-cons}}$ (Protocol 13). However, it is not straightforward to verify the correctness of the transformation using the MAC check since the secrets are shared over different positions in the input and output while the MAC check requires that the secrets for all sharings being verified are shared over the same positions $\mathsf{pos}_{\mathsf{def}}$. Consequently, we show that a series of calls to Π_{trans} only accumulates additive errors to the secret. A transformed sharing can then by verified for correctness by first transforming the secrets to be shared over $\mathsf{pos}_{\mathsf{def}}$ and then using the MAC check. For more details, we refer the reader to Section 6.2.

Complexity Analysis. $\Pi_{\text{rand-sharing}}$ has an amortized communication complexity of $O(n)$ field elements, an amortized computation complexity of $O\left(n^2\right)$, and a round complexity of 2 for generating a single pair of transformed random sharings. Π_{trans} has a communication complexity of $O(n)$ field elements and a computation complexity of $O\left(n^2\right)$ for transforming an input packed sharing. The round complexity of Π_{trans}, assuming the random sharings have been generated beforehand, is 2.

Lemma 6. $\Pi_{\text{rand-sharing}}$ (Protocol 7) t-securely realizes ℓ invocations of $\mathcal{F}_{\text{rand-sharing}}$ (Functionality 6) in the

Protocol 7: $\Pi_{\mathsf{rand\text{-}sharing}}(\{\mathsf{pos}_i, \mathsf{pos}'_i, f_i\}_{i=1}^{\ell})$

1. Parties invoke $\mathcal{F}_{\mathsf{zero}}$ $2n$ times and obtain zero sharings $[\mathbf{o}_1]_{n-1}, \dots, [\mathbf{o}_{2n}]_{n-1}$.
2. Parties invoke $\mathcal{F}_{\mathsf{rand}}$ $n+t$ times and receive random sharings $[\mathbf{u}_1], \dots, [\mathbf{u}_{n+t}]$.
3. Let $\mathbf{s}_i = \left\{(\mathbf{u}_j)_i\right\}_{j=1}^{\ell}$ for each $i \in [1, \ell]$ i.e., $\mathbf{s}_i$ is the vector consisting of the i-th secret in $\mathbf{u}_1, \dots, \mathbf{u}_\ell$. Similarly, let $\mathbf{r}_i^1 = \left\{(\mathbf{u}_j)_i\right\}_{j=\ell+1}^{\ell+t}$ and $\mathbf{r}_i^2 = \left\{(\mathbf{u}_j)_i\right\}_{j=\ell+t+1}^{n+t}$ for each $i \in [1, \ell]$. We define $[\mathbf{s}_i \,|\mathsf{pos}_i]_{n-1}$ and $[f_i(\mathbf{s}_i) \,|\mathsf{pos}'_i]$ as the sharings

$$[\mathbf{s}_i \,|\mathsf{pos}_i]_{n-1} = \mathsf{share}(n-1, \mathbf{s}_i; \mathsf{pos}_i, \mathbf{r}_i^2)$$
$$[f_i(\mathbf{s}_i) \,|\mathsf{pos}'_i] = \mathsf{share}(\mathsf{d}, f_i(\mathbf{s}_i); \mathsf{pos}'_i, \mathbf{r}_i^1).$$

Since share is linear, the j-th share in $[\mathbf{s}_i \,|\mathsf{pos}_i]_{n-1}$ can be expressed as a linear combination of elements in $\mathbf{s}_i$ and $\mathbf{r}_i^2$. Specifically, we have

$$([\mathbf{s}_i \,|\mathsf{pos}_i]_{n-1})_j = \sum_{k=1}^{\ell} \alpha_{i,k}^j (\mathbf{s}_i)_k + \sum_{k=1}^{n-\ell} \alpha_{i,\ell+t+k}^j (\mathbf{r}_i^2)_k = \sum_{k=1}^{\ell} \alpha_{i,k}^j (\mathbf{u}_k)_i + \sum_{k=\ell+t+1}^{n+t} \alpha_{i,k}^j (\mathbf{u}_k)_i.$$

Similarly, the j-th share in $[f_i(\mathbf{s}_i) \,|\mathsf{pos}'_i]$ can be expressed as the linear combination $\sum_{k=1}^{\ell+t} \beta_{i,k}^j (\mathbf{u}_k)_i$. Let $\boldsymbol{\alpha}_k^j = (\alpha_{1,k}^j, \dots, \alpha_{\ell,k}^j)$ for each $j \in [1, \ell] \cup [\ell+t+1, n+t]$ and let $\boldsymbol{\beta}_k^j = (\beta_{1,k}^j, \dots, \beta_{\ell,k}^j)$ for each $j \in [1, \ell+t]$. Parties compute

$$[\mathbf{x}_j]_{n-1} = \sum_{k=1}^{\ell} \boldsymbol{\alpha}_k^j [\mathbf{u}_k] + \sum_{k=\ell+t+1}^{n+t} \boldsymbol{\alpha}_k^j [\mathbf{u}_k] + [\mathbf{o}_j]_{n-1}$$
$$[\mathbf{y}_j]_{n-1} = \sum_{k=1}^{\ell+t} \boldsymbol{\beta}_k^j [\mathbf{u}_k] + [\mathbf{o}_{n+j}]_{n-1}$$

and send their shares in $[\mathbf{x}_j]_{n-1}$ and $[\mathbf{y}_j]_{n-1}$ to P_j for each $j \in [1, n]$.
4. Each $\mathsf{P}_i \in \mathcal{P}$ receives the whole sharing $[\mathbf{x}_i]_{n-1}$ and $[\mathbf{y}_i]_{n-1}$ and reconstructs the secrets $\mathbf{x}_i$ and $\mathbf{y}_i$.
5. Each $\mathsf{P}_i \in \mathcal{P}$ outputs its share $([\mathbf{s}_j \,|\mathsf{pos}_j]_{n-1})_i = (\mathbf{x}_i)_j$ and $([f_j(\mathbf{s}_j) \,|\mathsf{pos}'_j])_i = (\mathbf{y}_i)_j$ for each $j \in [1, \ell]$.

$\{\mathcal{F}_{\mathsf{rand}}, \mathcal{F}_{\mathsf{zero}}\}$-hybrid model.

Proof. Let $\mathcal{A}$ denote the adversary and C be the set of corrupt parties. Let $\mathcal{H}$ be the set of honest parties, $\mathcal{H}_{\mathcal{H}}$ be a fixed subset of $\mathcal{H}$ of size $\mathsf{d}+1$, and $\mathcal{H}_C = \mathcal{H} \setminus \mathcal{H}_{\mathcal{H}}$.

At a high level, since the shares of the corrupt parties in the output sharing are uniformly random, it is straightforward to the simulate $\mathcal{A}$'s view in the ideal world. The remainder of the work done by Sim is in extracting the additive errors to the secret and additive errors to the shares. Specifically, Sim computes the errors $\mathbf{X}_i$ and $\mathbf{Y}_i$ to the secrets $\mathbf{x}_i$ and $\mathbf{y}_i$ reconstructed by honest party P_i, using the additive errors $\left\{\Delta_{u_j}\right\}_{j=1}^{n+t}$ sent by $\mathcal{A}$ to $\mathcal{F}_{\mathsf{rand}}$ as well as from the errors in the shares of honest parties sent by $\mathcal{A}$ during reconstruction. Note that $\mathbf{x}_i$ and $\mathbf{y}_i$ correspond to the shares of P_i in the output sharing and thus $\mathbf{X}_i$ and $\mathbf{Y}_i$ are additive errors to these output shares. The simulator then translates $\mathbf{X}_i$ for all parties $\mathsf{P}_i \in \mathcal{P}$ and $\mathbf{Y}_i$ for every $\mathsf{P}_i \in \mathcal{H}_{\mathcal{H}}$ to an additive error to the secret of the output sharings $[f_j(\mathbf{s}_\mathrm{j}) \,|\mathsf{pos}']$ for each $j \in [1, \ell]$. The errors $\mathbf{Y}_i$ for every $\mathsf{P}_i \in \mathcal{H}_C$ is set as the additive errors to the shares in $[f_j(\mathbf{s}_\mathrm{j}) \,|\mathsf{pos}']$ for each $j \in [1, \ell]$. We proceed to formally describe the simulator Sim.

Protocol 8: $\Pi_{\mathsf{trans}}([\mathbf{x}\,|\mathsf{pos}]_{\mathsf{d}+\ell-1}, \mathsf{pos}', f)$

1. Parties invoke $\mathcal{F}_{\mathsf{rand\text{-}sharing}}$ with inputs pos, pos′, and f to receive the sharings $[\mathbf{r}\,|\mathsf{pos}]_{n-1}$ and $[f(\mathbf{r})\,|\mathsf{pos}']$.
2. Parties locally compute $[\mathbf{x}+\mathbf{r}\,|\mathsf{pos}]_{n-1} = [\mathbf{x}\,|\mathsf{pos}]_{\mathsf{d}+\ell-1} + [\mathbf{r}\,|\mathsf{pos}]_{n-1}$ and send their shares of $[\mathbf{x}+\mathbf{r}\,|\mathsf{pos}]_{n-1}$ to P_1.
3. P_1 receives the whole sharing $[\mathbf{x}+\mathbf{r}\,|\mathsf{pos}]_{n-1}$, reconstructs the secrets $\mathbf{x}+\mathbf{r}$, and computes $f(\mathbf{x}+\mathbf{r})$.
4. P_1 computes $[f(\mathbf{x}+\mathbf{r})\,|\mathsf{pos}'] \leftarrow \mathsf{share}(\mathsf{d}, f(\mathbf{x}+\mathbf{r}), \mathsf{pos}')$ and sends the i-th share $([f(\mathbf{x}+\mathbf{r})\,|\mathsf{pos}'])_i$ to P_i.
5. Parties receive their share in $[f(\mathbf{x}+\mathbf{r})\,|\mathsf{pos}']$ and output $[f(\mathbf{x})\,|\mathsf{pos}'] = [f(\mathbf{x}+\mathbf{r})\,|\mathsf{pos}'] - [f(\mathbf{r})\,|\mathsf{pos}']$.

Protocol 9: $\Pi_{\mathsf{auth\text{-}trans}}([\![\mathbf{x}\,|\mathsf{pos}]\!]_{\mathsf{d}+\ell-1}, \mathsf{pos}', f, \mathcal{S}_{\mathsf{cons}})$

1. Parties run $\Pi_{\mathsf{trans}}([\mathbf{x}\,|\mathsf{pos}]_{\mathsf{d}+\ell-1}, \mathsf{pos}', f)$ to receive the transformed sharing $[f(\mathbf{x})\,|\mathsf{pos}']$.
2. Parties run $\Pi_{\mathsf{trans}}([\mathsf{kmac}_i \cdot \mathbf{x}\,|\mathsf{pos}]_{\mathsf{d}+\ell-1}, \mathsf{pos}', f)$ to receive the sharing $[\mathsf{kmac}_i \cdot f(\mathbf{x})\,|\mathsf{pos}']$ for each $i \in [1, L_{\mathsf{mac}}]$.
3. Parties set $\mathcal{S}_{\mathsf{cons}} := \mathcal{S}_{\mathsf{cons}} \cup \{[f(\mathbf{x})\,|\mathsf{pos}']\} \cup \{[\mathsf{kmac}_i \cdot f(\mathbf{x})\,|\mathsf{pos}']\}_{i=1}^{L_{\mathsf{mac}}}$.
4. Parties output $[\![f(\mathbf{x})\,|\mathsf{pos}']\!] = ([f(\mathbf{x})\,|\mathsf{pos}'], \{[\mathsf{kmac}_i \cdot f(\mathbf{x})\,|\mathsf{pos}']\}_{i=1}^{L_{\mathsf{mac}}})$.

— Sim honestly emulates $\mathcal{F}_{\mathsf{zero}}$ and receives the shares of the corrupt parties in $[\mathbf{o}_i]_{n-1}$ for each $i \in [1, 2n]$.

— Sim honestly emulates $\mathcal{F}_{\mathsf{rand}}$ and receives the shares of the corrupt parties in $[\mathbf{u}_i]$ and the additive errors to the shares Δ_{u_i} for each $i \in [1, n+t]$.

— Sim computes the shares of the corrupt parties in $[\mathbf{x}_i]_{n-1}$ and $[\mathbf{y}_i]_{n-1}$ as in the protocol for every $i \in [1, n]$. It also computes $\Delta_{x_i} = \sum_{k=1}^{\ell} [\boldsymbol{\alpha}_k^j]_{\ell-1}\Delta_{u_k} + \sum_{k=\ell+t+1}^{n-\ell} [\boldsymbol{\alpha}_k^j]_{\ell-1}\Delta_{u_k}$ and $\Delta_{y_i} = \sum_{k=1}^{\ell+t} [\boldsymbol{\beta}_k^j]_{\ell-1}\Delta_{u_k}$.

— For each $i \in [1, n]$, Sim does one of the following based on the party P_i towards which the shares $[\mathbf{x}_i]_{n-1}$ and $[\mathbf{y}_i]_{n-1}$ are being reconstructed.

  – If P_i is corrupt, Sim samples $\mathbf{x}_i, \mathbf{y}_i \leftarrow \mathbb{F}^{\ell}$ uniformly at random and computes the sharing $[\mathbf{x}_i]_{n-1}$ and $[\mathbf{y}_i]_{n-1}$ such that the shares of the corrupt parties, as computed in the previous step, are consistent. It then sends the shares of honest parties in $[\mathbf{x}_i]_{n-1}$ and $[\mathbf{y}_i]_{n-1}$ to $\mathcal{A}$.

  – If $\mathsf{P}_i \in \mathcal{H}$, Sim receives the shares of corrupt parties in $[\overline{\mathbf{x}}_i]_{n-1}$ and $[\overline{\mathbf{y}}_i]_{n-1}$ where we use $[\overline{\mathbf{x}}]_{n-1}$ and $[\overline{\mathbf{y}}]_{n-1}$ to denote the fact that the shares of corrupt parties might differ from those in $[\mathbf{x}_i]_{n-1}$ and $[\mathbf{y}_i]_{n-1}$. Sim then computes a degree-$(n-1)$ sharing $[\mathbf{X}_i]_{n-1}$ as follows. The share of each party $\mathsf{P}_j \in \mathcal{H}_{\mathcal{H}}$ is set to 0 i.e., $([\mathbf{X}_i]_{n-1})_j = 0$. The share of each corrupt party $\mathsf{P}_j \in C$ is computed as $([\mathbf{X}_i]_{n-1})_j = ([\overline{\mathbf{x}}_i]_{n-1})_j - ([\mathbf{x}_i]_{n-1})_j$. The share of each honest party $\mathsf{P}_j \in \mathcal{H}_C$ is computed as $([\mathbf{X}_i]_{n-1})_j = (\Delta_{x_i})_j$. Sim computes $[\mathbf{Y}_i]_{n-1}$ in a similar manner. It then reconstructs the secrets $\mathbf{X}_i$ and $\mathbf{Y}_i$ from $[\mathbf{X}_i]_{n-1}$ and $[\mathbf{Y}_i]_{n-1}$ respectively.

— Sim does the following for the i-th invocation of $\mathcal{F}_{\mathsf{rand\text{-}sharing}}$ for each $i \in [1, \ell]$

  – Sim sends $\left\{((\mathbf{x}_i)_j, (\mathbf{y}_i)_j)\right\}_{\mathsf{P}_i \in C}$ to $\mathcal{F}_{\mathsf{rand\text{-}sharing}}$ as the shares of corrupt parties.

- Sim computes a degree-$(n-1)$ sharing $[\boldsymbol{\delta}_{s_i}]_{n-1}$ such that $([\boldsymbol{\delta}_{s_i}]_{n-1})_j = (\mathbf{X}_j)_i$ for every $\mathsf{P}_j \in \mathcal{H}$ and $([\boldsymbol{\delta}_{s_i}]_{n-1})_j = 0$ otherwise. It computes a degree-d sharing $[\boldsymbol{\delta}_{f_i}]$ such that $([\boldsymbol{\delta}_{f_i}])_j = (\mathbf{Y}_j)_i$ for every $\mathsf{P}_j \in \mathcal{H}_{\mathcal{H}}$ and $([\boldsymbol{\delta}_{f_i}]_{n-1})_j = 0$ otherwise. Sim reconstructs $\boldsymbol{\delta}_{s_i}$ and $\boldsymbol{\delta}_{f_i}$ and sends $\boldsymbol{\delta}_i = \boldsymbol{\delta}_{f_i} - f_i(\boldsymbol{\delta}_{s_i})$ to $\mathcal{F}_{\text{rand-sharing}}$ as the additive errors to the secrets.
- Sim sends Δ_i to $\mathcal{F}_{\text{rand-sharing}}$ where $(\Delta_i)_j = (\mathbf{Y}_j)_i$ for every $\mathsf{P}_j \in \mathcal{H}_C$ and $(\Delta_i)_j = 0$ otherwise.

We now show that the view of $\mathcal{A}$ is identically distributed in the real and ideal worlds. The main difference between the real and ideal worlds is that $\mathcal{A}$ sees the shares of honest parties in $[\mathbf{x}_i]_{n-1}$ and $[\mathbf{y}_i]_{n-1}$ for every $\mathsf{P}_i \in C$ whereas in the ideal world Sim generates the shares of the honest parties for a random degree-$(n-1)$ sharing. However, in the real world, $\mathbf{x}_i$ corresponds to the shares of P_i in $[\mathbf{s}_j\,|\text{pos}]_{n-1}$ and $\mathbf{y}_i$ corresponds to the shares of P_i in $[f_j(\mathbf{s}_j)\,|\text{pos}']$ for each $j \in [1, \ell]$. Since the shares of corrupt parties in degree-$(n-1)$ and degree-d sharings are uniformly distributed and independent of the secret, the secrets $\mathbf{x}_i$ and $\mathbf{y}_i$ are uniformly distributed in the real world. Moreover, since $[\mathbf{o}_i]_{n-1}$ and $[\mathbf{o}_{n+i}]_{n-1}$ are random degree-$(n-1)$ packed Shamir sharings, the sharings $[\mathbf{x}_i]_{n-1}$ and $[\mathbf{y}_i]_{n-1}$ are random degree-$(n-1)$ packed Shamir sharings given the secrets $\mathbf{x}_i$, $\mathbf{y}_i$ and the shares of the corrupt parties. Thus $\mathcal{A}$'s view is identically distributed in the real and ideal worlds.

We now show that the output of honest parties is identically distributed in both worlds conditioned on the view of $\mathcal{A}$. In the real world, every honest party $\mathsf{P}_i \in \mathcal{H}$ takes $\overline{\mathbf{x}}_i$ and $\overline{\mathbf{y}}_i$ that it reconstructs as its shares for $[\mathbf{s}_j\,|\text{pos}]_{n-1}$ and $[f_j(\mathbf{s}_j)\,|\text{pos}']$ respectively for each $j \in [1, \ell]$. We use $\overline{\mathbf{x}}_\mathbf{i}$ and $\overline{\mathbf{y}}_\mathbf{i}$ to denote that the reconstructed values are different from the expected $\mathbf{x}_i$ and $\mathbf{y}_i$ respectively, due to the additive errors to the random sharings and the errors $\hat{\mathbf{x}}_\mathbf{i}$ and $\hat{\mathbf{y}}_\mathbf{i}$ introduced by $\mathcal{A}$ to the shares of corrupt parties in $[\mathbf{x}_i]_{n-1}$ and $[\mathbf{y}_i]_{n-1}$ during reconstruction. We have

$$\begin{aligned}
[\overline{\mathbf{x}}_i]_{n-1} &= \sum_{k=1}^{\ell} \boldsymbol{\alpha}_k^i([\mathbf{u}_k]^{\mathcal{H}} + \Delta_{u_k}) + \sum_{k=\ell+t+1}^{n-\ell} \boldsymbol{\alpha}_k^i([\mathbf{u}_k]^{\mathcal{H}} + \Delta_{u_k}) + [\mathbf{o}_i]_{n-1} + \hat{\mathbf{x}}_i \\
&= [\mathbf{x}_i]_{n-1} + [\mathbf{X}_i]_{n-1} \\
[\overline{\mathbf{y}}_i]_{n-1} &= \sum_{k=1}^{\ell+t} \boldsymbol{\beta}_k^i([\mathbf{u}_k]^{\mathcal{H}} + \Delta_{u_k}) + [\mathbf{o}_{n+i}]_{n-1} + \hat{\mathbf{y}}_i \\
&= [\mathbf{y}_i]_{n-1} + [\mathbf{Y}_i]_{n-1}.
\end{aligned}$$

This implies that the secrets $\overline{\mathbf{s}}_j$ for the degree-$(n-1)$ packed Shamir shares output by parties in the real world correspond to $\overline{\mathbf{s}}_\mathbf{j} = \mathbf{s}_j + \boldsymbol{\delta}_{s_j}$ where $\mathbf{s}_j$ and $\boldsymbol{\delta}_{s_j}$ are defined by the sharings $[\mathbf{s}_j\,|\text{pos}_j]_{n-1} = ((\mathbf{x}_1)_j, \ldots, (\mathbf{x}_n)_j)$ and $[\boldsymbol{\delta}_{s_j}\,|\text{pos}_j]_{n-1} = ((\mathbf{X}_1)_j, \ldots, (\mathbf{X}_n)_j)$. On the other hand, since a degree-d sharing is defined by the shares of parties in $\mathcal{H}_{\mathcal{H}}$, the secrets $\overline{f_j(\mathbf{s}_j)}$ for the degree-d packed sharing output by parties in the real world correspond to $\overline{f_j(\mathbf{s})} = f_j(\mathbf{s}_j) + \boldsymbol{\delta}_{f_j}$ where $[f_j(\mathbf{s}_j)\,|\text{pos}']$ and $[\boldsymbol{\delta}_{f_j}\,|\text{pos}']$ are defined by the shares $\left\{(\mathbf{y}_i)_j\right\}_{i\in\mathcal{H}_{\mathcal{H}}}$ and $\left\{(\mathbf{Y}_i)_j\right\}_{i\in\mathcal{H}_{\mathcal{H}}}$. The errors $\mathbf{X}_i$ to the secrets $\mathbf{x}_i$ reconstructed by parties $\mathsf{P}_i \in \mathcal{H}_C$ would then correspond to additive errors Δ_{f_j} to the degree-d output sharings $[\overline{f_j(\mathbf{s}_j)}\,|\text{pos}']$. Note that $\mathcal{A}$'s view fixes the value of $\boldsymbol{\delta}_{s_j}$, $\boldsymbol{\delta}_{f_j}$, and Δ_{f_j}. $\mathbf{s}_j$ is uniformly random since it is sampled within $\mathcal{F}_{\text{rand}}$, which implies that $\overline{\mathbf{s}}_j$ is uniformly random. In the ideal world, $\mathcal{F}_{\text{rand-sharing}}$ samples $\mathbf{s}_j$ uniformly at random and so the underlying secrets for the degree-$(n-1)$ sharing are uniformly random in the real and ideal worlds. Moreover, note that $\overline{f_j(\mathbf{s}_j)} - f_j(\overline{\mathbf{s}}_j) = f_j(\mathbf{s}_j) + \boldsymbol{\delta}_{f_j} - f_j(\mathbf{s}_j) - f_j(\boldsymbol{\delta}_{s_j}) = \boldsymbol{\delta}_{f_j} - f_j(\boldsymbol{\delta}_{s_j}) = \boldsymbol{\delta}_j$ which is exactly the additive error to the secret sent by Sim to $\mathcal{F}_{\text{rand-sharing}}$. Since the simulator computes the values of $\boldsymbol{\delta}_{s_j}$, $\boldsymbol{\delta}_{f_j}$, and Δ_{f_j} correctly, it follows that the secrets and additive errors to the shares are identically distributed in the real and ideal worlds. Moreover, since $\mathcal{A}$'s view fixes the shares of the corrupt parties, it follows that the shares of honest parties in $[f_j(\mathbf{s}_j)\,|\text{pos}']$ are identically distributed in the real and ideal worlds since degree-d sharings are completely defined by the secrets, the shares of the corrupt parties and the additive errors to the shares which we have just shown to be identically distributed in the real and ideal worlds. Finally, $[\bar{\mathbf{s}}_j\,|\mathsf{pos}]_{n-1}$ is a random degree-$(n-1)$ sharing given the shares of the corrupt parties since $[\mathbf{s}_j\,|\mathsf{pos}]_{n-1}$ is a random degree-$(n-1)$ sharing given the shares of the corrupt parties. Sim sends the shares of the corrupt parties to $\mathcal{F}_{\text{rand-sharing}}$ and $\mathcal{F}_{\text{rand-sharing}}$ generates the shares of honest parties in $[\mathbf{s}_j\,|\mathsf{pos}]_{n-1}$ with the same distribution as in the real world.

