Vespa: Efficient Secure Aggregation with Integrity Defense under Server Privacy

Jia Hu[*,§], Hongwei Li[*✉], Meng Hao[†,✉], Hanxiao Chen[*], Pengzhi Xing[*], Wenbo Jiang[*], Dongxiao Liu[*], Haiyang Xue[†], and Robert H. Deng[†]
[*]University of Electronic Science and Technology of China
[†]Singapore Management University
{*jiahu, p.xing*}*@std.uestc.edu.cn,* {*menghao303, chenhanxiao.chx*}*@gmail.com,*
{*hongweili, wenbo_jiang, dongxiao.liu*}*@uestc.edu.cn,* {*haiyangxue, robertdeng*}*@smu.edu.sg*

***Abstract*—Secure aggregation with input validation enables a server to securely compute the summation of private data from multiple clients while verifying that the inputs satisfy specific constraints. However, the practical deployment of this paradigm at scale is hindered by two primary limitations. First, existing schemes suffer from severe efficiency bottlenecks due to their reliance on generic, computationally expensive zero-knowledge proofs (ZKPs). Second, the scope of input validation is currently restricted to norm checks (e.g., L_2 and L_∞) against public bounds.**

This paper presents Vespa, a single-server secure aggregation scheme that simultaneously achieves practical efficiency and expands validation capabilities to support both *norm checks against public bounds* and *similarity checks against private server inputs*. Specifically, we design a novel validity-weighted secure aggregation protocol over committed inputs, built upon highly efficient vector oblivious linear evaluation (VOLE). To efficiently realize input validation, we introduce customized L_2 and L_∞ norm checks based on VOLE-based ZKPs, centered around optimized range proofs with reduced range sizes. Furthermore, we extend our framework to support advanced similarity-based checks against private server inputs, instantiated with widely used Euclidean and cosine similarities. This allows Vespa to first filter out invalid inputs using norm checks, followed by performing fine-grained validation via similarity metrics. We compare our scheme with the state-of-the-art secure aggregation works, including RoFL (S&P 2023), ACORN (USENIX Security 2023), and Armadillo (CCS 2025), and extensive evaluation shows that our L_2 and L_∞ norm checks achieve runtime improvements of up to $2 \sim 3$ and $1 \sim 3$ orders of magnitude, respectively. Our similarity checks also achieve practical performance, consuming only around 94 seconds for an 818k-entry vector.

I. Introduction

Secure aggregation [1], [2], [3], [4], [5], [6], [7], [8] allows a server to compute aggregate statistics over sensitive data from multiple clients without exposing individual inputs. Because of this strong privacy guarantee, it has seen widespread deployment in real-world applications, including federated learning for aggregating local model updates [3], [5], [4], [9], [1] and private telemetry for collecting browser and device statistics [10], [11], [12]. However, while these protocols successfully resolve *input privacy* concerns of clients, they inadvertently create a blind spot for the server regarding *input integrity*. As demonstrated by recent attacks [13], [14], [15], [3], [5], malicious clients can inject poisoned or manipulated inputs to compromise the reliability of the aggregation result.

To simultaneously achieve input privacy and integrity, a promising line of research explores *secure aggregation with input validation* [3], [5], [4], [11], [10], [2], [16], [17]. These systems introduce customized aggregation protocols that enable *norm checks* on client inputs using zero-knowledge proofs (ZKPs). Specifically, EIFFeL [5] achieves input validation by having clients act as mutual verifiers for each other under the server's supervision. Unfortunately, this design leads to quadratic complexity in terms of the number of clients, making it impractical for large-scale deployments. To avoid such a quadratic complexity, RoFL [3] integrates ElGamal-based commitments [18] with Bulletproofs [19] to verify L_2 and L_∞ norms. Building upon this, ACORN [4] and Armadillo [17] advance the paradigm by shifting to RLWE-based secure aggregation [20] and utilizing Lagrange's four-square theorem [21] to enforce these norm constraints.

However, two fundamental issues remain in existing schemes:

- *Efficiency bottlenecks.* Existing works either introduce quadratic complexity in terms of the number of clients (e.g., EIFFeL [5]) or incur prohibitively high computational overhead due to expensive zero-knowledge proofs. Specifically, protocols like RoFL [3], ACORN [4], and Armadillo [17] rely on Bulletproofs-based inner-product arguments [19] that proceed recursively over the witness. While this yields compact proofs, the complex arithmetization and recursive constraint evaluation make proof generation computationally expensive for high-dimensional input vectors. For example, ACORN [4] consumes around 4 minutes for an 818k-entry vector.
- *Limited integrity defenses.* Current state-of-the-art systems [3], [4], [17], [22] restrict input validation to basic L_2 and L_∞ norm checks. By construction, these norm-bounded defenses cannot detect sign-flipping attacks [23], [24], since negating the sign of every entry preserves both the L_2 and L_∞ norms. Similarity-based defenses [13], [25], [9] address this gap: each client's input is validated against a server-private reference vector by using direction-sensitive measures such as cosine similarity, so that a sign-flipped input is rejected because it points in

✉Corresponding author.
§This work was done at SMU as a visiting student.

the opposite direction of the reference. Unfortunately, the cryptographic models of existing schemes cannot accommodate *server privacy*: their zero-knowledge proofs assume the verifier, i.e., the server, holds no private input.

A. Our Contributions

Motivated by addressing the two issues, we propose Vespa (VOLE-ZK-based Efficient Secure server-Privacy-preserving Aggregation), a single-server secure aggregation framework that achieves high efficiency and comprehensive input integrity defense against malicious clients, supported by optimized norm checks and server-privacy-preserving similarity checks. At its core, Vespa follows a layered architecture: norm checks act as a lightweight pre-filter that eliminates clearly malformed inputs at low cost, a server-privacy-preserving similarity check provides fine-grained validation against the server's private reference, and the surviving inputs are then securely aggregated. We summarize and compare the security and efficiency properties of our framework with closely related works in Table I. Our contributions can be summarized as follows:

- We propose Vespa, an efficient single-server secure aggregation framework that simultaneously supports input privacy, input integrity, and server privacy through a validity-weighted aggregation pipeline in which integrity checks act as input filters. (Refer to Section III-A.)
- We design efficient L_2 and L_∞ norm check protocols based on optimized range proofs with reduced range sizes, and server-privacy-preserving similarity check protocols instantiated with Euclidean and cosine similarities. (Refer to Section III-B and III-C.)
- We conduct extensive experiments showing that our L_2 and L_∞ check protocols achieve substantial efficiency improvements over prior schemes. Specifically, our L_2 and L_∞ norm checks outperform RoFL [3] in runtime by up to $390\times$ and $60\times$, ACORN [4] by up to $900\times$ and $150\times$, and Armadillo [17] by up to $120\times$ and $400\times$. (Refer to Section VII.)

B. Related Works

Efficient Secure Aggregation. Early secure aggregation (SA) research primarily addressed clients' input privacy, communication efficiency, and dropout resilience. Bonawitz et al. [1] established the foundation using double masking to ensure privacy and robustness to client dropout. Bell et al. [7] further optimize communication and computation to polylogarithmic overhead. More recent works target system-level efficiency: Flamingo [26] and Mario [27] reduce unmasking complexity through reusable masks and multi-aggregator support, while Whisper [12] introduces streaming and sketching techniques for large-scale analytics; Aion [22] optimizes the synchronization of training rounds to minimize idle time[1]. A separate line of one-shot SA schemes such as Willow [28] and OPA [29] eliminates multiple interaction rounds, allowing clients to go offline immediately after their single contribution. More recently, Heli [30] introduces an asymmetric two-server model in which only the "heavy" server performs per-run work proportional to the number of clients, while the "light" server's per-run work is sublinear; this asymmetry enables a computationally limited party to serve as the second server in deployments with millions of clients. These schemes mainly focus on the efficiency and dropout-resilience and thus are also known as Private Aggregation because they do not verify input integrity rigorously.