Thus, the joint distribution consisting of the view of $\mathcal{A}$ and the output of honest parties is identically distributed in the real and ideal worlds which proves the security of $\Pi_{\text{rand-sharing}}$. □

5.7 Sharing MAC Keys

In this section, we describe the protocol $\Pi_{\text{mac-keygen}}$ (Section 5.7) that generates packed Shamir sharings of MAC keys. Specifically, since we re-order and re-pack secrets from different sharings in our MPC protocol, we require that each MAC key be of the form $\mathsf{kmac}\cdot\mathbf{1}$, where kmac is a random field element and $\mathbf{1}$ denotes a vector of all 1s. In other words, the MAC key share is a packed sharing of a secret vector of the form $(\mathsf{kmac},\ldots,\mathsf{kmac})$ where each element is identical. This ensures that packed sharings of the MAC always correspond to the same key kmac despite arbitrary re-ordering and re-packing of the secrets from different packed sharings. The protocol is similar to Π_{rand} except that parties also check that all output sharings are consistent and of the required form. This is done by using a similar approach as [BTH08], where a hyper-invertible matrix $\mathbf{H}_{n,n}$ is used to transform the shares dealt by each party and $2t$ transformed sharings are reconstructed towards individual parties and checked for correctness. The property of hyper-invertible matrices, combined with the fact that at least t parties that check the transformed sharings are honest implies that the output sharings are correct if all checks pass. The functionality $\mathcal{F}_{\text{mac-keygen}}$ that $\Pi_{\text{mac-keygen}}$ realizes is described in Functionality 7.

Complexity Analysis. The total communication complexity of the protocol is $O(n^2)$ field elements. However, each run of the protocol yields 2ℓ packed sharings and thus the amortized cost per packed sharing of a MAC key is $O(n)$ field elements. Similarly, the amortized computation complexity of the protocol for generating a packed sharing of a MAC key is $O(n^2)$. The round complexity of the protocol is 3.

Lemma 7. $\Pi_{\text{mac-keygen}}$ (Protocol 10) t-securely realizes 2ℓ invocations of $\mathcal{F}_{\text{mac-keygen}}$ (Functionality 7).

Proof. Let $\mathcal{A}$ denote the adversary and C be the set of corrupt parties. Let $\mathcal{H}$ be the set of honest parties, $\mathcal{H}_{\mathcal{H}}$ be a fixed subset of $\mathcal{H}$ of size $\mathsf{d}+1$, and $\mathcal{H}_C = \mathcal{H}\setminus\mathcal{H}_{\mathcal{H}}$. We construct a simulator Sim to simulate the behavior of honest parties.

The simulator Sim works as follows.

- Sim sends uniformly random values as the shares of corrupt parties in $[u_i]$ for every $\mathsf{P}_i \in \mathcal{H}$ and receives the shares of honest parties in $[\bar{\mathbf{u}}_j]$ for every $\mathsf{P}_j \in C$ from $\mathcal{A}$. Here we use $\bar{\mathbf{u}}_j$ to denote any errors that $\mathcal{A}$ might have introduced.

Functionality 7: $\mathcal{F}_{\text{mac-keygen}}$

1. $\mathcal{F}_{\text{mac-keygen}}$ receives status $\in \{\texttt{continue}, \texttt{abort}\}$ from Sim.
2. If status $=$ `continue`, $\mathcal{F}_{\text{mac-keygen}}$ receives the shares of the corrupt parties $\{s_i\}_{i\in C}$ from Sim. It then samples $\mathsf{kmac} \leftarrow \mathbb{F}$ uniformly at random and computes $[\mathsf{kmac}] = \mathsf{share}(\mathsf{d}, \mathsf{kmac}\cdot(1)^{\ell}, \{s_i\}_{i\in C})$. It then distributes $[\mathsf{kmac}]$ to the honest parties.
3. Else, $\mathcal{F}_{\text{mac-keygen}}$ sends `abort` to honest parties.

Protocol 10: $\Pi_{\mathsf{mac\text{-}keygen}}$

1. Every party $\mathsf{P}_i \in \mathcal{P}$ locally samples $u_i \leftarrow \mathbb{F}$ uniformly at random, computes $[u_i] \leftarrow \mathsf{share}(\mathsf{d}, u_i \cdot \mathbf{1})$[a], and then sends the j-th share $([u_i])_j$ to P_j for each $j \in [1, n]$.
2. Having received their shares in $[u_1], \ldots, [u_n]$, parties locally compute
$$[\mathsf{kmac}_1], \ldots, [\mathsf{kmac}_n] = \mathbf{H}_{n,n} \cdot ([u_1], \ldots, [u_n]).$$
3. Parties send their shares in $[\mathsf{kmac}_i]$ to P_i for each $i \in [1, 2t]$. P_i receives the whole sharing $[\mathsf{kmac}_i]$ and checks if all the shares form a valid degree-d packed Shamir sharing such that the reconstructed secret is of the form $\mathsf{kmac}_i \cdot \mathbf{1}$ for each $i \in [1, 2t]$. If the check fails, P_i sends `abort` to all parties else it sends `continue` to all parties.
4. If no party P_i for any $i \in [1, 2t]$ sent `abort`, parties output their shares in $[\mathsf{kmac}_{2t+1}], \ldots, [\mathsf{kmac}_n]$.

[a] $\mathbf{1}$ denotes the vector of all ones $(1, \ldots, 1)$.

- It then computes the whole sharing $[\overline{\mathbf{u}}_j]$ and $[\overline{\mathbf{u}}_j]^{\mathcal{H}}$, reconstructs the secret $\overline{\mathbf{u}}_\mathbf{j}$ and sets $\Delta_{u_j} = [\overline{\mathbf{u}}_j] - [\overline{\mathbf{u}}_j]^{\mathcal{H}}$ and $\boldsymbol{\delta}_{u_j} = (\overline{\mathbf{u}}_j)_1 \cdot \mathbf{1} - \overline{\mathbf{u}}_j$ for every $\mathsf{P}_j \in \mathcal{C}$. It sets $\Delta_{u_j} = \mathbf{0}$ and $\boldsymbol{\delta}_{u_j} = \mathbf{0}$ for each $\mathsf{P}_j \in \mathcal{H}$.
- Note that Sim now knows the shares of corrupt parties for $[u_i]$ for every $\mathsf{P}_i \in \mathcal{P}$. It can thus compute the shares of the corrupt party in $[\mathsf{kmac}_j]$ for each $j \in [1, n]$ by multiplying the shares of the corrupt parties in $[u_i]$ with $\mathbf{H}_{n,n}$ as in the protocol.
- Sim computes $(\Delta_1, \ldots, \Delta_n) = \mathbf{H}_{n,n}(\Delta_{u_1}, \ldots, \Delta_{u_n})$ and $(\boldsymbol{\delta}_1, \ldots, \boldsymbol{\delta}_n) = \mathbf{H}_{n,n}(\boldsymbol{\delta}_{u_1}, \ldots, \boldsymbol{\delta}_{u_n})$. Sim samples $\mathsf{kmac}_i \leftarrow \mathbb{F}$ uniformly at random, computes the sharing $[\mathsf{kmac}_i + \boldsymbol{\delta}_i]$ such that the shares of the corrupt parties are consistent. For every corrupt party $\mathsf{P}_i \in \mathcal{C}$ such that $i \in [1, 2t]$, Sim sends the shares of honest parties in $[\mathsf{kmac}_i + \boldsymbol{\delta}_i] + \Delta_i$ to $\mathcal{A}$. For every honest party $\mathsf{P}_i \in \mathcal{H}$ such that $i \in [1, 2t]$, Sim receives the shares of the corrupt parties for $[\overline{\mathsf{kmac}_i}]$ from $\mathcal{A}$. If corrupt parties shares in $[\overline{\mathsf{kmac}_i}]$ are different from those that Sim computed in $[\mathsf{kmac}_i]$ then Sim sends `abort` to every invocation of $\mathcal{F}_{\mathsf{mac\text{-}keygen}}$.
- If $\boldsymbol{\delta}_{u_i} \neq 0$ or $\Delta_{u_i} \neq 0$ for any $\mathsf{P}_i \in \mathcal{C}$, Sim sends `abort` to all invocations of $\mathcal{F}_{\mathsf{mac\text{-}keygen}}$. Else, it sends `continue` to all invocations of $\mathcal{F}_{\mathsf{mac\text{-}keygen}}$. It sends the shares of the corrupt parties in $[\mathsf{kmac}_i]$ to the i-th invocation of $\mathcal{F}_{\mathsf{mac\text{-}keygen}}$ for every $j \in [2t + 1, n]$.

We now show that $\mathcal{A}$'s view in the real world is identically distributed to its view in the simulation. We use $x + \boldsymbol{\delta}$ to denote $x \cdot \mathbf{1} + \boldsymbol{\delta}$ for brevity. The shares of corrupt parties in $[u_i]$ for each $\mathsf{P}_i \in \mathcal{H}$ is uniformly random in the real world since it is a degree-d sharing. Sim sends uniformly random values to $\mathcal{A}$ as the shares of corrupt parties too and so the shares of the corrupt parties are identically distributed. Let $[\overline{\mathbf{u}}_\mathbf{j}] = [\mathbf{u}_j + \boldsymbol{\delta}_{u_j}] + \Delta_{u_j}$ where $\mathbf{u}_j = (\overline{\mathbf{u}}_j)_1 \cdot \mathbf{1}$. We have

$$\begin{aligned}([\mathsf{kmac}_1 + \delta] + \Delta_1, \ldots, [\mathsf{kmac}_n + \delta] + \Delta_n) = \\ \mathbf{H}_{n,n}([u_1 + \boldsymbol{\delta}_{u_1}] + \Delta_{u_1}, \ldots, [u_n + \boldsymbol{\delta}_{u_n}] + \Delta_{u_n})\end{aligned}$$

where $\boldsymbol{\delta}_{u_i} = \mathbf{0}$ and $\Delta_{u_i} = \mathbf{0}$ for $\mathsf{P}_i \in \mathcal{H}$. Thus, $\boldsymbol{\delta}_i$ and Δ_i computed by Sim are identical to those in the real world. Moreover, given the secrets $\overline{\mathbf{u}}_i$ shares by corrupt parties $\mathsf{P}_i \in \mathcal{C}$, the secrets kmac_i for a t sized subset of $i \in [1, 2t]$ revealed to $\mathcal{A}$ are uniformly random since secrets shared by honest parties are uniformly random. It follows that the view of $\mathcal{A}$ in the simulation is identically distributed to that in the real world.

We now show that the shares output by honest parties is identically distributed in the real and ideal worlds conditioned on the view of $\mathcal{A}$.

- Let $\mathbf{H}^{\mathcal{H}}_{n,n-t}$ and $\mathbf{H}^{C}_{n,t}$ be the sub-matrix of $\mathbf{H}_{n,n}$ containing columns with indices in $\mathcal{H}$ and C respectively. Let $[\mathsf{kmac}_i + \boldsymbol{\delta}_i]$ denote the shares computed by parties in the real world for every $i \in [1, n]$, where $\boldsymbol{\delta}_i$ is the additive error to the secret because of corrupt parties $\mathsf{P}_j \in C$ that share secrets of the form $\overline{\mathbf{u}}_j = u_j \cdot \mathbf{1} + \boldsymbol{\delta}_{u_j}$ instead of $u_j \cdot \mathbf{1}$. We have

  $$\begin{aligned}(\mathsf{kmac}_1 + \boldsymbol{\delta}_1, \ldots, \mathsf{kmac}_n + \boldsymbol{\delta}_n) &= \mathbf{H}_{n,n}(u_1 + \boldsymbol{\delta}_{u_1}, \ldots, u_n + \boldsymbol{\delta}_{u_n}) \\ &= \mathbf{H}^{\mathcal{H}}_{n,n-t}(u_i + \boldsymbol{\delta}_{u_i})_{\mathsf{P}_i \in \mathcal{H}} + \mathbf{H}^{C}_{n,t}(u_i + \boldsymbol{\delta}_{u_i})_{\mathsf{P}_i \in C} \\ &= (\mathsf{kmac}_1, \ldots, \mathsf{kmac}_n) + \mathbf{H}^{\mathcal{H}}_{n,n-t}(\boldsymbol{\delta}_{u_i})_{\mathsf{P}_i \in \mathcal{H}} + \mathbf{H}^{C}_{n,t}(\boldsymbol{\delta}_{u_i})_{\mathsf{P}_i \in C}\end{aligned}$$

  When parties don't abort, we have $\boldsymbol{\delta}_{u_i} = \mathbf{0}$ for $\mathsf{P}_i \in \mathcal{H}$ and $\boldsymbol{\delta}_i = \mathbf{0}$ for $i \in [n - t + 1, n]$. From the property of hyper-invertible matrices, it follows that given any t outputs and $n - t$ inputs, one can linearly compute the remaining $n - t$ outputs and t inputs. If no party sent an abort, we have $\boldsymbol{\delta}_i = \mathbf{0}$ for a subset of $i \in [1, 2t]$ of size at least t, corresponding to the shares checked by honest parties $\mathsf{P}_i \in \mathcal{H}$. Moreover, $\boldsymbol{\delta}_{u_i} = \mathbf{0}$ for each $\mathsf{P}_i \in \mathcal{H}$. Thus, since t outputs and $n - t$ inputs are known to be zero vectors, it follows that the remaining $n - t$ outputs and t inputs are also zero vectors.

  Moreover, hyper-invertible matrices also guarantee that given t inputs, there is a bijection between the $n - t$ inputs and any $n - t$ outputs. $\mathcal{A}$'s view consits of the t input $\overline{\mathbf{u}}_j$ for each $\mathsf{P}_j \in C$. Since the honest parties share uniformly random secrets, it follows that $n - t$ outputs are uniformly random. $\mathcal{A}$ learns $\overline{\mathsf{kmac}}_i$ for a subset of $i \in [1, 2t]$ of size at most t. The remaining $n - 2t = 2\ell$ output secrets are uniformly random. In the ideal world, $\mathcal{F}_{\mathsf{rand}}$ samples $\mathsf{kmac}_i \leftarrow \mathbb{F}$ uniformly at random and secret shares $\mathsf{kmac}_i \cdot \mathbf{1}$ for every $i \in [1, 2\ell]$. Thus, the secrets are identically distributed in the real and ideal worlds.

- Note that $\mathcal{A}$'s view completely decides the shares of the corrupt parties in $[\overline{\mathbf{u}}_i]$ for each $i \in [1, n]$ since it consists of the shares of corrupt parties in sharings dealt by honest parties and shares of honest parties in sharings dealt by corrupt parties. Specifically, the latter suffices to compute the whole sharing. Moreover, Sim honestly follows the steps of the protocol using the shares of the corrupt parties. It thus follows that the shares of corrupt parties are identical in the real and ideal worlds.

- Using a similar argument to that used for showing that $\boldsymbol{\delta}_i = \mathbf{0}$ when no party P_i aborts for each $i \in [1, 2t]$, it can be shown that $\boldsymbol{\Delta}_i = \mathbf{0}$ for each $i \in [2t + 1, n]$ when no party aborts. In the ideal world, $\mathcal{F}_{\mathsf{mac\text{-}keygen}}$ does not introduce any additive errors to the shares of honest parties. Thus, the additive errors to shares of honest parties is identical and $\mathbf{0}$ in both worlds.

- It follows from the contrapositive of the argument made above that if $\boldsymbol{\delta}_{u_i} \neq 0$ for some $\mathsf{P}_i \in C$ then there exists $\boldsymbol{\delta}_i \neq 0$ for a subset $i \in [1, 2t]$ of size at least $t + 1$, as shown above. A similar argument holds for additive errors to shares of honest parties too. Thus it follows that if the check fails for an honest party in the real world then the simulator sends `abort` to $\mathcal{F}_{\mathsf{mac\text{-}keygen}}$.

The degree-d sharing $[\mathsf{kmac}_i]$ distributed by $\mathcal{F}_{\mathsf{mac\text{-}keygen}}$ is completely determined by the secret kmac_i, t shares of the corrupt parties and the additive errors to the shares of honest parties in $\mathcal{H}_C$. Since the latter are identically distributed in the real and ideal worlds, it follows that the shares of the honest parties are identically distributed in the real and ideal worlds.

Thus, the joint distribution consisting of the view of $\mathcal{A}$ and the output of honest parties is identically distributed in the real and ideal worlds which proves the security of $\Pi_{\mathsf{mac\text{-}keygen}}$. □

Protocol 11: $\Pi_{\mathsf{auth}}([\mathbf{x}], \{[\mathsf{kmac}_i]\}_{i=1}^{L_{\mathsf{mac}}}, \mathcal{S}_{\mathsf{cons}}, \mathcal{S}_{\mathsf{mac}})$

1. Parties invoke $\mathcal{F}_{\mathsf{mult}}$ on inputs $[\mathsf{kmac}_i]$ and $[\mathbf{x}]$ to receive the sharing $[\mathsf{kmac}_i \cdot \mathbf{x}]$ for each $i \in [1, L_{\mathsf{mac}}]$.
2. Parties set $\mathcal{S}_{\mathsf{cons}} := \mathcal{S}_{\mathsf{cons}} \cup \{[\mathsf{kmac}_i \cdot \mathbf{x}]\}_{i=1}^{L_{\mathsf{mac}}}$.
3. Parties compute $\llbracket \mathbf{x} \rrbracket = ([\mathbf{x}], \{[\mathsf{kmac}_i \cdot \mathbf{x}]\}_{i=1}^{L_{\mathsf{mac}}})$ and update $\mathcal{S}_{\mathsf{mac}} := \mathcal{S}_{\mathsf{mac}} \cup \{\llbracket \mathbf{x} \rrbracket\}$.
4. Parties output $\llbracket \mathbf{x} \rrbracket$.

Protocol 12: $\Pi_{\mathsf{check\text{-}bit}}(\llbracket \mathbf{b} \rrbracket, \mathcal{S}_{\mathsf{cons}}, \mathcal{S}_{\mathsf{mac}}, \mathcal{S}_{\mathsf{zero}})$

1. Parties run $\Pi_{\mathsf{auth\text{-}mult}}(\llbracket \mathbf{b} \rrbracket, \llbracket \mathbf{b} \rrbracket, \mathcal{S}_{\mathsf{cons}}, \mathcal{S}_{\mathsf{mac}})$ to obtain the authenticated sharing of the product $\llbracket \mathbf{b}^2 \rrbracket$.
2. Parties locally compute $\llbracket \mathbf{o} \rrbracket = \llbracket \mathbf{b}^2 \rrbracket - \llbracket \mathbf{b} \rrbracket$.
3. Parties set $\mathcal{S}_{\mathsf{zero}} := \mathcal{S}_{\mathsf{zero}} \cup \{[\mathbf{o}]\}$.

5.8 Authenticate Sharing

The protocol Π_{auth} presented in Protocol 11, takes a degree-d packed Shamir sharing $[\mathbf{x}]$ and shares of the MAC keys as input and outputs an authenticated packed sharing $\llbracket \mathbf{x} \rrbracket$ by securely multiplying the secret with each MAC key.

Complexity Analysis. The communication complexity of the protocol is $O(nL_{\mathsf{mac}})$, the computation complexity is $O(n^2 L_{\mathsf{mac}})$, and the round complexity is 2 assuming the random shares required for multiplication have been generated beforehand.

5.9 Verify Bit Vector Sharings

The protocol $\Pi_{\mathsf{check\text{-}bit}}$ presented in Protocol 12, is used to check whether the underlying secret of the input authenticated packed sharing is a vector of bits. This is done by checking if $\mathbf{b} \cdot \mathbf{b} - \mathbf{b} = \mathbf{0}$ which is true iff $\mathbf{b} \in \{0, 1\}^{\ell}$. Note that, $\Pi_{\mathsf{check\text{-}bit}}$ merely updates the appropriate sets with the sharing to be verified and the actual check for zero is run later in the main MPC protocol Π_{mpc} (Protocol 16). The communication, computation and round complexity of $\Pi_{\mathsf{check\text{-}bit}}$ is identical to the costs of a single run of $\Pi_{\mathsf{auth\text{-}mult}}$.

5.10 Verify Consistency of Sharing

Most of the sub-protocols discussed previously are semi-honest secure but guarantee perfect privacy against a malicious adversary. One class of attacks that the adversary can carry out to undermine the correctness of the protocol is to distribute *inconsistent* degree-d packed sharings where the shares of all honest parties don't lie on a polynomial of degree d. The protocol $\Pi_{\mathsf{check\text{-}cons}}$ (Protocol 13) adopted from [GPS22], ensures the consistency of the input sharings while preserving the privacy of the underlying secrets. The protocol computes a random linear combination of the shares to be verified, masks it with a random sharing and then publicly reconstructs the sharing. Intuitively, the random linear combination helps "compress" the inputs while ensuring that if any input sharing is inconsistent then the output sharing is also inconsistent, with probability $1 - |\mathbb{F}|^{-1}$. Since we use a field $\mathbb{F}$ of order independent of the security parameter, we repeat the check $L_{\mathsf{check}} \geq \frac{\kappa_s}{\log_2 |\mathbb{F}|}$ number of times to amplify the probability and ensure that, the probability of an adversary successfully affecting correctness without getting detected is negligible.

Protocol 13: $\Pi_{\text{check-cons}}(\mathcal{S}_{\text{cons}})$

1. Let $m = |\mathcal{S}_{\text{cons}}|$ and $\mathcal{S}_{\text{cons}} = \{[\mathbf{x}_i]\}_{i=1}^{m}$. For each $i \in [1, L_{\text{check}}]$, parties do the following in parallel
   (a) Invoke $\mathcal{F}_{\text{rand}}$ to receive the random sharing $[\mathbf{r}_i]$.
   (b) Invoke $\mathcal{F}_{\text{coin}}$ $m+1$ times to receive the random values $\left\{\alpha_1^i, \ldots, \alpha_{m+1}^i\right\}$.
   (c) Locally compute $[\mathbf{y}_i] = \sum_{j=1}^{m} \alpha_j^i [\mathbf{x}_j] + \alpha_{m+1}^i [\mathbf{r}_i]$.
2. Parties send their shares in $[\mathbf{y}_i]$ to every other party and receive the whole sharing $[\mathbf{y}_i]$ for each $i \in [1, L_{\text{check}}]$. Each $\mathsf{P}_j \in \mathcal{P}$ checks if the shares in $[\mathbf{y}_i]$ form a valid degree-d packed Shamir sharing for each $i \in [1, L_{\text{check}}]$. If the check passes for every $i \in [1, L_{\text{check}}]$, P_j accepts else it aborts.

Complexity Analysis. $\Pi_{\text{check-cons}}$ has a communication complexity of $O(mnL_{\text{check}} + n^2 L_{\text{check}})$ field elements, a computation complexity of $O(mnL_{\text{check}} + n^2 L_{\text{check}})$, and a round complexity of 2 assuming random sharings have been generated beforehand, where m is the number of packed sharings input to the protocol. Using PRFs to instantiate $\mathcal{F}_{\text{coin}}$, as discussed in Section 5.2, improves the communication complexity to $O(n^2 L_{\text{check}})$ making it independent of the number of packed sharings verified.

Lemma 8. If there exists $[\mathbf{x}] \in \mathcal{S}_{\text{cons}}$ such that $\mathbf{\Delta}_x \neq \mathbf{0}$ then honest parties abort in $\Pi_{\text{check-cons}}$ with probability at least $1 - |\mathbb{F}|^{-L_{\text{check}}}$.

Proof. From the definition of $\mathcal{F}_{\text{rand}}$, $\left\{\alpha_1^i, \ldots, \alpha_{m+1}^i\right\}$ are uniformly random for each $i \in [1, L_{\text{check}}]$. We compute the probability that the check fails for some arbitrary $i \in [1, L_{\text{check}}]$. For brevity, we use $\mathbf{\Delta}_j$ to denote $\mathbf{\Delta}_{x_j}$ and denote $\mathbf{r}_i$ as $\mathbf{x}_{m+1}$. Let the k-th sharing $[\mathbf{x}_k]$ be inconsistent so that $\mathbf{\Delta}_k \neq \mathbf{0}$ for some $k \in [1, m+1]$. Since parties abort if the received shares do not form a degree-d packed Shamir sharing and honest parties always send their shares to all parties, we require

$$\sum_{j=1}^{m+1} \alpha_j^i \mathbf{\Delta}_j = \mathbf{0}$$

despite $\mathbf{\Delta}_k \neq \mathbf{0}$. This is true if

$$\alpha_k^i = \left(\sum_{\substack{j \in [1,m] \\ j \neq k}} \alpha_j \cdot (\mathbf{\Delta}_j)_{\text{idx}} \right) \cdot (\mathbf{\Delta}_k)_{\text{idx}}^{-1}$$

for all idx $\in [1, \ell]$ such that $(\mathbf{\Delta}_k)_{\text{idx}} \neq 0$. Since α_k^i is uniformly random and sampled after the adversary introduces the additive errors to the shares, this probability is at most $\frac{1}{|\mathbb{F}|}$. Thus, the probability that every iteration fails is at most $|\mathbb{F}|^{-L_{\text{check}}}$ which in turn implies that the honest parties abort with probability at least $1 - |\mathbb{F}|^{-L_{\text{check}}}$ in case one of the sharings is inconsistent. □

5.11 Verify Correctness of Secrets

A malicious adversary can carry out two types of attacks to undermine the correctness of most of the subprotocols discussed above. While consistency of shares distributed by parties is ensured using $\Pi_{\text{check-cons}}$ (see Section 5.10), the protocol $\Pi_{\text{check-mac}}$ (Protocol 14) is used to verify the correctness of computation and ensure that the adversary did not introduce any additive errors to the secret during the computation. The protocol works by first computing a random linear combination over the MAC sharings and the secret sharings to compute $[\mathsf{kmac} \cdot \mathbf{x}]$ and $[\mathbf{x}]$ respectively. As in the case of $\Pi_{\text{check-cons}}$, the random linear combinations "compress" the inputs while ensuring that invalid inputs lead to invalid outputs. The MAC key kmac is then reconstructed and parties verify if $\mathsf{kmac} \cdot [\mathbf{x}] - [\mathsf{kmac} \cdot \mathbf{x}]$ is a sharing of the zero vector. Since the reconstructed MAC key is publicly known, $\mathsf{kmac} \cdot [\mathbf{x}]$ can be robustly computed while the MAC sharings are computed through MPC and can have additive errors to the secrets. However, since the MAC key was private during the computation, the probability that the adversary's attack goes undetected is proportional to $|\mathbb{F}|^{-1}$. Since we use a field $\mathbb{F}$ of order independent of the security parameter, we repeat the check $L_{\mathsf{mac}} \geq \frac{\kappa_s}{\log|\mathbb{F}|-1}$ number of times to amplify the probability and ensure that the probability an adversary successfully affects correctness without getting detected is negligible.

The protocol is similar to the check described in [CGH+18], except that we directly reconstruct $[\mathbf{check}_i]$ instead of using a check-zero functionality. This is not a concern in our protocol because the check is performed over random values generated during the pre-processing phase. In case of any malicious behaviour, parties abort the protocol and so any information learnt by the adversary is only over random values. When the adversary does not introduce any errors, the reconstructed value is $\mathbf{0}$ and hence it does not leak anything about the underlying secrets.

Complexity Analysis. $\Pi_{\text{check-mac}}$ has a communication complexity of $O\left(mnL_{\mathsf{mac}} + n^2 L_{\mathsf{mac}}\right)$ and a computation complexity of $O\left(mnL_{\mathsf{mac}} + n^2 L_{\mathsf{mac}}\right)$ where m is the number of authenticated sharings input to the protocol. Its round complexity is 3 assuming random sharings have been generated beforehand. Using PRFs to instantiate $\mathcal{F}_{\text{coin}}$, as discussed in Section 5.2, improves the communication complexity to $O\left(n^2 L_{\mathsf{mac}}\right)$ making it independent of the number of authenticated sharings verified.

Protocol 14: $\Pi_{\text{check-mac}}(\{[\mathsf{kmac}_i]\}_{i=1}^{L_{\mathsf{mac}}}, \mathcal{S}_{\mathsf{mac}})$

1. Let $m = |\mathcal{S}_{\mathsf{mac}}|$ and $\mathcal{S}_{\mathsf{mac}} = \left\{[\![\mathbf{x}_i]\!]\right\}_{i=1}^{m}$. For each $i \in [1, L_{\mathsf{mac}}]$, parties do the following in parallel
   (a) Invoke $\mathcal{F}_{\text{coin}}$ m times to obtain the random values $\left\{\alpha_1^i, \ldots, \alpha_m^i\right\}$.
   (b) Send their share in $[\mathsf{kmac}_i]$ to all parties, receive the whole sharing $[\mathsf{kmac}_i]$ and reconstruct the secret kmac_i.
   (c) Locally compute
   $$[\mathbf{y}_i] = \sum_{j=1}^{m} \alpha_j^i [\mathbf{x}_j]$$
   $$[\mathsf{kmac}_i \cdot \mathbf{y}_i] = \sum_{j=1}^{m} \alpha_j^i [\mathsf{kmac}_i \cdot \mathbf{x}_j]$$
   and $[\mathbf{check}_i] = [\mathsf{kmac}_i \cdot \mathbf{y}_i] - \mathsf{kmac}_i [\mathbf{y}_i]$.
   (d) Send their share in $[\mathbf{check}_i]$ to all parties and receive the whole sharing $[\mathbf{check}_i]$.
2. Parties abort if for any $i \in [1, L_{\mathsf{check}}]$, the shares in $[\mathbf{check}_i]$ do not form a valid degree-d packed Shamir sharing. Else, if the reconstructed value $\mathbf{check}_i \neq \mathbf{0}$ then parties output reject.

6 Main Protocols

In this section we present the protocols that enable parties to compute and evaluate the garbled circuit of the LPN based garbling scheme in a distributed and secure manner. We first present a subprotocol Π_{error}

for generating authenticated packed sharings of secrets sampled from the LPN error distribution. We then describe our constant round MPC protocol Π_{mpc}.

6.1 Sharing Biased Random Vectors

In this section, we describe a protocol Π_{error} (Protocol 15) that generates authenticated packed sharings of vectors where each element is sampled from the LPN error distribution $\mathsf{Ber}_{\tau_{\mathsf{lpn}}}$, which parties require when computing the garbled circuit in the MPC protocol. The LPN error distribution $\mathsf{Ber}_{\tau_{\mathsf{lpn}}}$ is obtained by sampling a uniformly random element from $\mathbb{F}$ with probability $2^{-\tau_{\mathsf{lpn}}}$ and 0 with probability $1 - 2^{-\tau_{\mathsf{lpn}}}$. To compute a packed sharing of errors, parties first compute a packed sharing of a biased bit vector where each element is 1 with probability $2^{-\tau_{\mathsf{lpn}}}$ by generating and multiplying τ_{lpn} uniformly random bit vectors. Parties then securely multiply the biased bit vector with a random vector to compute the packed sharing of the errors.

Complexity Analysis. The communication cost for generating a packed sharing of the error is $O(nL_{\mathsf{mac}})$ since we use the LPN assumption with constant noise rate where τ_{lpn} is a constant. Similarly, the computation cost for generating a packed sharing of the error is $O\big(n^2 L_{\mathsf{mac}}\big)$. Assuming the random sharings used in the protocol have been generated beforehand, the round complexity of the protocol is $6 + \log \tau_{\mathsf{lpn}}$.

Protocol 15: $\Pi_{\mathsf{error}}(\{[\mathsf{kmac}_i]\}_{i=1}^{L_{\mathsf{mac}}}, \mathcal{S}_{\mathsf{cons}}, \mathcal{S}_{\mathsf{mac}}, \mathcal{S}_{\mathsf{zero}})$

1. Parties invoke $\mathcal{F}_{\mathsf{bitrand}}$ τ_{lpn} times to get random bit vector sharings $\{[\mathbf{b}_i]\}_{i=1}^{\tau_{\mathsf{lpn}}}$.
2. Parties run $\Pi_{\mathsf{auth}}([\mathbf{b}_i], \{[\mathsf{kmac}_i]\}_{i=1}^{L_{\mathsf{mac}}}, \mathcal{S}_{\mathsf{cons}})$ to receive authenticated sharings $\llbracket \mathbf{b}_i \rrbracket$ for each $i \in [1, \tau_{\mathsf{lpn}}]$.
3. Parties run $\Pi_{\mathsf{check\text{-}bit}}(\llbracket \mathbf{b}_i \rrbracket, \mathcal{S}_{\mathsf{cons}}, \mathcal{S}_{\mathsf{mac}}, \mathcal{S}_{\mathsf{zero}})$.
4. Parties compute $\llbracket \mathbf{b} \rrbracket$ where $\boldsymbol{\epsilon} = \Pi_{i=1}^{\tau_{\mathsf{lpn}}} \mathbf{b}_i$ by running instances of $\Pi_{\mathsf{auth\text{-}mult}}$.
5. Parties invoke $\mathcal{F}_{\mathsf{rand}}$ and obtain a random sharing $[\mathbf{r}]$.
6. Parties invoke $\mathcal{F}_{\mathsf{mult}}$ with inputs $[\mathbf{b}]$ and $[\mathbf{r}]$ to obtain the sharing of the product $[\boldsymbol{\epsilon}]$.
7. Parties invoke $\mathcal{F}_{\mathsf{mult}}$ with inputs $[\mathsf{kmac}_i \cdot \mathbf{b}]$ and $[\mathbf{r}]$ to obtain the sharing of the product $[\mathsf{kmac}_i \cdot \boldsymbol{\epsilon}]$ for each $i \in [1, L_{\mathsf{mac}}]$.
8. Parties compute $\llbracket \boldsymbol{\epsilon} \rrbracket = ([\boldsymbol{\epsilon}], \{[\mathsf{kmac}_i \cdot \boldsymbol{\epsilon}]\}_{i=1}^{L_{\mathsf{mac}}})$ and update $\mathcal{S}_{\mathsf{cons}} \coloneqq \mathcal{S}_{\mathsf{cons}} \cup \{[\boldsymbol{\epsilon}]\} \cup \{[\mathsf{kmac}_i \cdot \boldsymbol{\epsilon}]\}_{i=1}^{L_{\mathsf{mac}}}$, and $\mathcal{S}_{\mathsf{mac}} \coloneqq \mathcal{S}_{\mathsf{mac}} \cup \big\{\llbracket \boldsymbol{\epsilon} \rrbracket\big\}$.
9. Parties output $\llbracket \boldsymbol{\epsilon} \rrbracket$.

6.2 Our Constant Round MPC Protocol

In this section, we present our constant-round maliciously secure MPC protocol. This protocol consists of three phases: a circuit independent *pre-processing phase*, where random sharings used for computing the garbled circuit are generated, a *garbling phase* where the garbled circuit is computed in a distributed manner, and the *evaluation phase* where clients provide their inputs and the garbled circuit is reconstructed and evaluated locally. The functionality $\mathcal{F}_{\mathsf{mpc}}$ realized by Π_{mpc} is described in Functionality 8.

In Π_{mpc}, wires and gates are grouped together in blocks of size ℓ for the purpose of packed secret sharing. We use $\mathsf{pos}_w = (\mathsf{slot}^{\mathsf{C}}_{w\ell+1}, \ldots, \mathsf{slot}^{\mathsf{C}}_{2w\ell})$ to denote the positions corresponding to the wires in the w-th block for each $w \in [1, \lceil W/\ell \rceil]$. We use $\mathsf{pos}^{\mathsf{left}}_g = (\mathsf{slot}^{\mathsf{C}}_{\mathsf{left}(g\ell+1)}, \ldots, \mathsf{slot}^{\mathsf{C}}_{\mathsf{left}(2g\ell)})$ to denote the

Functionality 8: $\mathcal{F}_{\mathsf{mpc}}$

1. $\mathcal{F}_{\mathsf{mpc}}$ receives the inputs from all clients. Let $\mathbf{x}$ denote the inputs and let C denote the circuit.
2. $\mathcal{F}_{\mathsf{mpc}}$ computes $\mathbf{y} = \mathsf{C}(\mathbf{x})$ and sends it to Sim and receives status $\in \{\texttt{abort}, \texttt{continue}\}$ from Sim.
   (a) If status = $\texttt{abort}$, $\mathcal{F}_{\mathsf{mpc}}$ sends $\texttt{abort}$ to all clients.
   (b) If status = $\texttt{continue}$, $\mathcal{F}_{\mathsf{mpc}}$ sends $\mathbf{y}$ to all clients.

set of positions for the left input wires for gates in the g-th block and $\mathsf{pos}_g^{\mathsf{right}}$, and $\mathsf{pos}_g^{\mathsf{out}}$ are defined similarly for positions corresponding to right input wires and output wires of gates in the g-th block for each $g \in [1, \lceil G/\ell \rceil]$. We use $\mathsf{left}'(g) = \lfloor \mathsf{left}(g)/\ell \rfloor$ (and similarly $\mathsf{right}'(g)$) to denote the index of the block containing the secret corresponding to the g-th gate. When garbling a gate, we require that the masks and keys for the left, right and output wires of gates in the g-th block are packed together in the same sharings for every $g \in [1, \lceil G/\ell \rceil]$. As discussed in Section 3, the select algorithm can be used to select secrets across different sharings followed by running Π_{trans} (Protocol 8) to have the secrets shared over the required positions. However, when the same wire is input to multiple gates in the same block, the corresponding secret needs to be repeated appropriately during Π_{trans}. We use f_g^{left} (and similarly f_g^{right}) to denote the function which takes a vector containing the selected secrets corresponding to the left (right) input wires for gates in the g-th block as input and maps it to a vector where elements are repeated appropriately to handle the above case when the same wire is used as the left (right) input wire for multiple gates in the g-th block. $f_I(\mathbf{x}) = \mathbf{x}$ is used to denote the identity function. Finally, in the description of our protocol we assume that if a party aborts at any point in the protocol, it broadcasts an $\texttt{abort}$ message and any party that receives such a message aborts too, ensuring unanimous abort.

Circuit Independent Pre-processing Phase. In this phase, parties generate the shares of the MAC keys, the labels and masks for each wire, and the errors used for computing ciphertexts using the LPN based encryption scheme. Note that step 2 in Π_{mpc} requires 3 rounds, steps 3, 4, and 5 can be run in parallel and the random packed sharings and random bit vector sharings used for the entire protocol can be generated in parallel in the first 2 rounds of the protocol. Thus the round complexity of this phase is $9+\log \tau_{\mathsf{lpn}}$ rounds. The communication complexity of this phase is $O\left(nL_{\mathsf{mac}} + nL_{\mathsf{lpn}}L_{\mathsf{mac}}\frac{W}{\ell} + nL_{\mathsf{mac}}Q_{\mathsf{lpn}}\frac{G}{\ell}\right) = O(|\mathsf{C}|\kappa_s\kappa_c + n\kappa_s)$ field elements. Similarly, the computation complexity of this phase is $O(n|\mathsf{C}|\kappa_s\kappa_c)$.

Garbling Phase. In this phase, parties compute the garbled circuit by encrypting the permuted labels on the output wire of each gate using the labels on the input wires. This requires first transforming the masks and labels generated during the pre-processing phase to ensure that the masks and keys for the left, right and output wires of gates in the same block are packed together and then correctly permuting the labels on the output wires. Note that permuting the labels on the output wire involves evaluating the gate function on the masks for the input wires of the gate. Since AND and XOR gates might be in the same block, we compute a label assuming all gates in the block are AND gates or XOR gates, and then select the correct label for each gate in the block. Once the labels on the output wire have been permuted, encryption is straightforward using the multiplicative friendliness of packed Shamir sharing (Section 3.2). Parties also prepare sharings of masks on circuit input and output wires to communicate inputs and outputs with clients in this phase. Finally, parties verify against malicious behavior of corrupt parties and ensure that the garbled circuit is computed correctly with overwhelming probability. Note that steps 6.a and 6.b can be run in parallel, steps 6.c and 6.d can be run in parallel, and steps 8.b, 9a, and 9.c can be run in parallel in this phase. Thus, the round complexity of this phase is 19 rounds. The communication complexity of this phase is $O\left(|\mathsf{C}|\kappa_s\kappa_c + n^2\kappa_s\right)$ field elements while the computation complexity is $O(n|\mathsf{C}|\kappa_s\kappa_c)$.

Protocol 16: Π_{mpc}

Circuit Independent Pre-processing Phase

1. Parties set $\mathcal{S}_{\mathsf{cons}} = \mathcal{S}_{\mathsf{mac}} = \mathcal{S}_{\mathsf{zero}} = \emptyset$.
2. **Generate MAC Keys:** Parties invoke $\mathcal{F}_{\mathsf{mac\text{-}keygen}}$ to obtain the sharing $[\mathsf{kmac}_i]$ for each $i \in [1, L_{\mathsf{mac}}]$. Let $\mathcal{K}_{\mathsf{mac}} = \{[\mathsf{kmac}_i]\}_{i=1}^{L_{\mathsf{mac}}}$.
3. **Generate Wire Labels:** For each $i \in [1, L_{\mathsf{lpn}}]$, $b \in \{0, 1\}$, and $w \in [1, W/\ell]$, parties do the following
   (a) Invoke $\mathcal{F}_{\mathsf{rand}}$ to obtain the random sharing $[\mathbf{k}_{w,i}^b]$.
   (b) Set $\mathcal{S}_{\mathsf{cons}} := \mathcal{S}_{\mathsf{cons}} \cup \{[\mathbf{k}_{w,i}^b]\}$.
   (c) Run $\Pi_{\mathsf{auth}}([\mathbf{k}_{w,i}^b], \mathcal{K}_{\mathsf{mac}}, \mathcal{S}_{\mathsf{cons}}, \mathcal{S}_{\mathsf{mac}})$ to obtain the authenticated sharing $[\![\mathbf{k}_{w,i}^b]\!]$.
4. **Generate Wire Masks:** For each $w \in [1, W/\ell]$, parties do the following
   (a) Invoke $\mathcal{F}_{\mathsf{bitrand}}$ to obtain the random bit vector sharing $[\boldsymbol{\lambda}_w]$.
   (b) Set $\mathcal{S}_{\mathsf{cons}} := \mathcal{S}_{\mathsf{cons}} \cup \{[\boldsymbol{\lambda}_w]\}$.
   (c) Run $\Pi_{\mathsf{auth}}([\boldsymbol{\lambda}_w], \mathcal{K}_{\mathsf{mac}}, \mathcal{S}_{\mathsf{cons}}, \mathcal{S}_{\mathsf{mac}})$ to obtain the authenticated sharing $[\![\boldsymbol{\lambda}_w]\!]$.
   (d) Run $\Pi_{\mathsf{check\text{-}bit}}([\![\boldsymbol{\lambda}_w]\!], \mathcal{S}_{\mathsf{cons}}, \mathcal{S}_{\mathsf{mac}}, \mathcal{S}_{\mathsf{zero}})$.
5. **Generate LPN Errors:** Parties run $\Pi_{\mathsf{error}}(\mathcal{K}_{\mathsf{mac}}, \mathcal{S}_{\mathsf{cons}}, \mathcal{S}_{\mathsf{mac}}, \mathcal{S}_{\mathsf{zero}})$ to obtain the sharing of the LPN error $[\![\epsilon_{g,i}^{\alpha,\beta}]\!]$ for each $\alpha, \beta \in \{0, 1\}$, $i \in [1, Q_{\mathsf{lpn}}]$, and $g \in [1, G/\ell]$.

Evaluation Phase. In this phase, parties first reconstruct the mask on the input wires towards the clients providing the inputs and the clients broadcast the masked input. Parties then reconstruct the appropriate labels for the circuit input wires, the garbled circuit, and the masks on the circuit output wires. Parties can then evaluate the garbled circuit locally. Note that once the masked inputs have been provided by clients, the shares for all secrets to be reconstructed can be broadcast in 1 round. Thus, the round complexity of this phase is 3 rounds. This phase involves a broadcast of $O\big(nQ_{\mathsf{lpn}}\frac{G}{\ell}\big) = O(|\mathsf{C}|\kappa_c)$ field elements and has a computation complexity of $O(n|\mathsf{C}|\kappa_c)$.

Thus, the overall round complexity of Π_{mpc} is $31 + \log \tau_{\mathsf{lpn}}$ rounds with a total communication complexity of $O\big(|\mathsf{C}|\kappa_s\kappa_c + n^2\kappa_s\big)$ field elements over point to point channels and $O(|\mathsf{C}|\kappa_c)$ field elements over the broadcast channel. The total computation complexity of the protocol is $O(n|\mathsf{C}|\kappa_c)$.

Lemma 9. Let $(\mathcal{G}, \mathbf{X}, \mathsf{aux}_{\mathsf{dec}})$ be reconstructed in Π_{mpc} (step 13) when run with input $\mathbf{v}$ for the circuit C. Let $(\mathcal{G}', \mathbf{X}', \mathsf{aux}'_{\mathsf{dec}})$ be such that $(\mathcal{G}', \mathsf{aux}'_{\mathsf{enc}}, \mathsf{aux}'_{\mathsf{dec}}) \leftarrow \mathsf{GC.Garble}(\mathsf{C}, \mathsf{GC.MaskGen}(\mathsf{C}, 1^{\kappa_c}), 1^{\kappa_c})$ and $\mathbf{X}' \leftarrow \mathsf{GC.Encode}(\mathbf{v}, \mathsf{aux}'_{\mathsf{enc}})$. If all parties and clients behave honestly, $(\mathcal{G}, \mathbf{X}, \mathsf{aux}_{\mathsf{dec}})$ is identically distributed to $(\mathcal{G}', \mathbf{X}', \mathsf{aux}'_{\mathsf{dec}})$.