Integrity-Aware Secure Aggregation. To defend against malicious clients, a line of works incorporate integrity checks into the SA process. Prio [10] introduces Secret-Shared Non-Interactive Proofs (SNIP) to enable servers to verify the well-formedness of client inputs without compromising privacy, and Prio+ [11] reduces overhead via boolean secret sharing. Building on these concepts, EIFFeL [5] adapts SNIP to the single-server setting using verifiable secret shares to support generic input validation, with clients acting as mutual verifiers under the server's supervision. ELSA [2] achieves an L_2 check and a relaxed L_∞ check in the two-server setting using client-assisted oblivious transfer. RoFL [3] uses ElGamal commitments and Bulletproofs [19] to validate L_2 and L_∞ norm bounds, while ACORN [4] combines RLWE-based encoding with Bulletproofs and recasts norm constraints into inner-product proofs via Lagrange's four-square theorem [21]. Armadillo [17] also uses Lagrange's four-square theorem and Bulletproofs-based inner-product proofs, but focuses on disruption resistance: the server obtains the correct sum regardless of how clients drop out or attempt to actively disrupt the aggregation. LZKSA [31] applies RLWE to norm verification but we identify security issues in its design[2] and discuss it in Appendix H. Our work falls into this category but is the first to additionally support server privacy, enabling similarity-based integrity checks against a server-private reference vector.

II. Preliminaries

A. Notation

Let κ and λ be the computational and statistical security parameters, respectively. We denote vectors using bold letters (e.g., $\mathbf{x}$), with $\mathbf{x}[i]$ or x_i representing the i-th element in $\mathbf{x}$. The ℓ_2 norm of $\mathbf{x}$ is denoted by $\|\mathbf{x}\|_2$ and the element-wise absolute value by $|\mathbf{x}|$. Vector operations include the element-wise (Hadamard) product $\mathbf{x}*\mathbf{y}$; the inner product $\mathbf{x}\cdot\mathbf{y}$; and the scalar-vector multiplication $x\cdot\mathbf{y}$. For $a, b \in \mathbb{Z}$ with $a < b$, we define $[a, b] := \{a, \ldots, b\}$ and $[a] := \{0, \ldots, a-1\}$. $1\{a < b\}$ equals 1 if $a < b$ is true otherwise 0.

Fixed-Point Representation. Following prior works [33], [34], [35], we use a fixed-point encoding to represent a real number $\hat{x} \in \mathbb{R}$ as a field element $x \in \mathbb{F}_p$, parameterized by a scale s for fractional precision. The conversions are defined as follows:

[1] Although Aion [22] includes an approximate norm check, it does not provide rigorous integrity verification and is therefore grouped here.

[2] Karthikeyan et al. also identify the security issue in LZKSA [31] and report it in [32].

TABLE I: Comparison of our scheme with related works. Big-O notation $O(\cdot)$ is omitted for brevity. d denotes the input vector size, and n is the number of clients. R and C denote the Regular clients and Committee members, and m is the number of committee members. L_∞^- denotes a relaxed L_∞ norm check with a special power-of-2 bound in ELSA. The runtime costs for the L_∞ norm check and L_2 norm check are measured under input sizes of 1M. *Values are estimated from the original paper because source codes are unavailable.

Schemes	Single server	Server privacy	Client comm.	Client comp.	Server comm.	Server comp.	Input validation	Runtime (L_2 norm)	Runtime (L_∞ norm)
ELSA [2]	×	×	d	d	dn	dn	L_2, L_∞^-	65.28 s	39.20 s
EIFFeL [5]	✓	×	dn^2	dn^2	dn^3	dn^3	Generic	>414.72 s*	>424.44 s*
RoFL [3]	✓	×	$d+\log n$	$d\log n$	$dn+n\log n$	dn	L_2, L_∞	186.56 s	145.12 s
ACORN [4]	✓	×	$d+\log n$	$d\log n$	$dn+n\log n$	dn	L_2, L_∞	>276.40 s*	>276.40 s*
Armadillo [17]	✓	×	R: $d+m$ C: $n+m$	R: $d+m$ C: $n+m$	$dn+mn$	$dn+mn$	L_2, L_∞	158.72 s*	981.76 s*
Ours	✓	✓	$d+\log n$	$d\log n$	$dn+n\log n$	dn	L_2, L_∞, Euc, Cos	0.32 s	2.10 s

- $\mathsf{ToField} : \mathbb{R} \to \mathbb{F}_p$, expressed as $x = \mathsf{ToField}(\hat{x}, p, s) = \lfloor \hat{x} \cdot 2^s \rfloor \bmod p$.
- $\mathsf{ToReal} : \mathbb{F}_p \to \mathbb{R}$, expressed as $\hat{x} = \mathsf{ToReal}(x, p, s) = (x - a \cdot p)/2^s$, where $a = 1\{x > (p-1)/2\}$, and the operations are performed in $\mathbb{R}$.