Proof Sketch. As in the garbling scheme, the keys $(k_{w,i}^b)_{i=1}^{L_{\mathsf{lpn}}}$ sampled in Π_{mpc} for each wire w and $b \in \{0, 1\}$ are uniformly random in $\mathbb{F}^{L_{\mathsf{lpn}}}$ from the definition of $\mathcal{F}_{\mathsf{rand}}$ while λ_w for every wire w is uniformly random in $\{0, 1\}$ from the definition of $\mathcal{F}_{\mathsf{bitrand}}$. Similarly, Π_{error} outputs sharings where the underlying secret ϵ is sampled from $\mathsf{Ber}_{\tau_{\mathsf{lpn}}}$ since by multiplying τ_{lpn} uniformly random bits it biases the secret to 0 with probability $1 - 2^{-\tau_{\mathsf{lpn}}}$ and a uniformly random value in $\mathbb{F}$ with probability $2^{-\tau_{\mathsf{lpn}}}$. In the garbling phase, parties essentially compute $\mathbf{ctx}_g^{\alpha,\beta} = \mathsf{LPN.Enc}(\mathbf{k}_w^b \| b, \mathbf{k}_{\mathsf{left}(g)}^{\alpha} + \mathbf{k}_{\mathsf{right}(g)}^{\beta})$ in secret shared form where $b =$

Protocol 16: Π_{mpc} (continued)

Garbling Phase

6. **Transform Masks and Labels:**

   (a) Parties run $\Pi_{\text{auth-trans}}(\llbracket \boldsymbol{\lambda}_w \rrbracket, \text{pos}_w, f_I, \mathcal{S}_{\text{cons}})$ to obtain the transformed sharing $\llbracket \boldsymbol{\lambda}_w | \text{pos}_w \rrbracket$ for each $w \in [1, W/\ell]$.

   (b) Parties run $\Pi_{\text{auth-trans}}(\llbracket \mathbf{k}_{w,i}^b \rrbracket, \text{pos}_w, f_I, \mathcal{S}_{\text{cons}})$ to obtain the transformed sharing $\llbracket \mathbf{k}_{w,i}^b | \text{pos}_w \rrbracket$ for each $w \in [1, W/\ell]$, $i \in [1, L_{\text{lpn}}]$, and $b \in \{0, 1\}$.

   (c) For each $g \in [1, G/\ell]$, parties locally compute:

   $$\llbracket \boldsymbol{\lambda}_g^{\text{left}} | \text{pos}_g^{\text{left}} \rrbracket_{d+\ell-1} = \text{select}(\left\{\llbracket \boldsymbol{\lambda}_{\text{left}'(g\ell+i)} | \text{pos}_{\text{left}'(g\ell+i)} \rrbracket\right\}_{i=1}^{\ell}, \text{pos}_g^{\text{left}})$$

   and run $\Pi_{\text{auth-trans}}(\llbracket \boldsymbol{\lambda}_g^{\text{left}} | \text{pos}_g^{\text{left}} \rrbracket_{d+\ell-1}, \text{pos}_{\text{def}}, f_g^{\text{left}}, \mathcal{S}_{\text{cons}})$ to obtain the transformed sharing $\llbracket \boldsymbol{\lambda}_g^{\text{left}} \rrbracket$ corresponding to the masks on the left input wires. Parties similarly compute $\llbracket \boldsymbol{\lambda}_g^{\text{right}} \rrbracket$ and $\llbracket \boldsymbol{\lambda}_g^{\text{out}} \rrbracket$ corresponding to the masks on the right input wires and output wires. Parties set $\mathcal{S}_{\text{mac}} := \mathcal{S}_{\text{mac}} \cup \left\{\llbracket \boldsymbol{\lambda}_g^{\text{left}} \rrbracket, \llbracket \boldsymbol{\lambda}_g^{\text{right}} \rrbracket, \llbracket \boldsymbol{\lambda}_g^{\text{out}} \rrbracket\right\}$.

   (d) For each $g \in [1, G/\ell]$, $i \in [1, L_{\text{lpn}}]$, and $b \in \{0, 1\}$, parties locally compute:

   $$\llbracket \mathbf{leftk}_{g,i}^b | \text{pos}_g^{\text{left}} \rrbracket_{d+\ell-1} = \text{select}(\left\{\llbracket \mathbf{k}_{\text{left}'(g\ell+j),i}^b | \text{pos}_{\text{left}'(g\ell+j)} \rrbracket\right\}_{j=1}^{\ell}, \text{pos}_g^{\text{left}})$$

   and run $\Pi_{\text{auth-trans}}(\llbracket \mathbf{leftk}_{g,i}^b | \text{pos}_g^{\text{left}} \rrbracket_{d+\ell-1}, \text{pos}_{\text{def}}, f_g^{\text{left}}, \mathcal{S}_{\text{cons}})$ to obtain the transformed sharing $\llbracket \mathbf{leftk}_{g,i}^b \rrbracket$ corresponding to the labels for the left input wires. Parties similarly compute $\llbracket \mathbf{rightk}_{g,i}^b \rrbracket$ and $\llbracket \mathbf{outk}_{g,i}^b \rrbracket$ corresponding to the labels for the right input wires and output wires. Parties set $\mathcal{S}_{\text{mac}} := \mathcal{S}_{\text{mac}} \cup \left\{\llbracket \mathbf{leftk}_{g,i}^b \rrbracket, \llbracket \mathbf{rightk}_{g,i}^b \rrbracket, \llbracket \mathbf{outk}_{g,i}^b \rrbracket\right\}$.

7. **Select Plaintexts:** Let $\mathbf{type}_g \in \mathbb{F}^\ell$ be such that $(\mathbf{type}_g)_i = 0$ if $(g\ell + i)$-th gate is a XOR gate else $(\mathbf{type}_g)_i = 1$. For each $\alpha, \beta \in \{0, 1\}$, and $g \in [1, G/\ell]$, parties do the following

   (a) Run $\Pi_{\text{auth-mult}}(\llbracket \boldsymbol{\lambda}_g^{\text{left}} \rrbracket + \alpha, \llbracket \boldsymbol{\lambda}_g^{\text{right}} + \beta \rrbracket, \mathcal{S}_{\text{cons}}, \mathcal{S}_{\text{mac}})$ and add $\llbracket \boldsymbol{\lambda}_g^{\text{out}} \rrbracket$ to the obtained authenticated product sharing to compute $\llbracket \mathbf{s\text{-}and}_g^{\alpha,\beta} \rrbracket$.

   (b) Locally compute $\llbracket \mathbf{s\text{-}xor}_g^{\alpha,\beta} \rrbracket = \llbracket \boldsymbol{\lambda}_g^{\text{left}} \rrbracket + \llbracket \boldsymbol{\lambda}_g^{\text{right}} \rrbracket + \llbracket \boldsymbol{\lambda}_g^{\text{out}} \rrbracket + \alpha + \beta$.

   (c) Locally compute $\llbracket \mathbf{s}_g^{\alpha,\beta} \rrbracket_{\mathsf{d}+\ell-1} = \mathbf{type}_g \cdot (\llbracket \mathbf{s\text{-}and}_g^{\alpha,\beta} \rrbracket - \llbracket \mathbf{s\text{-}xor}_g^{\alpha,\beta} \rrbracket) + \llbracket \mathbf{s\text{-}xor}_g^{\alpha,\beta} \rrbracket$.

   (d) Run $\Pi_{\text{auth-trans}}(\llbracket \mathbf{s}_g^{\alpha,\beta} \rrbracket_{\mathsf{d}+\ell-1}, \text{pos}_{\text{def}}, f_I, \mathcal{S}_{\text{cons}})$ to obtain the transformed sharing $\llbracket \mathbf{s}_g^{\alpha,\beta} \rrbracket$ and set $\mathcal{S}_{\text{mac}} := \mathcal{S}_{\text{mac}} \cup \left\{\llbracket \mathbf{s}_g^{\alpha,\beta} \rrbracket\right\}$.

   (e) Run $\Pi_{\text{auth-mult}}(\llbracket \mathbf{s}_g^{\alpha,\beta} \rrbracket, (\llbracket \mathbf{outk}_{g,i}^1 \rrbracket - \llbracket \mathbf{outk}_{g,i}^0 \rrbracket), \mathcal{S}_{\text{cons}}, \mathcal{S}_{\text{mac}})$ and add $\llbracket \mathbf{outk}_{g,i}^0 \rrbracket$ to the obtained authenticated product sharing to compute $\llbracket \mathbf{outk}_{g,i}^{\alpha,\beta} \rrbracket$ for each $i \in [1, L_{\text{lpn}}]$.

8. **Compute Ciphertexts:** For each $\alpha, \beta \in \{0, 1\}$, $g \in [1, G/\ell]$, parties first compute

   $$\left(\llbracket \mathbf{mssg}_{g,i}^{\alpha,\beta} \rrbracket\right)_{i=1}^{Q_{\text{ecc}}} = \mathsf{ECC.Enc}\left(\left(\llbracket \mathbf{outk}_{g,i}^{\alpha,\beta} \rrbracket\right)_{i=1}^{L_{\text{lpn}}}, \llbracket \mathbf{s}_g^{\alpha,\beta} \rrbracket\right)$$

   and then do the following for each $i \in [1, Q_{\text{ecc}}]$

(a) Compute

$$[\![\mathbf{ctx}^{\alpha,\beta}_{g,i}]\!]_{\mathsf{d}+\ell-1} = \sum_{i=1}^{L_{\mathsf{lpn}}} \mathrm{A}^{\alpha,\beta}_{g,i}([\![\mathbf{leftk}^{\alpha}_{g,i}]\!] + [\![\mathbf{rightk}^{\beta}_{g,i}]\!]) + [\![\boldsymbol{\epsilon}^{\alpha,\beta}_{g,i}]\!] + [\![\mathbf{mssg}^{\alpha,\beta}_{g,i}]\!].$$

(b) Run $\Pi_{\mathsf{auth\text{-}trans}}([\![\mathbf{ctx}^{\alpha,\beta}_{g,i}]\!]_{\mathsf{d}+\ell-1}, \mathsf{pos}_{\mathsf{def}}, f_I, \mathcal{S}_{\mathsf{cons}})$ to obtain $[\![\mathbf{ctx}^{\alpha,\beta}_{g,i}]\!]$.

(c) Set $\mathcal{S}_{\mathsf{mac}} := \mathcal{S}_{\mathsf{mac}} \cup \left\{[\![\mathbf{ctx}^{\alpha,\beta}_{g,i}]\!]\right\}_{i=1}^{Q_{\mathsf{ecc}}}$.

9. **Transform Shares for Input and Output:**

(a) For io $\in \{\mathsf{inp}, \mathsf{out}\}$, parties compute

$$[\![\lambda^{\mathsf{io}}_{w} \,|\mathsf{pos}_{w'}]\!]_{\mathsf{d}+\ell-1} = \mathsf{select}\left([\![\boldsymbol{\lambda}_{w'} \,|\mathsf{pos}_{w'}]\!], \left\{\mathsf{slot}^{\mathsf{C}}_{w}\right\}\right)$$

and run $\Pi_{\mathsf{auth\text{-}trans}}([\![\lambda^{\mathsf{io}}_{w} \,|\mathsf{pos}_{w'}]\!]_{\mathsf{d}+\ell-1}, \mathsf{slot}^{\mathsf{def}}_{1}, f_I, \mathcal{S}_{\mathsf{cons}})$ to receive $[\![\lambda^{\mathsf{io}}_{w}]\!]$ for each $w \in [1, W_{\mathsf{inp}}]$ if io = inp or $w \in \mathcal{W}_{\mathsf{out}}$ if io = out, and $w' \in \lfloor w/\ell \rfloor$.

(b) Parties update $\mathcal{S}_{\mathsf{mac}} := \mathcal{S}_{\mathsf{mac}} \cup \left\{[\![\lambda^{\mathsf{inp}}_{w}]\!]\right\}_{w=1}^{W_{\mathsf{inp}}} \cup \left\{[\![\lambda^{\mathsf{out}}_{w}]\!]\right\}_{w \in \mathcal{W}_{\mathsf{out}}}$.

(c) For each $i \in [1, L_{\mathsf{lpn}}]$, $b \in \{0, 1\}$, $w \in [1, W_{\mathsf{inp}}]$, parties compute

$$[\![\mathsf{inpk}^{b}_{w,i} \,|\mathsf{pos}_{w'}]\!]_{\mathsf{d}+\ell-1} = \mathsf{select}([\![\mathbf{k}^{b}_{w,i}]\!], \left\{\mathsf{slot}^{\mathsf{C}}_{w}\right\})$$

and run $\Pi_{\mathsf{auth\text{-}trans}}([\![\mathsf{inpk}^{b}_{w,i} \,|\mathsf{pos}_{w'}]\!]_{\mathsf{d}+\ell-1}, \mathsf{slot}^{\mathsf{def}}_{1}, f_I, \mathcal{S}_{\mathsf{cons}})$ to receive $[\![\mathsf{inpk}^{b}_{w,i}]\!]$.

(d) Parties update $\mathcal{S}_{\mathsf{mac}} := \mathcal{S}_{\mathsf{mac}} \cup \left\{[\![\mathsf{inpk}^{b}_{w,i}]\!]\right\}$ for each $w \in [1, W_{\mathsf{inp}}]$, $b \in \{0, 1\}$, and $i \in [1, L_{\mathsf{lpn}}]$.

10. **Verification:**

(a) Parties run $\Pi_{\mathsf{check\text{-}cons}}(\mathcal{S}_{\mathsf{cons}})$.

(b) Parties run $\Pi_{\mathsf{check\text{-}mac}}(\mathcal{K}_{\mathsf{mac}}, \mathcal{S}_{\mathsf{mac}})$.

(c) Let $|\mathcal{S}_{\mathsf{zero}}| = m$ and let $\mathcal{S}_{\mathsf{zero}} = \{[\mathbf{o}_1], \ldots, [\mathbf{o}_m]\}$. For each $i \in [1, L_{\mathsf{check}}]$, parties do the following

- Invoke $\mathcal{F}_{\mathsf{coin}}$ m times to obtain $\left\{\alpha^{i}_{j}\right\}_{j=1}^{m}$.
- Compute $[\mathbf{check}_i] = \sum_{j=1}^{m} \alpha^{i}_{j} [\mathbf{x}_j]$.
- Send their share in $[\mathbf{check}_i]$ to all parties to learn the whole sharing $[\mathbf{check}_i]$.

Parties abort if for any $i \in [1, L_{\mathsf{check}}]$, the shares in $[\mathbf{check}_i]$ do not form a valid degree-d packed Shamir sharing or if the reconstructed value $\mathbf{check}_i \neq \mathbf{0}$.

Protocol 16: Π_{mpc} (continued)

Evaluation Phase

11. **Share Inputs:** For each $w \in [1, W_{\mathsf{inp}}]$
    (a) Parties send their shares in $[\lambda_w^{\mathsf{inp}}]$ to $\mathsf{P}^{\mathsf{c}}_{\mathsf{client}(w)}$.
    (b) $\mathsf{P}^{\mathsf{c}}_{\mathsf{client}(w)}$ aborts if the received shares do not form a valid degree-d packed Shamir sharing. Else, it broadcasts $\rho_w = \lambda_w^{\mathsf{inp}} + v_w$.
    (c) Parties receive ρ_w and broadcast their share in $[\mathsf{inpk}_{w,i}^{\rho_w}]$ for every $i \in [1, L_{\mathsf{lpn}}]$.
    (d) Parties receive the whole sharing $[\mathsf{inpk}_{w,i}^{\rho_w}]$ and each $\mathsf{P}_i \in \mathcal{P}$ checks and aborts if the shares do not form a valid degree-d packed Shamir sharing for every $i \in [1, L_{\mathsf{lpn}}]$. Else, P_i reconstructs the secret $\mathsf{inpk}_{w,i}^{\rho_w}$ for every $i \in [1, L_{\mathsf{lpn}}]$. Let $\mathbf{inpk}_w^{\rho_w} = (\mathsf{inpk}_{w,1}^{\rho_w}, \dots, \mathsf{inpk}_{w,L_{\mathsf{lpn}}}^{\rho_w})$.
12. **Reconstruct Garbled Circuit:** Parties do the following
    (a) Broadcast their share in $[\boldsymbol{\lambda}_w^{\mathsf{out}}]$ for each $w \in \mathcal{W}_{\mathsf{out}}$ and their share in $[\mathbf{ctx}_{g,i}^{\alpha,\beta}]$ for each $\alpha, \beta \in \{0,1\}$, $g \in [1, G/\ell]$, and $i \in [1, Q_{\mathsf{lpn}}]$.
    (b) Each $\mathsf{P}_i \in \mathcal{P}$ checks if every sharing received in the previous step forms a valid degree-d packed Shamir sharing. If the check passes for every sharing then P_i reconstructs the secrets $[\boldsymbol{\lambda}_w^{\mathsf{out}}]$ for each $w \in \mathcal{W}_{\mathsf{out}}$, and $\mathbf{ctx}_{g,i}^{\alpha,\beta}$ for each $\alpha, \beta \in \{0,1\}$, $g \in [1, G/\ell]$, and $i \in [1, Q_{\mathsf{lpn}}]$. If the check fails for a sharing, P_i aborts.
13. **Evaluate Garbled Circuit:** Let $\mathbf{ctx}_{g,i}^{\alpha,\beta} = (\mathsf{ctx}_{g\ell+1,i}^{\alpha,\beta}, \dots, \mathsf{ctx}_{2g\ell}^{\alpha,\beta})$ for each $\alpha, \beta \in \{0,1\}$, $g \in [1, G/\ell]$, and $i \in [1, Q_{\mathsf{lpn}}]$. Parties set

$$\mathcal{G} = \bigcup_{\substack{\alpha,\beta \in \{0,1\} \\ g \in [1,G]}} \left\{ \mathsf{ctx}_{g,1}^{\alpha,\beta}, \dots, \mathsf{ctx}_{g,Q_{\mathsf{lpn}}}^{\alpha,\beta} \right\},$$
$$\mathbf{X} = (\mathbf{inpk}_w^{\rho_w}, \rho_w)_{w \in [1, W_{\mathsf{inp}}]}, \mathsf{aux}_{\mathsf{dec}} = \left\{ \lambda_w^{\mathsf{out}} \right\}_{w \in \mathcal{W}_{\mathsf{out}}}$$

and output $\mathsf{GC.Eval}(\mathbf{X}, \mathcal{G}, \mathsf{aux}_{\mathsf{dec}})$.

$g(\lambda_{\mathsf{left}(g)} + \alpha, \lambda_{\mathsf{right}(g)} + \beta) + \lambda_{W_{\mathsf{inp}}+g}$ and $g(\cdot,\cdot)$ is the function being computed by the g-th gate. Finally, in the evaluation phase, clients broadcast $\rho_w = \lambda_w + v_w$ for each input wire w and parties reconstruct $(\mathsf{inpk}_{w,i}^{\rho_w})_{i=1}^{L_{\mathsf{lpn}}}$ which corresponds to a distributed evaluation of $\mathsf{GC.Encode}(\mathbf{v}, \mathsf{aux}_{\mathsf{enc}})$. It then follows that $(\mathcal{G}, \mathbf{X}, \mathsf{aux}_{\mathsf{dec}})$ reconstructed in Π_{mpc} are identically distributed to the output of the garbling scheme. □

Theorem 3. Let $L_{\mathsf{check}} \geq \frac{\kappa_s}{\log_2 |\mathbb{F}|}$ and $L_{\mathsf{mac}} \geq \frac{\kappa_s}{\log |\mathbb{F}| - 1}$. Π_{mpc} t-securely realizes $\mathcal{F}_{\mathsf{mpc}}$ in the $\{\mathcal{F}_{\mathsf{rand}}, \mathcal{F}_{\mathsf{coin}}, \mathcal{F}_{\mathsf{bitrand}}, \mathcal{F}_{\mathsf{mult}}, \mathcal{F}_{\mathsf{rand\text{-}sharing}}, \mathcal{F}_{\mathsf{mac\text{-}keygen}}\}$-hybrid model.

Proof. Let $\mathcal{A}$ denote the adversary and C be the set of corrupt parties. Let $\mathcal{H}$ be the set of honest parties, $\mathcal{H}_{\mathcal{H}}$ be a fixed subset of $\mathcal{H}$ of size $\mathsf{d}+1$, and $\mathcal{H}_C = \mathcal{H} \setminus \mathcal{H}_{\mathcal{H}}$. We construct a simulator Sim to simulate the behavior of honest parties.

For a sharing $[\mathbf{x}]$, let $\langle \mathbf{x} \rangle = (\mathbf{x}, \{([\mathbf{x}])_i\}_{\mathsf{P}_i \in C}, \boldsymbol{\delta}_x, \Delta_x)$ denote the secret $\mathbf{x}$, the shares of corrupt parties in $[\mathbf{x}]$, the additive error to the secret $\boldsymbol{\delta}_x$, and the additive error to the shares Δ_x. $\langle \mathbf{x} \rangle_{d'}$ is similarly defined for $[\mathbf{x}]_{d'}$. The simulator Sim described below maintains the invariant that for every sharing $[\mathbf{x}]$ in the pre-processing and garbling phases of Π_{mpc}, Sim knows $\langle \mathbf{x} \rangle$ such that the secret $\mathbf{x}$, the shares of the corrupt parties, $\boldsymbol{\delta}_x$, and $\boldsymbol{\Delta}_x$ are identically distributed to that in the real world. Note that Sim can maintain an identically distributed secret for every share $[\mathbf{x}]$ in the pre-processing and garbling phases because the protocol only involves sharings of random secrets over $\mathbb{F}$ and $\mathbb{F}_2$ sampled in the pre-processing phase. We use $\langle\langle \mathbf{x} \rangle\rangle$ to denote $(\langle \mathbf{x} \rangle, \{\langle \mathsf{kmac}_i \cdot \mathbf{x} \rangle\}_{i=1}^{L_{\mathsf{mac}}})$.

We now show that the invariant can be maintained for linear operations on shares. Let $[\mathbf{z}] = \alpha[\mathbf{x}]+\beta[\mathbf{y}]$ where $\alpha, \beta \in \mathbb{F}$. We define $\langle \mathbf{z} \rangle = \alpha\langle \mathbf{x} \rangle + \beta\langle \mathbf{y} \rangle$ to consist of the secret $\mathbf{z} = \alpha\mathbf{x} + \beta\mathbf{y}$, the shares of the corrupt parties $\left\{([\mathbf{z}])_i \mid ([\mathbf{z}])_i = \alpha([\mathbf{x}])_i + \beta([\mathbf{y}])_i\right\}$, the additive errors to the secret $\boldsymbol{\delta}_z = \alpha\boldsymbol{\delta}_x + \beta\boldsymbol{\delta}_y$, and the additive errors to the shares $\boldsymbol{\Delta}_z = \alpha\boldsymbol{\Delta}_x + \beta\boldsymbol{\Delta}_y$, and $\mathbf{z} = \alpha\mathbf{x} + \beta\mathbf{y}$. It is easy to see that $\langle \mathbf{z} \rangle$ computed in this manner is consistent to $[\mathbf{z}]$ computed locally by parties in the real world. We now proceed to formally describe the simulator.