In our work, clients' real-valued inputs are first mapped to the field using ToField, and the final aggregated result is mapped back to a real number using ToReal.

B. Threat Model and Security Goals

1) Threat Model: We consider an adversary that can either corrupt a subset of clients maliciously or corrupt the server, but not both.

- **Malicious clients.** The adversary can corrupt up to $n-2$ clients and arbitrarily deviate from the protocol on their behalf, e.g., manipulating input vectors to compromise the correctness of the aggregation result.
- **Malicious server.** The adversary can corrupt the server and arbitrarily deviate from the protocol, attempting to extract information about honest clients' inputs.

The threat model varies slightly across the protocol components. Our norm check protocols achieve full malicious security on both sides, consistent with the setting of prior single-server works [3], [4], [17] and allowing for a fair comparison. Our similarity check protocols achieve malicious-client security with semi-honest server, following the standard threat model of prior secure two-party computation literature for achieving malicious-client security at semi-honest cost [36], [37]. A formal specification of the threat model and the corresponding security definitions is provided in Appendix B.

2) Security Goals: Our work aims to achieve the following three properties:

- **Input Privacy.** For an honest client $\mathcal{C}_h$ with input vector $\mathbf{x}_h$, neither the server nor any other client learns anything about $\mathbf{x}_h$ beyond what is implied by the aggregation result.
- **Input Integrity.** The server $\mathcal{S}$ verifies that each client's input vector satisfies a specified validation predicate (e.g., L_2 or L_∞ norm bounds, or similarity bounds against a reference vector), and inputs failing the predicate are excluded from the aggregation. This prevents malicious clients from manipulating the aggregation result through ill-formed inputs.
- **Server Privacy.** The server's reference vector $\mathbf{x}_s$ remains hidden from all clients throughout the protocol; clients learn no information about $\mathbf{x}_s$ beyond what can be implied by their output.

3) Single-server and Two-server Models: Most integrity-protected and verifiable secure aggregation protocols follow two main paradigms with distinct trade-offs:

- **Single-server model.** A central server coordinates clients using cryptographic primitives such as canceling masks [1], [38], [3], [4] to hide their inputs. This model relies on a single trusted aggregator and aligns with standard FL architectures, but can incur higher cryptographic overhead.
- **Two-server model.** Clients secret-share their data between two non-colluding servers [2], [11], [10], allowing lightweight primitives at the client side [11], [10]. The drawbacks are a non-collusion assumption[3] between the two server-side parties, and deployment challenges due to coordinating different trust domains [39].

We opt for the single-server model for its superior security and alignment with deployed FL systems. Our work addresses its primary drawback, the expensive cryptographic overhead, by designing efficient norm and similarity check protocols.

C. Cryptography Primitives

1) Information-theoretic MACs: We use information-theoretic message authentication codes (IT-MACs) [40], [41],

[3]As noted by ELSA [2], the distributed trust of the two-server model is a weaker notion of security compared to the single-aggregator model.

Macro $\mathsf{Auth}(\{x_h\}_{h\in[n]})$

On input $x_h \in \mathbb{F}_p$, $h \in [n]$, where $p \geq 2^\lambda$, this subroutine interacts with $n+1$ parties $\{\mathcal{C}_h\}_{h\in[n]}$ and $\mathcal{S}$, and generates authenticated values $[x_h]_p$ for the parties.

1) For $h \in [n]$, if $\mathcal{S}$ is honest, sample $K_h \leftarrow \mathbb{F}_p$; otherwise, receive $K_h \in \mathbb{F}_p$ from the adversary.
2) For $h \in [n]$, if $\mathcal{C}_h$ is honest, compute $M_h := K_h + x_h \cdot \Delta \in \mathbb{F}_p$; otherwise, receive $M_h \in \mathbb{F}_p$ from the adversary and recompute $K_h := M_h - x_h \cdot \Delta \in \mathbb{F}_p$.
3) Output $[x_h]_p$ to the parties, i.e., send (x_h, M_h) to $\mathcal{C}_h$ and $\{K_h\}_{h\in[n]}$ to $\mathcal{S}$.

Fig. 1: Macro for generating authenticated values.

[42], [43], [33] to authenticate values in a field $\mathbb{F}_p$. The setup involves a uniform global key $\Delta \in \mathbb{F}_p$ known only to the verifier $\mathcal{S}$. An IT-MAC authentication on a message $x \in \mathbb{F}_p$ is a commitment held jointly by the prover $\mathcal{C}$ and the verifier $\mathcal{S}$. Specifically, the prover $\mathcal{C}$ holds $x \in \mathbb{F}_p$ and a MAC tag $M_x \in \mathbb{F}_p$, while the verifier $\mathcal{S}$ holds a uniform local key $K_x \in \mathbb{F}_p$, such that $M_x = K_x + \Delta \cdot x$ over $\mathbb{F}_p$. For brevity, we denote this shared commitment as $[x]_p$ and refer to IT-MAC authentications as commitments hereafter.

IT-MACs are additively homomorphic, meaning both parties can locally compute linear combinations of commitments: given public constants $c_0, c_1, \ldots, c_n \in \mathbb{F}_p$ and commitments $[x_1]_p, \ldots, [x_n]_p$, they can compute $[y]_p := c_0 + \sum_{i\in[1,n]} c_i \cdot [x_i]_p$. To open a commitment $[x]_p$, the prover $\mathcal{C}$ send (x, M_x) to the verifier $\mathcal{S}$, who checks $M_x = K_x + \Delta \cdot x$ holds. The probability of forging an incorrect x is at most $1/|\mathbb{F}_p|$. A specialized batch operation, $\mathsf{CheckZero}$, is used to efficiently open n all zero values $[x_1]_p, \ldots, [x_n]_p$. In this case, $\mathcal{C}$ sends a hash $h := \mathsf{H}(M_{x_1}, \ldots, M_{x_n})$ to $\mathcal{S}$, who checks whether $h = \mathsf{H}(K_{x_1}, \ldots, K_{x_n})$, where $\mathsf{H} : \{0,1\}^* \to \{0,1\}^\kappa$ is a hash function modeled as a random oracle. The soundness error of $\mathsf{CheckZero}$ is at most $1/|\mathbb{F}_p| + q/2^\kappa$, where q is the number of queries to the random oracle [44]. Furthermore, random IT-MAC commitments can be generated with sublinear communication complexity using recent vector oblivious linear evaluation (VOLE) protocols [45].

2) Zero-Knowledge Proofs based on VOLE: We utilize VOLE-based interactive designated-verifier ZK proofs [42], [43], [46], [47], [48], [49], [33], [35], which belong to the commit-and-prove paradigm and utilize IT-MACs for commitments. To adopt VOLE-ZK in our n-client-1-server secure aggregation setting, we define a VOLE-ZK ideal functionality $\mathcal{F}_{\mathsf{nZK}}$ with n provers and 1 verifier in Figure 2. In this formulation, clients $\{\mathcal{C}_h\}_{h\in[n]}$ act as provers, and the server $\mathcal{S}$ acts as verifier. Specifically, the functionality $\mathcal{F}_{\mathsf{nZK}}$ lets the verifier utilize the same random global key Δ to authenticate values of all provers. Moreover, except for standard circuit satisfiability to verify the correctness of a circuit execution, we include range checks in the functionality $\mathcal{F}_{\mathsf{nZK}}$, which verifies

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

This functionality is parameterized by a prime $p \geq 2^\lambda$, involves n provers $\{\mathcal{C}_h\}_{h\in[n]}$ and a verifier $\mathcal{S}$, and can invoke a macro $\mathsf{Auth}()$ defined in Figure 1.

Initialize: On receiving (Init) from $\{\mathcal{C}_h\}_{h\in[n]}$ and $\mathcal{S}$, sample $\Delta \leftarrow \mathbb{F}_p$ if $\mathcal{S}$ is honest, and receive $\Delta \in \mathbb{F}_p$ from adversary otherwise. Store Δ and send it to $\mathcal{S}$, and ignore all subsequent (Init) commands.

Input: On receiving $\{(\mathsf{Input}, x_h)\}_{h\in[n]}$ from $\{\mathcal{C}_h\}_{h\in[n]}$ and (Input) from $\mathcal{S}$, where $x_h \in \mathbb{F}_p$, execute $\mathsf{Auth}(\{x_h\}_{h\in[n]})$ so that $\mathcal{C}_h$ and $\mathcal{S}$ obtain $[x_h]_p$.