The simulator initializes $\mathsf{flag} = 0$, and $\mathcal{S}_{\mathsf{cons}} = \mathcal{S}_{\mathsf{mac}} = \mathcal{S}_{\mathsf{zero}} = \emptyset$ and proceeds as follows:

- **Generate MAC Keys:** For every invocation of $\mathcal{F}_{\mathsf{mac\text{-}keygen}}$, Sim receives the shares of the corrupt parties and status from $\mathcal{A}$. If status = reject, Sim sends `abort` to $\mathcal{F}_{\mathsf{mpc}}$ and stops. Else, Sim samples $\mathsf{kmac} \leftarrow \mathbb{F}$ uniformly at random and initializes $\langle \mathsf{kmac} \rangle$ with the secret $\mathsf{kmac} \cdot \mathbf{1}$, $\boldsymbol{\delta}_{\mathsf{kmac}} = \mathbf{0}$, $\boldsymbol{\Delta}_{\mathsf{kmac}} = \mathbf{0}$, and the shares of the corrupt parties received from $\mathcal{A}$.

- **Generate Wire Labels:** For every invocation of $\mathcal{F}_{\mathsf{rand}}$, Sim receives the shares of the corrupt parties and the additive errors to the shares $\boldsymbol{\Delta}_r$ from $\mathcal{A}$. It then samples $\mathbf{r} \leftarrow \mathbb{F}^{\ell}$ uniformly at random and initializes $\langle \mathbf{r} \rangle$ with the secret $\mathbf{r}$, the shares of the corrupt parties received from $\mathcal{A}$, $\boldsymbol{\delta}_r = \mathbf{0}$, and $\boldsymbol{\Delta}_r$.

  To simulate Π_{auth}, Sim needs to emulate the invocations to $\mathcal{F}_{\mathsf{mult}}$, which it does as follows. For every invocation of $\mathcal{F}_{\mathsf{mult}}$ with inputs $[\mathbf{x}]$ and $[\mathbf{y}]$, Sim knows $\langle \mathbf{x} \rangle$ and $\langle \mathbf{y} \rangle$ due to the invariant. Thus, Sim sends $\boldsymbol{\Delta}_x$, $\boldsymbol{\Delta}_y$, and the shares of the corrupt parties in $[\mathbf{x}]$ and $[\mathbf{y}]$ to $\mathcal{A}$ and receives the shares of the corrupt parties in the product sharing $[\mathbf{z}]$, the additive errors to the product $\boldsymbol{\delta}_z$, and the additive errors to the product shares $\boldsymbol{\Delta}_z$. Sim sets $\mathbf{z} = \mathbf{x} \cdot \mathbf{y} + \boldsymbol{\delta}_z$ and thus learns $\langle \mathbf{z} \rangle$.

  Sim now simulates Π_{auth} by emulating each invocation of $\mathcal{F}_{\mathsf{mult}}$ as above. Note that the invariant is satisfied for every input. It follows that Sim learns $\langle\langle \mathbf{x} \rangle\rangle$ when Π_{auth} is called with inputs $[\mathbf{x}]$ and $\{[\mathsf{kmac}_i]\}_{i=1}^{L_{\mathsf{mac}}}$. Sim updates $\mathcal{S}_{\mathsf{cons}}$ as in the protocol.

- **Generate Wire Masks:** For every invocation of $\mathcal{F}_{\mathsf{bitrand}}$, Sim receives the shares of the corrupt parties, the additive errors to the secrets $\boldsymbol{\delta}_b$ and the additive errors to the shares $\boldsymbol{\Delta}_b$ from $\mathcal{A}$. If $\boldsymbol{\delta}_b \notin \{0,1\}^{\ell}$, Sim sets $\mathsf{flag} = 1$. It then samples $\mathbf{b} \leftarrow \{0,1\}^{\ell}$ and initializes $\langle \mathbf{b} \rangle$ with the secret $\mathbf{b} + \boldsymbol{\delta}_b$, the shares of the corrupt parties in $[\mathbf{b}]$, additive error of $\mathbf{0}$ to the secrets, and additive error of $\boldsymbol{\Delta}_b$ to the shares.

  Sim simulates Π_{auth} as above.

  To simulate $\Pi_{\mathsf{check\text{-}bit}}$, Sim needs to simulate $\Pi_{\mathsf{auth\text{-}mult}}$ which it does by emulating calls to $\mathcal{F}_{\mathsf{mult}}$ as above and updating $\mathcal{S}_{\mathsf{cons}}$ and $\mathcal{S}_{\mathsf{mac}}$ as in the protocol. Note that Sim knows $\langle\langle \mathbf{x} \rangle\rangle$ and $\langle\langle \mathbf{y} \rangle\rangle$ when $\Pi_{\mathsf{auth\text{-}mult}}$ is run with inputs $[\![\mathbf{x}]\!]$ and $[\![\mathbf{y}]\!]$ due to the invariant. Sim thus learns $\langle\langle \mathbf{z} \rangle\rangle$ for the output $[\![\mathbf{z}]\!]$ of $\Pi_{\mathsf{auth\text{-}mult}}$.

  To simulate $\Pi_{\mathsf{check\text{-}bit}}$, Sim simulates $\Pi_{\mathsf{auth\text{-}mult}}$ as above. It then computes $\langle \mathbf{o} \rangle = \langle \mathbf{b}^2 \rangle - \langle \mathbf{b} \rangle$ and updates $\mathcal{S}_{\mathsf{zero}}$ as in the protocol.

- **Generate LPN Errors:** Sim simulates Π_{error} by emulating $\mathcal{F}_{\mathsf{bitrand}}$, $\mathcal{F}_{\mathsf{rand}}$, and $\mathcal{F}_{\mathsf{mult}}$ and simulating Π_{auth}, $\Pi_{\mathsf{auth\text{-}mult}}$, and $\Pi_{\mathsf{check\text{-}bit}}$ as above. Since the simulator maintains the invariant when simulating each subprotocol and emulating each functionality, it follows that the invariant holds true for Π_{error} and that the simulator learns $\langle\langle \boldsymbol{\epsilon} \rangle\rangle$ for each run of Π_{error}.

- **Transform Masks and Labels:** To simulate $\Pi_{\mathsf{auth\text{-}trans}}$, Sim needs to simulate Π_{trans} which it does as follows.

  - The input sharing for Π_{trans} is either a degree-d packed sharing or a degree-$(\mathrm{d}+\ell-1)$ packed sharing of a secret $\mathbf{x}$. Sim knows $\langle\mathbf{x}\rangle$ in case of the former, from the invariant. On the other hand, a degree-$(\mathrm{d}+\ell-1)$ packed sharing in the protocol is obtained from local computation with a publicly known degree-$(\ell-1)$ sharing of a vector of constants. Thus Sim can compute the shares of the corrupt parties, the secret $\mathbf{x}$, and the additive errors to the secret $\boldsymbol{\delta}_x$ in this case too by emulating the local computation.

  - Sim emulates $\mathcal{F}_{\text{rand-sharing}}$ and receives the shares of the corrupt parties in $[\mathbf{r}\,|\mathrm{pos}]_{n-1}$ and the shares of the corrupt parties in $[f(\mathbf{r})\,|\mathrm{pos}']^{\mathcal{H}}$, the additive error to the transformed output $\boldsymbol{\delta}_{f(r)}$, and the additive errors to the transformed sharing $\Delta_{f(r)}$.

  - Sim computes the shares of the corrupt parties in $[\mathbf{x}+\mathbf{r}\,|\mathrm{pos}]_{n-1}$. It then samples the secret $\mathbf{x}+\mathbf{r} \longleftarrow \mathbb{F}^{\ell}$ uniformly at random and computes $f(\mathbf{x}+\mathbf{r})$, and the whole sharing $[\mathbf{x}+\mathbf{r}\,|\mathrm{pos}]_{n-1}$ such that the shares of the corrupt parties are consistent. It sends the shares of the honest parties to $\mathcal{A}$.

  - Sim receives the shares of the honest parties in $[f(\mathbf{x}+\mathbf{r})\,|\mathrm{pos}']$ from $\mathcal{A}$. It then computes the whole sharing $[f(\mathbf{x}+\mathbf{r})\,|\mathrm{pos}']^{\mathcal{H}}$ from the shares of the parties in $\mathcal{H}_{\mathcal{H}}$ and sets $\Delta_{f(x+r)} = [f(\mathbf{x}+\mathbf{r})\,|\mathrm{pos}'] - [f(\mathbf{x}+\mathbf{r})\,|\mathrm{pos}']^{\mathcal{H}}$. It reconstructs the secret $\overline{f(\mathbf{x}+\mathbf{r})}$ from the sharing $[f(\mathbf{x}+\mathbf{r})\,|\mathrm{pos}']^{\mathcal{H}}$ and sets $\boldsymbol{\delta}_{f(x+r)} = f(\mathbf{x}+\mathbf{r}) - \overline{f(\mathbf{x}+\mathbf{r})}$.

  - Sim then computes $\langle f(\mathbf{x})\rangle$ by computing the shares of the corrupt parties in $[f(\mathbf{x})\,|\mathrm{pos}']^{\mathcal{H}}$ as the shares of corrupt parties in $[f(\mathbf{x}+\mathbf{r})\,|\mathrm{pos}']^{\mathcal{H}} - [f(\mathbf{r})\,|\mathrm{pos}']^{\mathcal{H}}$, the additive error to the secret $\boldsymbol{\delta}_{f(x)} = f(\boldsymbol{\delta}_x) + \boldsymbol{\delta}_{f(x+r)} - \boldsymbol{\delta}_{f(r)}$, the secret $f(\mathbf{x}) + \boldsymbol{\delta}_{f(x+r)} - \boldsymbol{\delta}_{f(r)}$ and the additive error to the share $\Delta_{f(x)} = \Delta_{f(x+r)} - \Delta_{f(r)}$.

  Sim simulates $\Pi_{\text{auth-trans}}$ by simulating every instance of Π_{trans} as above. It updates $\mathcal{S}_{\text{mac}}$ as in the protocol.

- **Compute Plaintexts, Compute Ciphertexts, and Transform Shares for Input and Output:** These steps of the protocol involve either linear operations of shares or instances of $\Pi_{\text{auth-mult}}$ and $\Pi_{\text{auth-trans}}$ (run after multiplication with a vector of constants) all of which Sim simulates as discussed previously. Importantly, the invariant is maintained across all these steps.

- **Verification:** Sim simulates $\Pi_{\text{check-cons}}$ as follows.

  - Sim emulates $\mathcal{F}_{\text{rand}}$ as above and learns $\langle\mathbf{r}_i\rangle$ for every $i \in [1, L_{\text{check}}]$.

  - Sim emulates each invocation of $\mathcal{F}_{\text{coin}}$ by sampling $\alpha \longleftarrow \mathbb{F}$ uniformly at random and sending it to $\mathcal{A}$. If $\mathcal{A}$ sends `abort`, Sim sends `abort` to $\mathcal{F}_{\text{mpc}}$ and stops.

  - Note that from the invariant, Sim knows $\langle\mathbf{x}\rangle$ for every $[\mathbf{x}] \in \mathcal{S}_{\text{cons}}$. Sim thus computes $\langle\mathbf{y}_i\rangle = \sum_{j=1}^{m} \alpha_j^i \langle\mathbf{x}_j\rangle + \alpha_{m+1}^i \langle\mathbf{r}_i\rangle$ for each $i \in [1, L_{\text{check}}]$ where α_j^i for $j \in [1, m+1]$ and $i \in [1, L_{\text{check}}]$ have been sampled by Sim in the previous step.

  - Sim samples $\mathbf{y}_i \longleftarrow \mathbb{F}^{\ell}$ uniformly at random and then computes the whole sharing $[\mathbf{y}_i]^{\mathcal{H}}$ such that the shares of the corrupt parties are the same as those in $\langle\mathbf{y}_i\rangle$ for each $i \in [1, L_{\text{check}}]$. It then computes $[\mathbf{y}_i] = [\mathbf{y}_i]^{\mathcal{H}} + \Delta_{y_i}$ and follows the steps of the protocol.

  - If no party aborts at the end of the protocol but there exists $\Delta_x \neq \mathbf{0}$ for some $[\mathbf{x}] \in \mathcal{S}_{\text{cons}}$ then Sim aborts.

  Sim simulates $\Pi_{\text{check-mac}}$ as follows.

- Sim emulates $\mathcal{F}_{\mathsf{coin}}$ as described above.
- Sim computes the whole sharing $[\mathsf{kmac}_i]$ using the secret and the shares of corrupt parties in $\langle\mathsf{kmac}_i\rangle$ for each $i \in [1, L_{\mathsf{mac}}]$. It then sends the shares of the honest parties in $[\mathsf{kmac}_i]$ to $\mathcal{A}$ and receives the shares of the corrupt parties for each $i \in [1, L_{\mathsf{mac}}]$. If reconstruction of the secret kmac_i fails for any $i \in [1, L_{\mathsf{mac}}]$, Sim sends `abort` to $\mathcal{F}_{\mathsf{mpc}}$ and stops.
- Note that Sim knows $\langle\langle\mathbf{x}\rangle\rangle$ for every $[\![\mathbf{x}]\!] \in \mathcal{S}_{\mathsf{mac}}$ as well as α_j^i for each $j \in [1, m]$ and $i \in [1, L_{\mathsf{mac}}]$. Sim can thus compute $\langle\mathbf{check}_i\rangle$ for every $i \in [1, L_{\mathsf{mac}}]$.
- Sim computes the whole sharing $[\mathbf{check}_i]$ using the secret and the shares of corrupt parties in $\langle\mathbf{check}_i\rangle$ for each $i \in [1, L_{\mathsf{mac}}]$. It then sends the shares of honest parties in $[\mathbf{check}_i]$ to $\mathcal{A}$ and receives the corrupt parties shares in $[\mathbf{check}_i]$ from $\mathcal{A}$ for each $i \in [1, L_{\mathsf{mac}}]$. If the shares received by $\mathcal{A}$ are different from those in $\langle\mathbf{check}_i\rangle$ for any $i \in [1, L_{\mathsf{mac}}]$ or if $\mathbf{check}_i \neq \mathbf{0}$ for some $i \in [1, L_{\mathsf{mac}}]$, then Sim sends `abort` to $\mathcal{F}_{\mathsf{mpc}}$ and stops. If $\boldsymbol{\delta}_x \neq \mathbf{0}$ for any $[\![\mathbf{x}]\!] \in \mathcal{S}_{\mathsf{mac}}$ but $\mathbf{check}_i = \mathbf{0}$ for all $i \in [1, L_{\mathsf{mac}}]$, Sim aborts.

Sim emulates the check for shares in $\mathcal{S}_{\mathsf{zero}}$ as follows.

- Sim emulates $\mathcal{F}_{\mathsf{coin}}$ as described above.
- Note that Sim knows $\langle\mathbf{o}\rangle$ for every $[\mathbf{o}] \in \mathcal{S}_{\mathsf{zero}}$ as well as α_j^i for each $j \in [1, m]$ and $i \in [1, L_{\mathsf{check}}]$. Sim can thus compute $\langle\mathbf{check}_i\rangle$ for every $i \in [1, L_{\mathsf{check}}]$.
- Sim computes the whole sharing $[\mathbf{check}_i]$ using the secret and the shares of corrupt parties in $\langle\mathbf{check}_i\rangle$ for each $i \in [1, L_{\mathsf{check}}]$. It then sends the shares of honest parties in $[\mathbf{check}_i]$ to $\mathcal{A}$ and receives the corrupt parties shares in $[\mathbf{check}_i]$ from $\mathcal{A}$ for each $i \in [1, L_{\mathsf{check}}]$. If the shares received by $\mathcal{A}$ are different from those in $\langle\mathbf{check}_i\rangle$ for any $i \in [1, L_{\mathsf{check}}]$ or if $\mathbf{check}_i \neq \mathbf{0}$ for some $i \in [1, L_{\mathsf{check}}]$, Sim sends `abort` to $\mathcal{F}_{\mathsf{mpc}}$ and stops. Additionally, if $\mathsf{flag} = 1$ and $\mathbf{check}_i = \mathbf{0}$ for each $i \in L_{\mathsf{check}}$, Sim aborts.

- **Share Inputs:** Let $\mathcal{W}_{\mathcal{H}}^{\mathsf{inp}}$ denote the set of input wires to the circuit corresponding to honest clients and let $\mathcal{W}_{C}^{\mathsf{inp}}$ denote the set of input wires corresponding to corrupt clients. Note that the Sim knows the shares of the corrupt parties in $[\lambda_w^{\mathsf{inp}}]$ for every $w \in [1, W_{\mathsf{inp}}]$.
  - Sim runs $\mathsf{Sim}_{\mathsf{gc}}(\mathsf{C}, \mathcal{W}_{\mathcal{H}}^{\mathsf{inp}}, 1^{\kappa_c})$ to obtain $\{\lambda_w\}_{w \in \mathcal{W}_C^{\mathsf{inp}}}$, $\{\rho_w\}_{w \in \mathcal{W}_{\mathcal{H}}^{\mathsf{inp}}}$, and state.
  - It then computes the whole sharing $[\lambda_w^{\mathsf{inp}}]$ for each $w \in \mathcal{W}_C^{\mathsf{inp}}$ such that the shares of the corrupt parties are consistent and where λ_w^{inp} is equal λ_w output by $\mathsf{Sim}_{\mathsf{gc}}$ in the previous step. It then sends the shares of the honest parties in $[\lambda_w^{\mathsf{inp}}]$ to $\mathcal{A}$ for each $w \in \mathcal{W}_C^{\mathsf{inp}}$ and receives the shares of the corrupt parties in $[\lambda_w^{\mathsf{inp}}]$ for each $w \in \mathcal{W}_{\mathcal{H}}^{\mathsf{inp}}$. If the shares of corrupt parties in $[\lambda_w^{\mathsf{inp}}]$ sent by $\mathcal{A}$ are incorrect for any $w \in \mathcal{W}_{\mathcal{H}}^{\mathsf{inp}}$, Sim sends `abort` to $\mathcal{F}_{\mathsf{mpc}}$ and stops.
  - Sim emulates a broadcast of ρ_w for each $w \in \mathcal{W}_{\mathcal{H}}^{\mathsf{inp}}$ and receives ρ_w from $\mathcal{A}$ for each $w \in \mathcal{W}_C^{\mathsf{inp}}$.
  - For every $w \in \mathcal{W}_C^{\mathsf{inp}}$, Sim extracts the input of the corrupt clients by computing $v_w = \rho_w + \lambda_w$. It then sends the inputs of the corrupt clients to $\mathcal{F}_{\mathsf{mpc}}$ and receives the output $\mathsf{C}(\mathbf{x})$.
  - Sim then computes $(\mathcal{G}, \mathbf{X}, \mathsf{aux}_{\mathsf{dec}}) \leftarrow \mathsf{Sim}_{\mathsf{gc}}(\mathsf{C}(\mathbf{x}), \{v_w\}_{w \in \mathcal{W}_C^{\mathsf{inp}}}, \mathsf{state})$.
  - Note that the simulator has the corrupt parties shares in $[\mathsf{inpk}_{w,i}^{\rho_w}]$ for each $w \in [1, W_{\mathsf{inp}}]$, and $i \in$

$[1, Q_{\mathsf{lpn}}]$. It parses $\mathbf{X}$ such that the w-th element $(\mathbf{X})_w = (\left\{\mathsf{inpk}_{w,i}^{\rho_w}\right\}_{i=1}^{Q_{\mathsf{lpn}}}, \rho_w)$ for each $w \in [1, W_{\mathsf{inp}}]$ and then computes the whole sharing $[\mathsf{inpk}_{w,i}^{\rho_w}]$ such that the shares of the corrupt parties are consistent for every $w \in [1, W_{\mathsf{inp}}]$, and $i \in [1, Q_{\mathsf{lpn}}]$. It sends the shares of the honest parties to $\mathcal{A}$ and receives the shares of the corrupt parties from $\mathcal{A}$. If the shares of corrupt parties received from $\mathcal{A}$ are different from the ones held by Sim, Sim sends `abort` to $\mathcal{F}_{\mathsf{mpc}}$ and stops.

- **Reconstruct Garbled Circuit:** Sim already knows the shares of corrupt parties in $[\mathbf{ctx}_{g,i}^{\alpha,\beta}]$ for each $\alpha, \beta \in \{0,1\}$, $g \in [1, G/\ell]$, and $i \in [1, Q_{\mathsf{lpn}}]$. Sim parses $\mathcal{G}$ computed in the previous step as $\mathcal{G} = \left\{\mathbf{ctx}_g^{0,0}, \mathbf{ctx}_g^{0,1}, \mathbf{ctx}_g^{1,0}, \mathbf{ctx}_g^{1,1}\right\}_{g=1}^{W}$ and each $\mathbf{ctx}_g^{\alpha,\beta} = \left\{\mathsf{ctx}_{g,i}^{\alpha,\beta}\right\}_{i=1}^{Q_{\mathsf{lpn}}}$. It then computes the whole sharing $[\mathbf{ctx}_{g,i}^{\alpha,\beta}]$ for each $\alpha, \beta \in \{0,1\}$, $g \in [1, G/\ell]$, and $i \in [1, Q_{\mathsf{lpn}}]$ and follows the steps of the protocol.
- **Evaluate Garbled Circuit:** Sim sends `continue` to $\mathcal{F}_{\mathsf{mpc}}$ and outputs the same value as $\mathcal{A}$.

Lemma 10. The joint distribution consisting of the view of the adversary $\mathcal{A}$ and the outputs of honest parties are computationally indistinguishable in the real and ideal worlds.

Proof. We prove that the distributions are computationally indistinguishable using a hybrid argument.

- Hyb_0: This is the real world execution.
- Hyb_1: In this hybrid, Sim simulates the circuit independent pre-processing phase as above.

  $\mathsf{Hyb}_0 \overset{\mathrm{p}}{=} \mathsf{Hyb}_1$: In this hybrid, the simulator emulates calls to $\mathcal{F}_{\mathsf{mac\text{-}keygen}}$, $\mathcal{F}_{\mathsf{rand}}$, $\mathcal{F}_{\mathsf{mult}}$ and $\mathcal{F}_{\mathsf{bitrand}}$ in the pre-processing phase. Since the simulator follows the description of the functionality in all cases, the shares of corrupt parties and additive errors to the shares are identically distributed in Hyb_1 and Hyb_0. Moreover, the simulator can correct compute the shares of corrupt parties and additive errors to the shares in case of local operations as argued earlier. Thus, Hyb_1 and Hyb_0 are identically distributed.

- Hyb_2: In this hybrid, Sim simulates $\Pi_{\mathsf{auth\text{-}trans}}$ as above and honestly emulates $\mathcal{F}_{\mathsf{mult}}$ when parties invoke $\Pi_{\mathsf{auth\text{-}mult}}$ in the garbling phase.

  $\mathsf{Hyb}_1 \overset{\mathrm{p}}{=} \mathsf{Hyb}_2$: Since Sim honestly emulates $\mathcal{F}_{\mathsf{mult}}$, it follows that view of $\mathcal{A}$ in $\Pi_{\mathsf{auth\text{-}mult}}$ in Hyb_1 and Hyb_2 are identically distributed. We next observe that $\mathcal{A}$'s view for $\mathcal{F}_{\mathsf{rand\text{-}sharing}}$, invoked in $\Pi_{\mathsf{auth\text{-}trans}}$, is identically distributed in Hyb_1 and Hyb_2 since its view only includes sending the shares of the corrupt parties and the additive errors to the secrets and shares to Sim. The only difference in $\mathcal{A}$'s view during Π_{trans} between Hyb_1 and Hyb_2 is that Sim sends the shares of the honest parties in a random degree-$(n-1)$ packed Shamir sharing instead of actually computing the shares of honest parties in $[\mathbf{x} + \mathbf{r}\,|\mathsf{pos}]_{n-1}$. However, $[\mathbf{r}\,|\mathsf{pos}]_{n-1}$ is a random degree-$(n-1)$ sharing which implies that $[\mathbf{x} + \mathbf{r}\,|\mathsf{pos}]_{n-1}$ is a random degree-$(n-1)$ packed Shamir sharing too. Thus, P_1 receives the shares of a random degree-$(n-1)$ packed Shamir sharing in both Hyb_1 and Hyb_2 and thus the view of $\mathcal{A}$ in Hyb_1 and Hyb_2 for Π_{trans}, and in turn $\Pi_{\mathsf{auth\text{-}trans}}$, is identical. It follows that Hyb_1 and Hyb_2 are identically distributed.

- Hyb_3: In this hybrid, Sim simulates $\Pi_{\mathsf{auth\text{-}mult}}$ as above.

  $\mathsf{Hyb}_2 \overset{\mathrm{p}}{=} \mathsf{Hyb}_3$: In Hyb_2, Sim uses the real shares of corrupt parties and the additive errors to the shares while in Hyb_3, these are computed by Sim. However, it is easy to see that the two are identically distributed since either $\mathcal{A}$ directly sends the shares of corrupt parties and additive errors to Sim or they are a linear combination of values sent by $\mathcal{A}$. Thus, Hyb_2 and Hyb_3 are identically distributed.

- Hyb_4: In this hybrid, Sim simulates $\Pi_{\mathsf{check\text{-}cons}}$ as above.

$\mathsf{Hyb}_3 \overset{s}{\approx} \mathsf{Hyb}_4$: Consider an arbitrary $i \in [1, L_{\mathsf{check}}]$. Compared to Hyb_3, the shares of honest parties in $[\mathbf{y}_i]^{\mathcal{H}}$ correspond to a sharing of a uniformly random vector $\mathbf{y}_i$ instead of the linear combination $\mathbf{y}_i = \sum_{j=1}^{m} \alpha_j^i \mathbf{x}_j + \alpha_{m+1}^i \mathbf{r}_i$. However, since $\mathbf{r}_i$ is uniformly random and unknown to $\mathcal{A}$, the distribution of $\mathbf{y}_i$ in $\Pi_{\mathsf{check\text{-}cons}}$ is also uniformly random to $\mathcal{A}$. Moreover, $\mathbf{\Delta}_{y_i}$ is identically distributed in Hyb_3 and Hyb_4 since α_j^i is uniformly random for every $j \in [1, m]$. Since Sim computes the shares of honest parties as $[\mathbf{y}_i] = [\mathbf{y}_i]^{\mathcal{H}} + \mathbf{\Delta}_{y_i}$ it follows that $[\mathbf{y}_i]$ is also identically distributed in both hybrids. Thus, the shares of honest parties received by $\mathcal{A}$ are identically distributed in Hyb_3 and Hyb_4.

Finally, Sim aborts in this hybrid, if $\mathbf{\Delta}_x \neq 0$ for any $[\mathbf{x}] \in \mathcal{S}_{\mathsf{cons}}$ even if $[\mathbf{y}_i]$ is a valid sharing for every $i \in [1, L_{\mathsf{check}}]$. However, the probability that this happens is $|\mathbb{F}|^{-L_{\mathsf{check}}}$ from Lemma 8 which is negligible when $L_{\mathsf{check}} \geq \kappa_s / \log_2 |\mathbb{F}|$. Thus, Hyb_4 is statistically close to Hyb_3.

— Hyb_5: In this hybrid, Sim simulates $\Pi_{\mathsf{check\text{-}mac}}$ as above.

$\mathsf{Hyb}_4 \overset{s}{\approx} \mathsf{Hyb}_5$: Note that $\mathcal{A}$'s view up to this point is independent of the secret $\mathbf{x}$ for any $\langle \mathbf{x} \rangle$ known by Sim since the simulator only uses the shares of the corrupt parties and additive errors to the shares $\mathbf{\Delta}_x$; both of which are independent of the secret. In essence, Sim starts simulating the secrets in this hybrid. Compared to Hyb_4, the shares of honest parties in $[\mathsf{kmac}_i]$ now correspond to a sharing of uniformly random value kmac_i instead of the actual MAC key for each $i \in [1, L_{\mathsf{mac}}]$. Moreover, Sim computes the shares of honest parties based on its computation of $\mathbf{check}_i$ for each $i \in [1, L_{\mathsf{mac}}]$. Additionally, Sim aborts if $\boldsymbol{\delta}_x \neq \mathbf{0}$ for any $[\![\mathbf{x}]\!] \in \mathcal{S}_{\mathsf{mac}}$.

We first show that $\{\mathsf{kmac}_i\}_{i=1}^{L_{\mathsf{mac}}}$ are identically distributed in Hyb_4 and Hyb_5 since the view of $\mathcal{A}$ until this point is independent of it. Specifically, $\mathcal{A}$ only learns the shares of corrupt parties in $[\mathsf{kmac}_i]$ and $[\mathsf{kmac}_i \cdot \mathbf{x}]$ for any authenticated packed sharing $[\![\mathbf{x}]\!]$ in the pre-processing and garbling phases. This is easy to see in case of Π_{auth}, $\Pi_{\mathsf{auth\text{-}mult}}$, and $\mathcal{F}_{\mathsf{mac\text{-}keygen}}$ from the definition of the simulator. In case of $\Pi_{\mathsf{auth\text{-}trans}}$, Sim sends the shares of honest parties for a random degree-$(n-1)$ to $\mathcal{A}$ which is independent of the secrets $\{\mathsf{kmac}_i\}_{i=1}^{L_{\mathsf{mac}}}$. Finally, in case of $\Pi_{\mathsf{check\text{-}cons}}$, Sim sends the shares of honest parties in a random degree-d sharing which is again independent of the MAC keys. Thus, the shares of honest parties for $[\mathsf{kmac}_i]$ received by $\mathcal{A}$ are identically distributed in Hyb_4 and Hyb_5 for each $i \in [1, L_{\mathsf{mac}}]$.

We next show that the shares of honest parties in $[\mathbf{check}_i]$ are identically distributed in both hybrids for each $i \in [1, L_{\mathsf{mac}}]$. Consider any arbitrary $i \in [1, L_{\mathsf{mac}}]$. We first show that $\mathbf{check}_i$, as computed by Sim in Hyb_5 is identically distributed to that in Hyb_4. Since parties only perform local computation on their shares in $\Pi_{\mathsf{check\text{-}mac}}$ to compute their shares in $[\mathbf{check}_i]$, the shares of corrupt parties are identically distributed in both hybrids. Thus, the shares of honest parties in $[\mathbf{check}_i]$ are identically distributed in both hybrids since they are completely determined by the secret and the shares of corrupt parties. We now proceed to prove that $\mathbf{check}_i$ is identically distributed in both hybrids. In Hyb_4, $\mathbf{check}_i$ is computed using secrets sampled by $\mathcal{F}_{\mathsf{rand}}$, $\mathcal{F}_{\mathsf{mac\text{-}keygen}}$, and $\mathcal{F}_{\mathsf{bitrand}}$ and computed in $\mathcal{F}_{\mathsf{mult}}$ and Π_{trans}. In Hyb_5, Sim samples and computes the secrets itself. However, Sim samples the secrets identical to the functionality and receives any additive errors to the secret from $\mathcal{A}$ in case of $\mathcal{F}_{\mathsf{rand}}$, $\mathcal{F}_{\mathsf{mac\text{-}keygen}}$, and $\mathcal{F}_{\mathsf{bitrand}}$. In case of $\mathcal{F}_{\mathsf{mult}}$, we have $\mathbf{\Delta}_x = \mathbf{\Delta}_y = \mathbf{0}$ since Sim did abort at the end of $\Pi_{\mathsf{check\text{-}cons}}$ which in turn implies $\mathbf{z} = \mathbf{x} \cdot \mathbf{y} + \boldsymbol{\delta}_z$. Since $\boldsymbol{\delta}_z$ is sent by $\mathcal{A}$, $\mathbf{z}$ computed by Sim is identically distributed to the underlying secrets for the product sharing in Hyb_4 if the underlying secrets, $\mathbf{x}$ and $\mathbf{y}$, for the input sharings to $\mathcal{F}_{\mathsf{mult}}$ are identically distributed. In case of Π_{trans} in Hyb_4, the transformed secret corresponds to $\overline{f(\mathbf{x}+\mathbf{r})} - \overline{f(\mathbf{r})} = f(\mathbf{x}+\mathbf{r}) + \boldsymbol{\delta}_{f(x+r)} - f(\mathbf{r}) - \boldsymbol{\delta}_{f(r)} = f(\mathbf{x}) + \boldsymbol{\delta}_{f(x+r)} - \boldsymbol{\delta}_f(r)$ where $\boldsymbol{\delta}_{f(x+r)}$ is the error introduced by P_1 on $f(\mathbf{x}+\mathbf{r})$ which it reconstructs in Π_{trans}, and $\boldsymbol{\delta}_{f(r)}$ is the error $\mathcal{A}$ introduces in $\mathcal{F}_{\mathsf{rand\text{-}sharing}}$. Since $\boldsymbol{\delta}_{f(x+r)}$ and $\boldsymbol{\delta}_{f(r)}$ are sent by $\mathcal{A}$, the transformed secret is identically distributed in Hyb_4 and Hyb_5 if the input secret is identically distributed in both hybrids, since Sim

computes the transformed secret similar to that in Hyb_4. Thus, Sim maintains the invariant that the secret $\mathbf{x}$ in $\langle \mathbf{x} \rangle$ are identically distributed to those in Hyb_4 if the inputs to corresponding functionalities and protocols are identically distributed. However, the sharings input to any functionality or protocol in the pre-processing and garbling phase are either shares output by $\mathcal{F}_{\mathsf{rand}}, \mathcal{F}_{\mathsf{mac\text{-}keygen}}$, and $\mathcal{F}_{\mathsf{bitrand}}$ in which case we have shown them to be identically distributed in both hybrids or are themselves the output of $\mathcal{F}_{\mathsf{mult}}$, Π_{trans} and local computation. It follows that $\mathbf{check}_i$ is identically distributed in both hybrids which in turn implies that the shares of honest parties in $[\mathbf{check}_i]$ are identically distributed in both hybrids.

We now proceed to show that the probability that $\boldsymbol{\delta}_x \neq \mathbf{0}$ for some $\llbracket \mathbf{x} \rrbracket \in \mathcal{S}_{\mathsf{mac}}$ but $\mathbf{check}_i = \mathbf{0}$ for each $i \in [1, L_{\mathsf{mac}}]$ is negligible. This in turn implies that the probability Sim aborts in Hyb_5 is negligible. Towards proving this, we first show that if $\boldsymbol{\delta}_x \neq \mathbf{0}$ for some $\llbracket \mathbf{x} \rrbracket \in \mathcal{S}_{\mathsf{mac}}$ then $\mathbf{check}_i = \mathbf{0}$ in Hyb_4 with probability at most $\frac{2}{|\mathbb{F}|}$ for each $i \in [1, L_{\mathsf{mac}}]$. Consider an arbitrary $i \in [1, L_{\mathsf{mac}}]$. In $\Pi_{\mathsf{check\text{-}mac}}$, we have $[\mathbf{y}_i] = \sum_{j=1}^{m} \alpha_j^i [\mathbf{x}_j]$ and $[\mathsf{kmac}_i \cdot \mathbf{y}_i] = \sum_{j=1}^{m} \alpha_j^i [\mathsf{kmac}_i \cdot \mathbf{x}_j]$. We use $\mathbf{X}_k^i$ to denote the underlying secret for the sharing $[\mathsf{kmac_i} \cdot \mathbf{x_k}]$. Let $\llbracket \mathbf{x}_1 \rrbracket, \ldots, \llbracket \mathbf{x}_m \rrbracket$ be the order in which the shares were added into $\mathcal{S}_{\mathsf{mac}}$ and let k be the smallest index such that either $\boldsymbol{\delta}_{x_k} \neq 0$ or $\boldsymbol{\delta}_{X_k^i} \neq 0$. We have the following cases based on how $\llbracket \mathbf{x}_k \rrbracket$ was computed.

- We first consider the case when $\llbracket \mathbf{x}_k \rrbracket$ is the output of running Π_{auth} on $[\mathbf{x}]$ and $\mathcal{K}_{\mathsf{mac}}$. From the definition of $\mathcal{F}_{\mathsf{mult}}$ and the fact that $\Delta_{x_k} = \Delta_{\mathsf{kmac}_i} = \mathbf{0}$, we have $\mathbf{X}_k^i = \mathsf{kmac}_i \cdot \mathbf{x}_k + \boldsymbol{\delta}_{X_k^i}$. Thus, $\mathsf{kmac}_i \cdot \mathbf{x}_k - \mathbf{X}_k^i = \boldsymbol{\delta}_{X_k^i}$. This implies that if $\mathbf{check}_i = \mathbf{0}$, we have

  $$\alpha_j^k \cdot \boldsymbol{\delta}_{X_k^i} = \mathsf{kmac}_i \cdot \left(\sum_{j=1, j \neq k}^{m} \alpha_j^i \mathbf{x}_j \right) - \left(\sum_{j=1, j \neq k}^{m} \alpha_j^i \mathbf{X}_k^i \right).$$

  Since $\boldsymbol{\delta}_{X_k^i} \neq 0$ by definition of k, the above equality holds with probability at most $\frac{1}{|\mathbb{F}|}$ since α_j^k is uniformly random in $\mathbb{F}$.

- Consider the case when $\llbracket \mathbf{x}_k \rrbracket$ is the output of $\Pi_{\mathsf{auth\text{-}mult}}$ when run with inputs $\llbracket \mathbf{u} \rrbracket$ and $\llbracket \mathbf{v} \rrbracket$. From the definition of $\mathcal{F}_{\mathsf{mult}}$ and the fact that $\Delta_u = \Delta_v = \mathbf{0}$, we have $\mathbf{x}_k = \mathbf{u} \cdot \mathbf{v} + \boldsymbol{\delta}_{x_k}$. Similarly, we have $\mathbf{X}_k^i = (\mathsf{kmac}_i \cdot \mathbf{u} + \boldsymbol{\delta}') \cdot \mathrm{v} + \boldsymbol{\delta}_{X_k^i}$ where we use $\boldsymbol{\delta}'$ to denote the accumulated additive error on $[\mathsf{kmac}_i \cdot \mathbf{u}_k]$ from prior computation. However, since k is the smallest index such that either $\boldsymbol{\delta}_{x_k} \neq 0$ or $\boldsymbol{\delta}_{X_k^i} \neq 0$, we have $\boldsymbol{\delta}' = \mathbf{0}$. Thus,

  $$\begin{aligned} \mathbf{X}_k^i - \mathsf{kmac}_i \cdot \mathbf{x}_k &= \mathsf{kmac}_i \cdot \mathbf{u} \cdot \mathbf{v} + \mathsf{kmac}_i \cdot \boldsymbol{\delta}_{x_k} - (\mathsf{kmac}_i \cdot \mathbf{u}) \cdot \mathbf{v} - \boldsymbol{\delta}_{X_k^i} \\ &= \mathsf{kmac}_i \cdot \boldsymbol{\delta}_{x_k} - \boldsymbol{\delta}_{X_k^i}. \end{aligned}$$

  This implies that if $\mathbf{check}_i = 0$, we have

  $$\alpha_j^k (\mathsf{kmac}_i \cdot \boldsymbol{\delta}_{x_k} - \boldsymbol{\delta}_{X_k^i}) = \mathsf{kmac}_i \cdot \left(\sum_{j=1, j \neq k}^{m} \alpha_j^i \mathbf{x}_j \right) - \left(\sum_{j=1, j \neq k}^{m} \alpha_j^i \mathbf{X}_k^i \right).$$

  If $(\mathsf{kmac}_i \cdot \boldsymbol{\delta}_{x_k} - \boldsymbol{\delta}_{X_k^i}) \neq \mathbf{0}$ then the above equality holds with probability at most $|\mathbb{F}|^{-1}$ since α_j^k is uniformly random over $\mathbb{F}$. On the other hand, if $(\mathsf{kmac}_i \cdot \boldsymbol{\delta}_{x_k} - \boldsymbol{\delta}_{X_k^i}) = \mathbf{0}$ the equality always holds. However, the probability of that happening is at most $|\mathbb{F}|^{-1}$ since kmac_i is unknown to $\mathcal{A}$ when $\Pi_{\mathsf{auth\text{-}mult}}$ is run. Thus, the probability that $\mathbf{check}_i = \mathbf{0}$ despite $\boldsymbol{\delta}_x \neq \mathbf{0}$ is at most $\frac{1}{|\mathbb{F}|} + (1 - \frac{1}{|\mathbb{F}|}) \frac{1}{|\mathbb{F}|} \leq \frac{2}{|\mathbb{F}|}$.

- If $\llbracket \mathbf{x}_k \rrbracket$ is the output of Π_{error}, then it follows from a similar argument as above that $\mathbf{check}_i = \mathbf{0}$ with probability at most $\frac{1}{|\mathbb{F}|}$.

– Finally, we consider the case when $[\![\mathbf{x}_k]\!]$ is the output of one or more instances of $\Pi_{\text{auth-trans}}$. Note that in our protocol, we start with a sharing $[\![\mathbf{u}]\!]$ over the default positions pos_{def} and run $\Pi_{\text{auth-trans}}$ one or more times in succession (with only local computation in between successive runs) to obtain $[\![\mathbf{x}_k]\!]$. In every run of $\Pi_{\text{auth-trans}}$, $\mathcal{A}$ can introduce an additive error which accumulates across consecutive runs. Specifically, we have $\mathbf{x}_k = f(\mathbf{u}' + \boldsymbol{\delta}_{u'}) + \boldsymbol{\delta}_{f(u')} = f(\mathbf{u}') + f(\boldsymbol{\delta}_{u'}) + \boldsymbol{\delta}_{f(u')}$ where $\mathbf{u}'$ is the secret corresponding to the output of a previous transform, $\boldsymbol{\delta}_{u'}$ is the existing additive error on $\mathbf{u}'$, and $\boldsymbol{\delta}_{f(u')}$ is the additive error introduced in transformation of $\mathbf{u}'$ to $\mathbf{x}_k$. Thus, $\boldsymbol{\delta}_{x_k} = f(\boldsymbol{\delta}_{u'}) + \boldsymbol{\delta}_{f(u')}$. Sim computes $\boldsymbol{\delta}_{x_k}$ in a similar manner in this hybrid and aborts if $\boldsymbol{\delta}_{x_k} \neq 0$.

  Thus, we have $\mathbf{x}_k = f(\mathbf{u}) + \boldsymbol{\delta}_{x_k}$ and $\mathbf{X}^i_k = \mathsf{kmac}_i \cdot f(\mathbf{u}) + \boldsymbol{\delta}_{X^i_k}$ where $f(\cdot)$ is the composition of the individual linear functions applied in consecutive runs of $\Pi_{\text{auth-trans}}$. This implies that $\mathbf{X}^i_k - \mathsf{kmac}_i \cdot \mathbf{x}_k = \boldsymbol{\delta}_{X^i_k} - \mathsf{kmac}_i \cdot \boldsymbol{\delta}_{x_k}$. This implies that

$$\alpha^k_j(\boldsymbol{\delta}_{X^i_k} - \mathsf{kmac}_i \cdot \boldsymbol{\delta}_{x_k}) = \mathsf{kmac}_i \cdot \left(\sum_{j=1, j\neq k}^{m} \alpha^i_j \mathbf{x}_j \right) - \left(\sum_{j=1, j\neq k}^{m} \alpha^i_j \mathbf{X}^i_k \right).$$

  If $\boldsymbol{\delta}_{X^i_k} - \mathsf{kmac}_i \cdot \boldsymbol{\delta}_{x_k} \neq \mathbf{0}$ the above equation holds with probability at most $\frac{1}{|\mathbb{F}|}$ since α^k_j is uniformly random over $\mathbb{F}$. On the other hand, if $\boldsymbol{\delta}_{X^i_k} - \mathsf{kmac}_i \cdot \boldsymbol{\delta}_{x_k} = \mathbf{0}$ then the above equation always holds true. However, the probability that $\boldsymbol{\delta}_{X^i_k} - \mathsf{kmac}_i \cdot \boldsymbol{\delta}_{x_k} = \mathbf{0}$ is $\frac{1}{|\mathbb{F}|}$ since kmac_i is uniformly random over $\mathbb{F}$ and is unknown to $\mathcal{A}$ when $\Pi_{\text{auth-trans}}$ is run. Thus, the probability that $\mathbf{check}_i = \mathbf{0}$ despite $\boldsymbol{\delta}_x \neq \mathbf{0}$ is at most $\frac{1}{|\mathbb{F}|} + (1 - \frac{1}{|\mathbb{F}|})\frac{1}{|\mathbb{F}|} \leq \frac{2}{|\mathbb{F}|}$.

Thus, if $\boldsymbol{\delta}_x \neq \mathbf{0}$ for some $[\![\mathbf{x}]\!] \in \mathcal{S}_{\text{mac}}$ then $\mathbf{check}_i = \mathbf{0}$ in Hyb_4 with probability at most $\frac{2}{|\mathbb{F}|}$ for each $i \in [1, L_{\text{mac}}]$. This implies that the probability that $\mathbf{check}_i = \mathbf{0}$ for every $i \in [1, L_{\text{mac}}]$ is at most $|\mathbb{F}|^{-L_{\text{mac}}}$. Thus, the probability that Sim aborts when $\boldsymbol{\delta}_x \neq \mathbf{0}$ for any $[\![\mathbf{x}]\!] \in \mathcal{S}_{\text{mac}}$ but $\mathbf{check}_i = \mathbf{0}$ for every $i \in [1, L_{\text{mac}}]$ is at most $\left(\frac{2}{|\mathbb{F}|}\right)^{-L_{\text{mac}}}$ which is negligible when $L_{\text{mac}} \geq \frac{\kappa_s}{\log|\mathbb{F}|-1}$. Hence, Sim aborts with negligible probability in Hyb_5.

Finally, note that in Hyb_5, $\mathbf{check}_i$ is computed using the secrets sampled by Sim while the remainder of $\mathcal{A}$'s view as well as the output of honest parties are not computed using these secrets. On the other hand, $\mathbf{check}_i$ and the remainder of $\mathcal{A}$'s view is computed using the same underlying secret in Hyb_4. This does not render the hybrids distinguishable because when $\mathbf{check}_i \neq \mathbf{0}$ for some $i \in [1, L_{\text{mac}}]$, the view of $\mathcal{A}$ terminates at this point in both hybrids since in Hyb_4, honest parties abort during $\Pi_{\text{check-mac}}$ while Sim stops the simulation in Hyb_5. Moreover, $\mathbf{check}_i$ is identically distributed in Hyb_4 and Hyb_5 in this case, as shown previously. Similarly, the output of honest parties is `abort` in Hyb_4 as well as Hyb_5 in this case since $\mathbf{check}_i \neq \mathbf{0}$. On the other hand, when $\mathbf{check}_i = \mathbf{0}$ for each $i \in [1, L_{\text{mac}}]$, it is independent of the secrets. Thus, Hyb_4 and Hyb_5 are statistically indistinguishable.