CirEval: On receiving $\{(\mathsf{CirEval}, \mathsf{C}, [x_h]_p)\}_{h\in[n]}$ from $\{\mathcal{C}_h\}_{h\in[n]}$ and $\mathcal{S}$, where $x_h \in \mathbb{F}_p$ and C is a circuit, check if $[x_h]_p$ is valid and $\mathsf{C}(x_h) = 1$ for $h \in [n]$, and output abort to $\mathcal{S}$ if any check fails; otherwise output success.

RangeCheck: On receiving $\{(\mathsf{RangeCheck}, [a, b], [x_h]_p)\}_{h\in[n]}$ from $\{\mathcal{C}_h\}_{h\in[n]}$ and $\mathcal{S}$, for all $h \in [n]$, check if $[x_h]_p$ is valid and $x_h \in [a, b]$, and output abort to $\mathcal{S}$ if any check fails; otherwise output success.

Fig. 2: Ideal functionality for ZK proofs with n provers and 1 verifier.

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

This functionality is parameterized by corruption rate η, dropout rate δ, and the number of participating clients n.

- $\mathcal{F}_{\mathsf{PA}}$ receives from a malicious adversary $\mathcal{A}$ a set of corrupted parties, denoted as $\mathcal{C} \subset [n]$, where $|\mathcal{C}|/n \leq \eta$.
- $\mathcal{F}_{\mathsf{PA}}$ receives a dropout set $\mathcal{O} \subset [n]$, where $|\mathcal{O}|/n \leq \delta$, and $|\mathcal{C}|/n \leq \eta$, and inputs $\mathbf{x}_i$ from client $i \in [n] \setminus (\mathcal{O} \cup \mathcal{C})$.
- $\mathcal{F}_{\mathsf{PA}}$ sends $\mathcal{O}$ to $\mathcal{A}$, and asks $\mathcal{A}$ for a set M: if $\mathcal{A}$ replies with $M \subseteq [n] \setminus \mathcal{O}$ such that $|M|/n \geq 1 - \delta$, then $\mathcal{F}_{\mathsf{PA}}$ computes $\mathbf{y} = \sum_{i\in M\setminus\mathcal{C}} \mathbf{x}_i$ and continues; otherwise $\mathcal{F}_{\mathsf{PA}}$ sends abort to all the honest parties.
- $\mathcal{F}_{\mathsf{PA}}$ asks $\mathcal{A}$ for a shift $\mathbf{a}$ and outputs $\mathbf{y} + \mathbf{a}$ to the server.

Fig. 3: Ideal functionality for private aggregation.

if a value falls within a specified range. For detailed VOLE-ZK protocols and proofs for $\mathcal{F}_{\mathsf{nZK}}$, please refer to Theorem 2 and 3 in Quicksilver [43] and Theorem 1 in [50]).

D. Private Aggregation Protocol

Our framework relies on a private aggregation functionality $\mathcal{F}_{\mathsf{PA}}$ as a sub-routine, formalized in Figure 3. At a high level, $\mathcal{F}_{\mathsf{PA}}$ takes vector inputs from n clients and reveals to the server only the aggregate sum over a surviving subset of clients.

We treat $\mathcal{F}_{\mathsf{PA}}$ as a black-box ideal functionality: any existing single-server private aggregation protocol that securely realizes $\mathcal{F}_{\mathsf{PA}}$ can be plugged into our framework. Concrete protocols realizing this functionality include the original construction by Bonawitz et al. [1] based on pairwise masking and Shamir secret sharing, the polylogarithmic-overhead variant by Bell et al. [7], and recent one-shot schemes such as OPA [29] and Willow [28] that further reduce client interaction.

In our framework, $\mathcal{F}_{\mathsf{PA}}$ is invoked to aggregate the validity-weighted client shares $\{\mathbf{y}_{h,c}\}_{h\in\mathcal{U}}$ together with their IT-MAC tags $\{\mathbf{M}_h\}_{h\in\mathcal{U}}$, yielding the aggregate sum $\mathbf{y}_c$ and aggregate MAC $\mathbf{M}$ used in the final consistency check (see Section IV).