— Hyb_6: In this hybrid, Sim simulates the check for shares in $\mathcal{S}_{\text{zero}}$ as described above.

$\mathsf{Hyb}_5 \overset{s}{\approx} \mathsf{Hyb}_6$: Compared to Hyb_5, Sim now computes $\mathbf{check}_i$ using the secrets it sampled for each $i \in [1, L_{\text{check}}]$. Moreover, Sim aborts if $\mathcal{A}$ sent $\boldsymbol{\delta}_b \notin \{0,1\}^\ell$ in any of the emulations of $\mathcal{F}_{\text{bitrand}}$.

We first show that the shares of honest parties in $[\mathbf{check}_i]$ sent by Sim are identically distributed in Hyb_5 and Hyb_6. Note that this step in the protocol is executed if Sim did not abort at the end of $\Pi_{\text{check-mac}}$, in which case, $\mathcal{A}$ received the shares of honest parties in a degree-d *zero* sharing in $\Pi_{\text{check-mac}}$. Thus, $\mathcal{A}$'s view up to this point is independent of the underlying secrets $\mathbf{b}$ for the sharing output by $\mathcal{F}_{\text{bitrand}}$. Note that Sim samples $\mathbf{b}$ uniformly at random and receives the additive errors to the secrets $\boldsymbol{\delta}_b$ from $\mathcal{A}$ and computes $\overline{\mathbf{b}} = \mathbf{b} + \boldsymbol{\delta}_b$ as the secret in $\langle \mathbf{b} \rangle$. Thus, the underlying secrets for $\mathcal{F}_{\text{bitrand}}$ are identically distributed in Hyb_5 and Hyb_6. All sharings in $\mathcal{S}_{\mathsf{zero}}$ are computed in instances of $\Pi_{\mathsf{check\text{-}bit}}$ and correspond to a sharing of the secret $\mathbf{o} = \overline{\mathbf{b}}^2 - \overline{\mathbf{b}}$ where there is no error in the computation of $\overline{\mathbf{b}}^2$ because Sim did not abort in $\Pi_{\mathsf{check\text{-}mac}}$. Since α^i_j are uniformly random in both hybrids for each $j \in [1, m]$ and $i \in [1, L_{\mathsf{check}}]$, it follows that $\mathbf{check}_i$ is identically distributed in both hybrids. Moreover, the shares of corrupt parties are identically distributed in both hybrids since parties compute their shares in $[\mathbf{check}_i]$ using local computation. It thus follows that the shares of honest parties in $[\mathbf{check}_i]$ are identically distributed in Hyb_5 and Hyb_6 since they are completely determined by the secret and the shares of corrupt parties.

Sim aborts in Hyb_6 if $\mathsf{flag} = 1$ and $\mathbf{check}_i = \mathbf{0}$ for all $i \in [1, L_{\mathsf{check}}]$. Consider an arbitrary $i \in [1, L_{\mathsf{check}}]$. We argue that the probability $\mathsf{flag} = 1$ and $\mathbf{check}_i = \mathbf{0}$ is at most $|\mathbb{F}|^{-1}$ in Hyb_5. This is because, for every $[\mathbf{o}] \in \mathcal{S}_{\mathsf{zero}}$, $\mathbf{o} = \overline{\mathbf{b}}^2 - \overline{\mathbf{b}} = \mathbf{0}$ if $\overline{\mathbf{b}} \in \{0, 1\}^\ell$ which in turn implies that $\mathbf{b} + \boldsymbol{\delta}_b \in \{0, 1\}^\ell$. Since $\mathbf{b}$ is a uniformly random boolean vector by definition, we have $\mathbf{o} = \mathbf{0}$ if $\boldsymbol{\delta}_b \in \{0, 1\}^\ell$. Thus, using a similar approach to the proof of Lemma 8, we can show that the probability that there exists $\boldsymbol{\delta}_b \notin \{0, 1\}^\ell$ but $\mathbf{check}_i = \mathbf{0}$ is at most $|\mathbb{F}|^{-1}$. Thus, the probability that $\mathbf{check}_i = \mathbf{0}$ for every $i \in [1, L_{\mathsf{check}}]$ despite $\boldsymbol{\delta}_b \notin \{0, 1\}^\ell$ is $|\mathbb{F}|^{-L_{\mathsf{check}}}$ which is negligible when $L_{\mathsf{check}} \geq \kappa_s / \log_2 |\mathbb{F}|$. Thus, Sim aborts with negligible probability in Hyb_6.

Finally, note that in Hyb_6, $\mathbf{check}_i$ is computed using the secrets sampled by Sim while the remainder of $\mathcal{A}$'s view as well as the output of honest parties are not computed using these secrets. On the other hand, $\mathbf{check}_i$ and the remainder of $\mathcal{A}$'s view is computed using the same underlying secret in Hyb_5. However, this does not render the hybrids distinguishable because when $\mathbf{check}_i = \mathbf{0}$ for each $i \in [1, L_{\mathsf{check}}]$, they are independent of the underlying secrets $\mathbf{b}$ for the sharing output by $\mathcal{F}_{\mathsf{bitrand}}$. On the other hand, when $\mathbf{check}_i \neq \mathbf{0}$ for some $i \in [1, L_{\mathsf{check}}]$, we have already shown that $\mathbf{check}_i$ is identically distributed in Hyb_5 and Hyb_6. Moreover, the view of $\mathcal{A}$ terminates at this point in Hyb_5 since honest parties abort while Sim stops the simulation in Hyb_6. Similarly, the output of honest parties is `abort` in Hyb_5 as well as Hyb_6 since there is some $i \in [1, L_{\mathsf{check}}]$ such that $\mathbf{check}_i \neq \mathbf{0}$. Thus, Hyb_6 is statistically close to Hyb_5.

- Hyb_7: In this hybrid, Sim simulates the reconstruction of λ^{inp}_w and the subsequent broadcast of ρ_w for every $w \in [1, W_{\mathsf{inp}}]$ as above.

  $\mathsf{Hyb}_6 \overset{c}{\approx} \mathsf{Hyb}_7$: In Hyb_6, λ^{inp}_w reconstructed towards corrupt clients corresponded to uniformly random bits and ρ_w broadcast by honest parties was equal to the masked values of their inputs. In Hyb_7, Sim runs $\mathsf{Sim}_{\mathsf{gc}}$ when reconstructing the secrets for λ^{inp}_w for every $w \in \mathcal{W}^{\mathsf{inp}}_{\mathcal{C}}$ and ρ_w for every $w \in \mathcal{W}^{\mathsf{inp}}_{\mathcal{H}}$.

  Note that the view of $\mathcal{A}$ up to this point is independent of the secrets λ^{inp}_w for each $w \in [1, W_{\mathsf{inp}}]$. Moreover, λ^{inp}_w is uniformly random in $\{0, 1\}^\ell$ since Sim did not abort during verification. Specifically, the underlying secret $\mathbf{b} + \boldsymbol{\delta}_b$ for the sharing output by $\mathcal{F}_{\mathsf{bitrand}}$ is a uniformly random bit vector when $\mathbf{b}$ is uniformly random and $\mathbf{b}, \boldsymbol{\delta}_b \in \{0, 1\}^\ell$. Thus, λ^{inp}_w in Hyb_6 are distributed identical to the output of $\mathsf{GC.MaskGen}$ for each $w \in \mathcal{W}^{\mathsf{inp}}_{\mathcal{C}}$. Moreover, ρ_w for each $w \in \mathcal{W}^{\mathsf{inp}}_{\mathcal{H}}$ correspond to the actual masked value of honest parties in Hyb_6. In Hyb_7, λ^{inp}_w for each $w \in \mathcal{W}^{\mathsf{inp}}_{\mathcal{C}}$ and ρ_w for each $w \in \mathcal{W}^{\mathsf{inp}}_{\mathcal{H}}$ correspond to the output of $\mathsf{Sim}_{\mathsf{gc}}$. Since the shares of corrupt parties held by Sim are identically distributed to those in Hyb_6 and the shares of honest parties are completely determined by the secrets and the shares of corrupt parties, it follows that Hyb_7 is computationally indistinguishable from Hyb_6 from the security of the garbling scheme. Specifically, the output of $\mathsf{Sim}_{\mathsf{gc}}$ is computationally indistinguishable from $\{\lambda_w\}_{w \in \mathcal{W}^{\mathsf{inp}}_{\mathcal{C}}}$ and $\{\rho_w\}_{w \in \mathcal{W}^{\mathsf{inp}}_{\mathcal{H}}}$ computed using the output of $\mathsf{GC.MaskGen}$.

- Hyb_8: In this hybrid, Sim simulates the rest of the evaluation phase as above.

$\mathsf{Hyb}_7 \overset{c}{\approx} \mathsf{Hyb}_8$: In comparison to Hyb_7, Sim now runs $\mathsf{Sim}_{\mathsf{gc}}$ to compute the secrets that are then reconstructed. However, from Lemma 9 $\mathcal{G}$, $\mathbf{X}$ and $\mathsf{aux}_{\mathsf{dec}}$ in Hyb_7 (step 13 of Π_{mpc}) are distributed such that $(\mathcal{G}, \mathsf{aux}_{\mathsf{enc}}, \mathsf{aux}_{\mathsf{dec}}) \leftarrow \mathsf{GC.Garble}(\mathsf{C}, \{\lambda_w\}_{w=1}^{W}, 1^{\kappa_c})$ and $\mathbf{X} \leftarrow \mathsf{GC.Encode}(\mathbf{x}, \mathsf{aux}_{\mathsf{enc}})$. The security of the garbling scheme then implies that Hyb_7 and Hyb_8 are computationally indistinguishable.

— Hyb_9: In this hybrid, the output of honest parties is from $\mathcal{F}_{\mathsf{mpc}}$.

$\mathsf{Hyb}_8 \overset{p}{\equiv} \mathsf{Hyb}_9$: Compared to Hyb_8, where the output of honest parties was from Π_{mpc}, the output in this hybrid is from $\mathcal{F}_{\mathsf{mpc}}$. Whenever Sim detects an abort in Hyb_8, it stops the simulation for $\mathcal{A}$ and sends `abort` to $\mathcal{F}_{\mathsf{mpc}}$ which then outputs `abort` to honest parties. It thus follows that the output of honest parties is identically distributed in Hyb_9 and Hyb_8 when there is an `abort`. We next show that when there is no abort, the output of honest parties is identically distributed in both hybrids. In both hybrids, $\mathcal{A}$ broadcasts $\{\rho_w\}_{w \in \mathcal{W}_C^{\mathsf{inp}}}$ on behalf of the corrupt clients. It then follows from Lemma 9 and the correctness of the garbling scheme that the output of honest parties is $\mathsf{C}(\mathbf{x})$ where $\mathbf{x}$ is the combined input of corrupt and honest clients. On the other hand, in Hyb_9, Sim computes the input of corrupt clients using $\{\rho_w\}_{w \in \mathcal{W}_C^{\mathsf{inp}}}$ broadcast by $\mathcal{A}$. It follows by the definition of GC.Encode that Sim computes the same inputs for corrupt clients as used by honest parties for computing the output in Hyb_8. Since Sim sends `continue` to $\mathcal{F}_{\mathsf{mpc}}$ in this case, it follows that the output of honest parties is $\mathsf{C}(\mathbf{x})$ from the definition of the functionality. Thus, Hyb_9 and Hyb_8 are identically distributed.

Hyb_9 corresponds to the distribution of the view of $\mathcal{A}$ and output of honest parties in the ideal world. Thus, it follows that the joint view consisting of the view of $\mathcal{A}$ and the output of honest parties in the real world is computationally indistinguishable from that in the ideal world. □

From Lemma 10, it follows that Π_{mpc} securely realizes $\mathcal{F}_{\mathsf{mpc}}$. □

7 Protocol Evaluation and Analysis

In this section, we attempt to get a better picture of the concrete performance of our protocol by analyzing its communication and computation costs. We first discuss some modifications to the protocol that improve its performance in practice followed by a discussion on the choice of optimal parameters for the LPN assumption and the binary super-invertible matrix. We then discuss the concrete communication and computation costs of our protocol and compare it to prior works.

7.1 Practical Protocol Optimizations

There are many simple optimizations or changes that can be made to reduce the constants that are involved in the communication complexity of the overall protocol. The following are some simple optimizations to reduce the concrete costs of the protocol.

1. **Pack circuit input wires separately.** Instead of packing keys and masks for blocks of all wires in the circuit together, separately pack the keys and masks for circuit input wires and the remaining wires (which correspond to output wires of individual gates) in the circuit. Then, $\boldsymbol{\lambda}_g^{\mathsf{out}}$ and $\mathbf{outk}_{g,i}^{b}$ need not be computed in steps 6(c) and 6(d) as they are equal to the values sampled in the pre-processing phase.

2. **Pack XOR and AND gates separately.** Instead of packing the ciphertexts in the garbled tables for gates of different types together, pack the ciphertexts for garbled tables of the same gate type together. While this does lead to slightly inefficient packing, it allows us to skip 7(c) and 7(d) entirely and only requires one of 7(a) or 7(b) to be computed for each packed sharing of the plaintext.

3. **Reduce cost for computing mask bits.** For AND gates, we only need one multiplication to compute $[\mathbf{s\text{-}and}_{\mathsf{g}}^{\alpha,\beta}]$ across all $\alpha, \beta \in \{0, 1\}$. This reduces the cost of garbling an AND gate by 3 multiplications. See [WRK17a] for more details.

4. **Replace $\Pi_{\mathsf{auth\text{-}trans}}$ with degree reduction wherever possible.** Sometimes in the protocol, e.g. in 5(b), $\Pi_{\mathsf{auth\text{-}trans}}$ is called not to change the positions of the secrets but solely to perform a degree reduction. In these cases $\Pi_{\mathsf{auth\text{-}trans}}$ is overly expensive. We instead replace $\Pi_{\mathsf{auth\text{-}trans}}$ with a call to Π_{rand} and Π_{zero} to perform a leader based degree reduction as in Π_{mult}.

7.2 LPN Parameters

To choose the parameters of LPN, following the work [BCGI18], we consider all known attacks for LPN. According to [BCGI18], the best known attacks for LPN are Gaussian elimination, information set decoding [Pra62] (ISD), BKW algorithm [BKW00], combination of attacks [EKM17], scaled-down BKW [Lyu05], and low-weight parity check attack [Zic17]. In our setting, we set the noise rate τ_{lpn} of our LPN-based encryption to be a constant, and only polynomial number of samples (See Definition of LPN in Section 3.4). Then we choose the parameters of Reed-Solomon codes to correct constant fraction of errors. As observed in [BCGI18], in such a parameter regime, the most effective attacks are low-parity check, Gaussian elimination, and ISD. For LPN problem with secret key size V, number of instance Q, and noise rate τ, we estimate the running time of the above attacks in the same way as [BCGI18]. We include the estimation of the $\log_2$ of attacking time here for completeness.

- **Gaussian Elimination:** $2.8 \log_2 V - V \cdot \log_2(1 - \tau)$
- **Low Weight Parity Check:** $\log_2(V + 1) + (\tau \cdot Q) \cdot \log_2 \left(\frac{Q}{Q-V-1} \right)$
- **ISD:** $\log_2 \left(\frac{\binom{Q}{\tau Q}}{\binom{Q-V}{\tau Q}} \right) + 2.8 \log_2(Q - V)$

We write a Python script to enumerate all parameters to find the smallest ciphertext size, ensuring all existing attacks for LPN need at least 2^{80} time. Specifically, to correctly decrypt the error-correcting code, we note that the weight of the noise vector $\mathbf{e}$ follows the binomial distribution. Namely, $\Pr_{\mathbf{e}}[\mathsf{weight}(\mathbf{e}) \leq \lfloor (d-1)/2 \rfloor] = \Pr[\mathsf{Binom}(Q, \tau) \leq \lfloor (d-1)/2 \rfloor]$. We numerically obtain the binomial distribution from SciPy library and set the decryption error to be bounded by 2^{-20}. For the error-correcting codes, we use the Reed-Solomon code with minimum distance d. To find the best parameters, we first enumerate the parameter Q from $1, 2, 3, \ldots$, then we try every possible $1 \leq V \leq Q$, and set $d = Q - V + 1$ as the distance for the Reed-Solomon code, then we enumerate $\mathsf{pow} = 1, 2, \ldots$ to find the largest $\tau = 2^{-\mathsf{pow}}$ such that

$$\Pr[\mathsf{Binom}(Q, \tau) \leq \lfloor (d-1)/2 \rfloor] \leq 2^{-20},$$

and check if the LPN parameter (Q, V, τ) has 80-bits security, according to the best known attacks above. We stop the enumeration when we find the first such Q to minimize the ciphertext size. Finally, we choose the field size to be any integer larger than Q to ensure the Reed-Solomon code has the correct parameters.

Our Python script finds the following parameters with smallest ciphertext size with 80-bit security: $Q = 721, V = 246, \tau = 2^{-2}$. For the Reed-Solomon code we choose $[721, 247, 475]_q$.

7.3 Parameters for Binary Super-Invertible Matrices

We use a concatenation of an outer Reed Solomon code and an inner binary error-correcting code to obtain the super-invertible matrix.

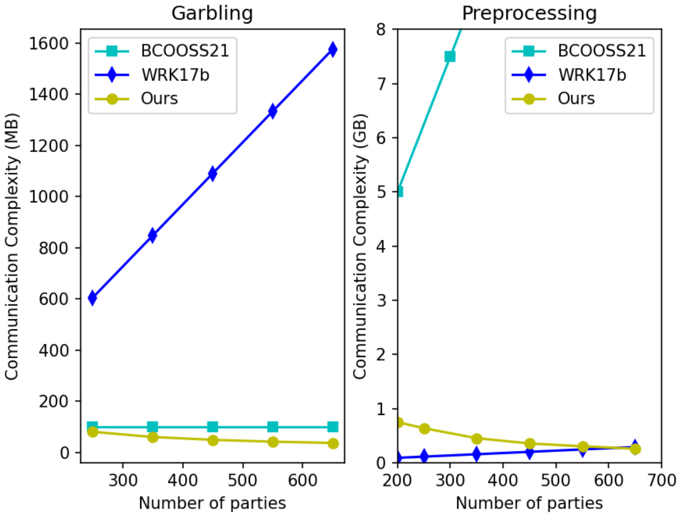

Figure 2: Communication cost per party in both the pre-processing and garbling phases for different BMR style protocols, scaling with the total number of parties n who participate. The circuit is AES-128 and in our protocol $t \approx \frac{n}{4}$ while $t > \frac{n}{2}$ for the other protocols.

In more detail, let the Reed Solomon code parameters be $[Q_r, L_r, d_r]_q$, where $d_r = Q_r - L_r + 1$ and $q \geq Q_r$ is a power of 2. Let the inner code parameters be $[Q_i, L_i, d_i]_2$ with $q = 2^{L_i}$. Then by Theorem 2, the concatenated code has parameters $[Q_rQ_i, L_rL_i, d_rd_i]_2$. Hence, we need $Q_rQ_i \geq Q$. However, when $Q_rQ_i > Q$, then we need to truncate $(Q_rQ_i - Q)$-rows of the generating matrix. This causes a loss of $(Q_rQ_i - Q)$ in the distance, and we thus obtain a $[Q, L_rL_i, d_rd_i - (Q_rQ_i - Q)]_2$-code. By Theorem 1, if we have $\lfloor Q/3 \rfloor$ malicious parties, then we need $d_rd_i - (Q_rQ_i - Q) + 1 \geq \lfloor Q/3 \rfloor$. In summary, we need to choose the parameters which maximize message length L_rL_i with the following constraints.

$$Q_r \cdot Q_i \geq Q$$
$$q = 2^{L_i} \geq Q_r$$
$$Q_r \cdot Q_i - d_r \cdot d_i \leq Q - \lfloor Q/3 \rfloor.$$

In our setting, we assume $Q = n$, i.e., the number of parties. If we use Reed Solomon Codes and the inner code with constant rate, then the resulting concetenation code will also have constant rate. In this case, it is easy to see that $L_rL_i \in O(n)$. We can now use the generator matrix of $[Q, L_rL_i, d_rd_i - (Q_rQ_i - Q)]_2$-code as our binary super-invertible matrix. The dimension of this matrix will be $Q \times L_rL_i$, i.e., $n \times O(n)$, which is what we want.

For concrete parameters, we take the BCH codes [BRC60] as the inner codes, and use a Python script to enumerate all combinations of the Reed Solomon codes and the BCH codes to find the largest possible L_r. Our script also enumerates random linear codes achieving Gilbert–Varshamov bound as the inner code. Here we list some concrete parameters. For $Q = 16$, we choose $[16, 7, 6]_2$-BCH code[11]. For $Q = 56$, we choose $[7, 3, 5]_{2^4}$-Reed Solomon code concatenated with $[8, 4, 4]_2$-BCH code. For $Q = 104$, we choose $[13, 9, 5]_{2^4}$-Reed Solomon code concatenated with $[8, 4, 4]_2$-BCH code. For $Q = 512$, we choose $[32, 4, 29]_{2^7}$-Reed Solomon code concatenated with $[16, 7, 6]_2$-BCH code.

[11] We add a parity check bit to the BCH code to slightly increase the distance by 1.

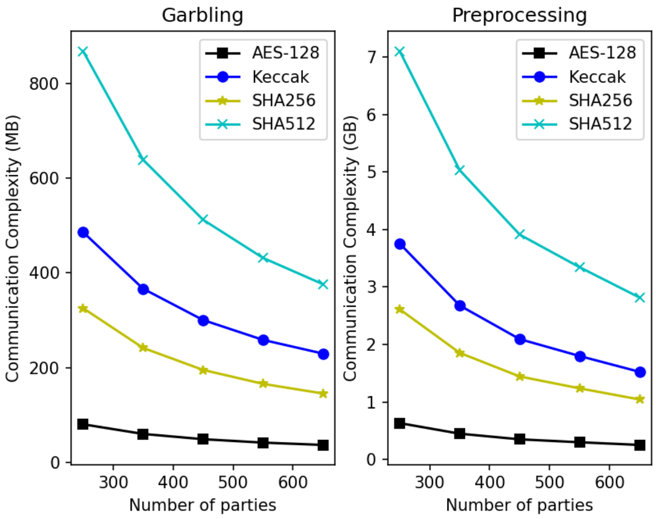

Figure 3: Communication cost per party in both the pre-processing and garbling phases for our protocol with different types of circuits, scaling with the total number of parties n who participate. The fraction of malicious parties tolerated is $t \approx \frac{n}{4}$. All circuit descriptions are taken from [AAL+].

7.4 Analysis of Communication Costs

In this work we focus on protocols that achieve good communication complexity. This is motivated by practice: unless communication among parties is done within a single data center over a network with incredibly high throughput, protocols will need to be deployed in scenarios where network resources are constrained. Significant delays can occur during protocol execution if these resources are exhausted. Therefore, we focus on calculating the communication costs of our protocol with respect to the most communication intensive phases: pre-processing and garbling phases. The cost is reported in bytes and parameters are chosen to represent realistic real world scenarios: in particular, we report numbers for common cryptographic functionalities like hash functions and symmetric key encryption with the number of parties communicating representing what can reasonably be expected from volunteer networks that exist on blockchains or through Tor. For statistical security and computational security, we use parameters of $\kappa_s = 40$ and $\kappa_c = 80$ respectively. As eluded to in Section 7.2 the LPN encryption scheme we use has a ciphertext size of 721 bits with keys of 246 bits and a noise parameter of $\tau = 2^{-2}$.

Simulating Communication To report our communication costs as accurately as possible, we write a `python` script that takes in parameters of the protocol—including the number of parties, the corruption threshold, chosen LPN parameters, and a circuit description—and then emulates the communication in each sub-protocol, recording the number of bits that each party sends to every other party in a given round. The final output is saved into a directory that contains a json file for each party along with relevant protocol details. For the sake reproducibility, the scripts used for computing communication and computation (see Section 7.5) costs have been included in a public github repository[12]. Next, we compare our communication complexity against other constant round MPC protocols.

[12] https://github.com/becgabri/sg-microbenchmarks

7.4.1 Comparison to Prior Work

The most recent work that closely resembles ours is that of [BCO+21]. In their work, the authors construct a BMR-style protocol in the dishonest majority setting which only requires $O(n)$ communication per party in the online phase and makes use of a somewhat similar LPN encryption scheme. Because their total communication complexity is $O(n^2|C|)$, we expect our protocol to require less networking resources than theirs for both pre-processing and garbling. We also compare our protocol against the authenticated garbling protocol of [WRK17b] which has a total communication complexity of $O(n|C|)$. Because communication costs is only reported for $n = 2$ to $n = 16$, we extrapolate to larger values by doing linear interpolation on the values reported in [WRK17b, Table 8]. Figure 2 compares the average communication cost per party for an AES-128 circuit. Notably, for our protocol, both garbling and pre-processing communication decreases as n grows. While for smaller numbers of parties the protocol of [WRK17b] is more efficient, our protocol begins to exhibit lower cost in the pre-processing phase around ≈ 600 parties.

Communication costs for other circuits In addition to running simulations with solely an AES circuit we also calculate communication cost for other standard circuits such as Keccak and various SHA variants. The relevant per party communication is presented in Figure 3. As with the AES circuit, communication always decreases when more parties are present. For circuits that are already costly to compute, even adding a hundred parties substantially decreases per party communication although at a certain point the benefit of adding more parties appears to be marginal.

Due to our comparisons with other efficient constant round protocols and the ability to reduce an individual party's workload as the total number of parties increases, these results suggest that not only is our protocol asymptotically efficient but that it has the potential to be concretely as well for large numbers of parties.

7.5 Analysis of Computation Costs

We analyze the concrete computation overhead of our protocol by first benchmarking the performance of computing field operations (addition and multiplication) followed by programmatically estimating the number of field operations carried out in a protocol execution. We benchmarked the performance of computing field operations using the Fast Galois Field Arithmetic Library[13] and found that addition and multiplication over $\mathrm{GF}(2^{18})$ takes 0.16 ns and 7.68 ns respectively on a machine equipped with an Intel® Xeon® CPU E5-2695 v4 @ 2.10 GHz. The estimates for the computation cost for the semi-honest and maliciously secure protocol are presented in Table 1 and Table 2 respectively. We note that SHA-512 is benchmarked over the field $\mathrm{GF}(2^{19})$.

Table 1: Computation cost for evaluating different circuits using the semi-honest garbling protocol where each party uses 8 threads.

Circuit	Pre-processing	Garbling
AES-128	1.94 min	1.64 min
Keccak	10.15 min	8.57 min
SHA-256	7.14 min	6.03 min
SHA-512	21.11 min	17.82 min

Table 2: Computation cost for evaluating different circuit using the maliciously secure garbling protocol where each party uses 8 threads.

Circuit	Pre-processing	Garbling
AES-128	20.29 min	3.91 min
Keccak	1.77 hours	20.45 min
SHA-256	1.24 hours	14.38 min
SHA-512	3.68 hours	42.52 min

Comparison To Prior Work. Ben-Efraim et al. [BCO+21] do not evaluate the computational cost of

[13] http://web.eecs.utk.edu/jplank/plank/papers/CS-07-593/

their pre-processing and garbling phases. Wang et al. [WRK17b] report a protocol runtime of 2.4 minutes for evaluating AES-128 over a WAN network with 128 parties, each running on a machine with a 2 GHz, 36 cores CPU. This is comparable to the estimated computational cost of our protocol for 200 parties – about 2.66 minutes if each party runs on a similar 36 core machine supporting hyperthreading. In some sense, this shows that the overhead of using (field) LPN (as opposed to OT, in [WRK17b]) is counteracted by our savings in communication and computation in the number of parties. Importantly, if we increase the number of parties, we should expect similar computation times and reduced communication cost for our protocol. In contrast, the performance of [WRK17b]—where *per-party* computation and communication costs grow linearly with the number of parties—should degrade with an increase in the number of parties.

Acknowledgements

Gabriel Kaptchuk is supported by the National Science Foundation under Grant #2030859 to the Computing Research Association for the CIFellows Project and is supported by DARPA under Agreement No. HR00112020021. Gabrielle Beck and Aditya Hegde were supported by DARPA under Contract No. HR001120C0084. Aarushi Goel, Aditya Hegde, Abhishek Jain and Zhengzhong Jin were supported in part by NSF CNS-1814919, NSF CAREER 1942789 and Johns Hopkins University Catalyst award. Abhishek Jain was additionally supported in part by AFOSR Award FA9550-19-1-0200, Office of Naval Research Grant N00014-19-1-2294 and research gifts from Ethereum, Stellar and Cisco. Zhengzhong Jin was additionally supported in part by DARPA under Agreement No. HR00112020023 and by an NSF grant CNS-2154149. This work was done in part when Aarushi Goel and Zhengzhong Jin were students at Johns Hopkins University. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Government or DARPA.

References

[AAB15] Benny Applebaum, Jonathan Avron, and Christina Brzuska. Arithmetic cryptography: Extended abstract. In Tim Roughgarden, editor, *ITCS 2015*, pages 143–151. ACM, January 2015.

[AAL+] David Archer, Victor Arribas Abril, Steve Lu, Pieter Maene, Nele Mertens, Danilo Sijacic, and Nigel Smart. 'bristol fashion' MPC circuits. Available at https://homes.esat.kuleuven.be/nsmart/MPC/.

[ADI+17] Benny Applebaum, Ivan Damgård, Yuval Ishai, Michael Nielsen, and Lior Zichron. Secure arithmetic computation with constant computational overhead. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 223–254. Springer, Heidelberg, August 2017.

[AIK11] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. How to garble arithmetic circuits. In Rafail Ostrovsky, editor, *52nd FOCS*, pages 120–129. IEEE Computer Society Press, October 2011.

[AJL+12] Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. Multiparty computation with low communication, computation and interaction via threshold FHE. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 483–501. Springer, Heidelberg, April 2012.

[Ale03] Michael Alekhnovich. More on average case vs approximation complexity. In *44th FOCS*, pages 298–307. IEEE Computer Society Press, October 2003.

[BCGI18] Elette Boyle, Geoffroy Couteau, Niv Gilboa, and Yuval Ishai. Compressing vector OLE. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 896–912. ACM Press, October 2018.

[BCO+21] Aner Ben-Efraim, Kelong Cong, Eran Omri, Emmanuela Orsini, Nigel P. Smart, and Eduardo Soria-Vazquez. Large scale, actively secure computation from LPN and free-XOR garbled circuits. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part III*, volume 12698 of *LNCS*, pages 33–63. Springer, Heidelberg, October 2021.

[BDOZ11] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In Kenneth G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 169–188. Springer, Heidelberg, May 2011.

[Ben18] Aner Ben-Efraim. On multiparty garbling of arithmetic circuits. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part III*, volume 11274 of *LNCS*, pages 3–33. Springer, Heidelberg, December 2018.

[BGG19] Sean Bowe, Ariel Gabizon, and Matthew D. Green. A multi-party protocol for constructing the public parameters of the pinocchio zk-SNARK. In Aviv Zohar, Ittay Eyal, Vanessa Teague, Jeremy Clark, Andrea Bracciali, Federico Pintore, and Massimiliano Sala, editors, *FC 2018 Workshops*, volume 10958 of *LNCS*, pages 64–77. Springer, Heidelberg, March 2019.

[BGG+20] Fabrice Benhamouda, Craig Gentry, Sergey Gorbunov, Shai Halevi, Hugo Krawczyk, Chengyu Lin, Tal Rabin, and Leonid Reyzin. Can a public blockchain keep a secret? In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part I*, volume 12550 of *LNCS*, pages 260–290. Springer, Heidelberg, November 2020.

[BGJK21] Gabrielle Beck, Aarushi Goel, Abhishek Jain, and Gabriel Kaptchuk. Order-C secure multiparty computation for highly repetitive circuits. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part II*, volume 12697 of *LNCS*, pages 663–693. Springer, Heidelberg, October 2021.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th ACM STOC*, pages 1–10. ACM Press, May 1988.

[BHP17] Zvika Brakerski, Shai Halevi, and Antigoni Polychroniadou. Four round secure computation without setup. In Yael Kalai and Leonid Reyzin, editors, *TCC 2017, Part I*, volume 10677 of *LNCS*, pages 645–677. Springer, Heidelberg, November 2017.

[BHR12] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012*, pages 784–796. ACM Press, October 2012.

[BKW00] Avrim Blum, Adam Kalai, and Hal Wasserman. Noise-tolerant learning, the parity problem, and the statistical query model. In *32nd ACM STOC*, pages 435–440. ACM Press, May 2000.

[BLMR13] Dan Boneh, Kevin Lewi, Hart William Montgomery, and Ananth Raghunathan. Key homomorphic PRFs and their applications. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 410–428. Springer, Heidelberg, August 2013.

[BLO16] Aner Ben-Efraim, Yehuda Lindell, and Eran Omri. Optimizing semi-honest secure multiparty computation for the internet. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 578–590. ACM Press, October 2016.

[BLO17] Aner Ben-Efraim, Yehuda Lindell, and Eran Omri. Efficient scalable constant-round MPC via garbled circuits. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part II*, volume 10625 of *LNCS*, pages 471–498. Springer, Heidelberg, December 2017.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *22nd ACM STOC*, pages 503–513. ACM Press, May 1990.

[BMR16] Marshall Ball, Tal Malkin, and Mike Rosulek. Garbling gadgets for Boolean and arithmetic circuits. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 565–577. ACM Press, October 2016.

[BNP08] Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: a system for secure multiparty computation. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *ACM CCS 2008*, pages 257–266. ACM Press, October 2008.

[BOSS20] Carsten Baum, Emmanuela Orsini, Peter Scholl, and Eduardo Soria-Vazquez. Efficient constant-round MPC with identifiable abort and public verifiability. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 562–592. Springer, Heidelberg, August 2020.

[BRC60] R.C. Bose and D.K. Ray-Chaudhuri. On a class of error correcting binary group codes. *Information and Control*, 3(1):68–79, 1960.

[BTH08] Zuzana Beerliová-Trubíniová and Martin Hirt. Perfectly-secure MPC with linear communication complexity. In Ran Canetti, editor, *TCC 2008*, volume 4948 of *LNCS*, pages 213–230. Springer, Heidelberg, March 2008.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (abstract) (informal contribution). In Carl Pomerance, editor, *CRYPTO'87*, volume 293 of *LNCS*, page 462. Springer, Heidelberg, August 1988.

[CCD$^+$20] Megan Chen, Ran Cohen, Jack Doerner, Yashvanth Kondi, Eysa Lee, Schuyler Rosefield, and abhi shelat. Multiparty generation of an RSA modulus. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part III*, volume 12172 of *LNCS*, pages 64–93. Springer, Heidelberg, August 2020.

[CGG$^+$21] Arka Rai Choudhuri, Aarushi Goel, Matthew Green, Abhishek Jain, and Gabriel Kaptchuk. Fluid MPC: Secure multiparty computation with dynamic participants. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part II*, volume 12826 of *LNCS*, pages 94–123, Virtual Event, August 2021. Springer, Heidelberg.

[CGH$^+$18] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority MPC for malicious adversaries. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 34–64. Springer, Heidelberg, August 2018.

[CHI$^+$20] Megan Chen, Carmit Hazay, Yuval Ishai, Yuriy Kashnikov, Daniele Micciancio, Tarik Riviere, abhi shelat, Muthu Venkitasubramaniam, and Ruihan Wang. Diogenes: Lightweight scalable RSA modulus generation with a dishonest majority. Cryptology ePrint Archive, Report 2020/374, 2020. https://eprint.iacr.org/2020/374.

[DGN$^+$17] Nico Döttling, Satrajit Ghosh, Jesper Buus Nielsen, Tobias Nilges, and Roberto Trifiletti. TinyOLE: Efficient actively secure two-party computation from oblivious linear function evaluation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 2263–2276. ACM Press, October / November 2017.

[DI05] Ivan Damgård and Yuval Ishai. Constant-round multiparty computation using a black-box pseudorandom generator. In Victor Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 378–394. Springer, Heidelberg, August 2005.

[DIK$^+$08] Ivan Damgård, Yuval Ishai, Mikkel Krøigaard, Jesper Buus Nielsen, and Adam Smith. Scalable multiparty computation with nearly optimal work and resilience. In David Wagner, editor, *CRYPTO 2008*, volume 5157 of *LNCS*, pages 241–261. Springer, Heidelberg, August 2008.

[DIK10] Ivan Damgård, Yuval Ishai, and Mikkel Krøigaard. Perfectly secure multiparty computation and the computational overhead of cryptography. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 445–465. Springer, Heidelberg, May / June 2010.

[DKL$^+$13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS 2013*, volume 8134 of *LNCS*, pages 1–18. Springer, Heidelberg, September 2013.

[DN07] Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In Alfred Menezes, editor, *CRYPTO 2007*, volume 4622 of *LNCS*, pages 572–590. Springer, Heidelberg, August 2007.

[DOS18] Ivan Damgård, Claudio Orlandi, and Mark Simkin. Yet another compiler for active security or: Efficient MPC over arbitrary rings. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 799–829. Springer, Heidelberg, August 2018.

[Döt15] Nico Döttling. Low noise LPN: KDM secure public key encryption and sample amplification. In Jonathan Katz, editor, *PKC 2015*, volume 9020 of *LNCS*, pages 604–626. Springer, Heidelberg, March / April 2015.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, Heidelberg, August 2012.

[DSZ15] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *NDSS 2015*. The Internet Society, February 2015.

[EKM17] Andre Esser, Robert Kübler, and Alexander May. LPN decoded. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part II*, volume 10402 of *LNCS*, pages 486–514. Springer, Heidelberg, August 2017.

[FL19] Jun Furukawa and Yehuda Lindell. Two-thirds honest-majority MPC for malicious adversaries at almost the cost of semi-honest. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 1557–1571. ACM Press, November 2019.

[FY92] Matthew K. Franklin and Moti Yung. Communication complexity of secure computation (extended abstract). In *24th ACM STOC*, pages 699–710. ACM Press, May 1992.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *41st ACM STOC*, pages 169–178. ACM Press, May / June 2009.

[GGJS12] Sanjam Garg, Vipul Goyal, Abhishek Jain, and Amit Sahai. Concurrently secure computation in constant rounds. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 99–116. Springer, Heidelberg, April 2012.

[GHK+21] Craig Gentry, Shai Halevi, Hugo Krawczyk, Bernardo Magri, Jesper Buus Nielsen, Tal Rabin, and Sophia Yakoubov. YOSO: You only speak once - secure MPC with stateless ephemeral roles. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part II*, volume 12826 of *LNCS*, pages 64–93, Virtual Event, August 2021. Springer, Heidelberg.

[GIP+14] Daniel Genkin, Yuval Ishai, Manoj Prabhakaran, Amit Sahai, and Eran Tromer. Circuits resilient to additive attacks with applications to secure computation. In David B. Shmoys, editor, *46th ACM STOC*, pages 495–504. ACM Press, May / June 2014.

[GIP15] Daniel Genkin, Yuval Ishai, and Antigoni Polychroniadou. Efficient multi-party computation: From passive to active security via secure SIMD circuits. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 721–741. Springer, Heidelberg, August 2015.

[GLS15] S. Dov Gordon, Feng-Hao Liu, and Elaine Shi. Constant-round MPC with fairness and guarantee of output delivery. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 63–82. Springer, Heidelberg, August 2015.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987.

[Gol04] Oded Goldreich. *Foundations of Cryptography: Basic Applications*, volume 2. Cambridge University Press, Cambridge, UK, 2004.

[GPS21] Vipul Goyal, Antigoni Polychroniadou, and Yifan Song. Unconditional communication-efficient MPC via hall's marriage theorem. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part II*, volume 12826 of *LNCS*, pages 275–304, Virtual Event, August 2021. Springer, Heidelberg.

[GPS22] Vipul Goyal, Antigoni Polychroniadou, and Yifan Song. Sharing transformation and dishonest majority MPC with packed secret sharing. Cryptology ePrint Archive, Report 2022/831, 2022. https://eprint.iacr.org/2022/831.

[GS20] Vipul Goyal and Yifan Song. Malicious security comes free in honest-majority MPC. Cryptology ePrint Archive, Report 2020/134, 2020. https://eprint.iacr.org/2020/134.

[GSY21] S. Dov Gordon, Daniel Starin, and Arkady Yerukhimovich. The more the merrier: Reducing the cost of large scale MPC. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part II*, volume 12697 of *LNCS*, pages 694–723. Springer, Heidelberg, October 2021.

[GSZ20] Vipul Goyal, Yifan Song, and Chenzhi Zhu. Guaranteed output delivery comes free in honest majority MPC. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 618–646. Springer, Heidelberg, August 2020.

[HHNZ19] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. SoK: General purpose compilers for secure multi-party computation. In *2019 IEEE Symposium on Security and Privacy*, pages 1220–1237. IEEE Computer Society Press, May 2019.

[HOSS18a] Carmit Hazay, Emmanuela Orsini, Peter Scholl, and Eduardo Soria-Vazquez. Concretely efficient large-scale MPC with active security (or, TinyKeys for TinyOT). In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part III*, volume 11274 of *LNCS*, pages 86–117. Springer, Heidelberg, December 2018.

[HOSS18b] Carmit Hazay, Emmanuela Orsini, Peter Scholl, and Eduardo Soria-Vazquez. TinyKeys: A new approach to efficient multi-party computation. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 3–33. Springer, Heidelberg, August 2018.

[HSS17] Carmit Hazay, Peter Scholl, and Eduardo Soria-Vazquez. Low cost constant round MPC combining BMR and oblivious transfer. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 598–628. Springer, Heidelberg, December 2017.

[HSS20] Carmit Hazay, Peter Scholl, and Eduardo Soria-Vazquez. Low cost constant round MPC combining BMR and oblivious transfer. *Journal of Cryptology*, 33(4):1732–1786, October 2020.

[IPS09] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Secure arithmetic computation with no honest majority. In Omer Reingold, editor, *TCC 2009*, volume 5444 of *LNCS*, pages 294–314. Springer, Heidelberg, March 2009.

[JLS21] Aayush Jain, Huijia Lin, and Amit Sahai. *Indistinguishability Obfuscation from Well-Founded Assumptions*, page 60–73. Association for Computing Machinery, New York, NY, USA, 2021.

[KOS16] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 830–842. ACM Press, October 2016.

[KPR18] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ great again. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 158–189. Springer, Heidelberg, April / May 2018.

[LN17] Yehuda Lindell and Ariel Nof. A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 259–276. ACM Press, October / November 2017.

[LPSY15] Yehuda Lindell, Benny Pinkas, Nigel P. Smart, and Avishay Yanai. Efficient constant round multi-party computation combining BMR and SPDZ. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 319–338. Springer, Heidelberg, August 2015.

[LSS16] Yehuda Lindell, Nigel P. Smart, and Eduardo Soria-Vazquez. More efficient constant-round multi-party computation from BMR and SHE. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B, Part I*, volume 9985 of *LNCS*, pages 554–581. Springer, Heidelberg, October / November 2016.

[Lyu05] Vadim Lyubashevsky. The parity problem in the presence of noise, decoding random linear codes, and the subset sum problem. In Chandra Chekuri, Klaus Jansen, José D. P. Rolim, and Luca Trevisan, editors, *Approximation, Randomization and Combinatorial Optimization. Algorithms and Techniques*, pages 378–389, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[MNPS04] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay - secure two-party computation system. In Matt Blaze, editor, *USENIX Security 2004*, pages 287–302. USENIX Association, August 2004.

[MR18] Payman Mohassel and Peter Rindal. ABY3: A mixed protocol framework for machine learning. Cryptology ePrint Archive, Report 2018/403, 2018. https://eprint.iacr.org/2018/403.

[MSs11] Steven Myers, Mona Sergi, and abhi shelat. Threshold fully homomorphic encryption and secure computation. Cryptology ePrint Archive, Report 2011/454, 2011. https://eprint.iacr.org/2011/454.

[MW16] Pratyay Mukherjee and Daniel Wichs. Two round multiparty computation via multi-key FHE. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 735–763. Springer, Heidelberg, May 2016.

[MW19] Eleftheria Makri and Tim Wood. Full-threshold actively-secure multiparty arithmetic circuit garbling. Cryptology ePrint Archive, Report 2019/1098, 2019. https://eprint.iacr.org/2019/1098.

[NST17] Jesper Buus Nielsen, Thomas Schneider, and Roberto Trifiletti. Constant round maliciously secure 2PC with function-independent preprocessing using LEGO. In *NDSS 2017*. The Internet Society, February / March 2017.

[NV18] Peter Sebastian Nordholt and Meilof Veeningen. Minimising communication in honest-majority MPC by batchwise multiplication verification. In Bart Preneel and Frederik Vercauteren, editors, *ACNS 18*, volume 10892 of *LNCS*, pages 321–339. Springer, Heidelberg, July 2018.

[OSV20] Emmanuela Orsini, Nigel P. Smart, and Frederik Vercauteren. Overdrive2k: Efficient secure MPC over $\mathbb{Z}_{2^k}$ from somewhat homomorphic encryption. In Stanislaw Jarecki, editor, *CT-RSA 2020*, volume 12006 of *LNCS*, pages 254–283. Springer, Heidelberg, February 2020.

[Pra62] E. Prange. The use of information sets in decoding cyclic codes. *IRE Transactions on Information Theory*, 8(5):5–9, 1962.

[QLJ+19] Lucy Qin, Andrei Lapets, Frederick Jansen, Peter Flockhart, Kinan Dak Albab, Ira Globus-Harris, Shannon Roberts, and Mayank Varia. From usability to secure computing and back again. Cryptology ePrint Archive, Report 2019/734, 2019. https://eprint.iacr.org/2019/734.

[Sha79] Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979.

[WJS+19] Ryan Wails, Aaron Johnson, Daniel Starin, Arkady Yerukhimovich, and S. Dov Gordon. Stormy: Statistics in tor by measuring securely. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 615–632. ACM Press, November 2019.

[WRK17a] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 21–37. ACM Press, October / November 2017.

[WRK17b] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 39–56. ACM Press, October / November 2017.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167. IEEE Computer Society Press, October 1986.

[Zic17] Lior Zichron. Locally computable arithmetic pseudorandom generators. *Master's thesis, School of Electrical Engineering, Tel Aviv University*, 2017.