III. Technical Overview

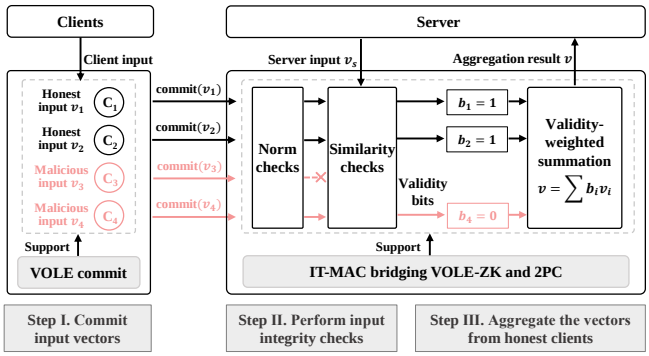


Fig. 4: A high-level overview of Vespa.

Our goal is to design an efficient secure aggregation framework in the single-server, multi-client setting that simultaneously ensures input privacy for both clients and the server, and input integrity. As shown in Figure 4, Vespa consists of an IT-MAC-based private aggregation framework that sums the valid input vectors after integrity checks, supported by two complementary integrity check protocols. The first integrity check is norm check based on commonly used L_2 and L_∞ norm checks. The second integrity check utilizes fine-grained similarity measurements based on Euclidean and Cosine distances, which is motivated by server privacy and can combine with norm checks. We provide a detailed illustration in the following.

A. Integrity-protected Secure Aggregation

We begin with the overall workflow of Vespa, and defines the architecture into which the integrity check protocols plug.

Input. Each client $\mathcal{C}_i$ holds a private input $\mathbf{x}_i \in \mathbb{F}_p^d$, and the server $\mathcal{S}$ holds a private reference $\mathbf{x}_s \in \mathbb{F}_p^d$. Server privacy introduces a fundamental challenge: since $\mathbf{x}_s$ must remain hidden from the clients, any computation that combines a client's input with the server's private reference cannot be expressed as a standard zero-knowledge proof, in which the verifier holds no private inputs. A hybrid operation between client-committed values and server-private values is therefore naturally required.

Our key observation is that the IT-MAC commitments from VOLE-ZK also conform to the structure of a BDOZ-style authentication [51], which is the standard authenticated-share format used in secure two-party computation (2PC) protocols and precisely the structure that supports such hybrid operations. This motivates our choice of IT-MAC as the carrier of client inputs: an IT-MAC $[\mathbf{x}_i]_p$ simultaneously serves as a VOLE-ZK commitment for integrity proofs and as a BDOZ-style authentication for 2PC, with $\mathcal{C}_i$ holding $(\mathbf{x}_i, \mathbf{M}_{i,x})$, $\mathcal{S}$ holding a uniform global key $\Delta \in \mathbb{F}_p$ and the local key $\mathbf{K}_{i,x}$, subject to $\mathbf{M}_{i,x} = \mathbf{K}_{i,x} + \mathbf{x}_i \cdot \Delta$. This dual role enables a primitive we call *committed-private operations*: an interactive procedure between $\mathcal{C}_i$ and $\mathcal{S}$ that takes a committed vector $[\mathbf{x}_i]_p$ and server-held private $\mathbf{x}_s$, and produces shares of $\mathbf{y}_i = f(\mathbf{x}_i, \mathbf{x}_s)$ for some specified operation f (e.g., multiplication, comparison), with $\mathcal{C}_i$ receiving a fresh commitment $[\mathbf{y}_i^{\mathcal{C}}]_p$ and $\mathcal{S}$ receiving a private share $\mathbf{y}_i^{\mathcal{S}}$, subject to $\mathbf{y}_i = \mathbf{y}_i^{\mathcal{C}} + \mathbf{y}_i^{\mathcal{S}}$. We instantiate this primitive in integrity checks (Section III-C), and use it as the connective tissue between the integrity checks and the aggregation, allowing $[\mathbf{x}_i]_p$ to flow through the entire workflow with no recommitment or representation conversion.

Integrity checks. The server and clients run the norm checks (refer to Section III-B) and similarity check (refer to Section III-C) directly on the committed inputs $\{[\mathbf{x}_i]_p\}_{i\in[n]}$. Norm checks operate purely within VOLE-ZK. The similarity check additionally requires cross-terms between a client's committed input and the server's private $\mathbf{x}_s$, which we evaluate via *committed-private operations*. The output is a validity bit $b_i \in \{0, 1\}$ for each client, opened to the server to indicate whether $\mathcal{C}_i$'s committed input passed all checks.

Validity-weighted summation. Given the validity bits, Vespa computes $\mathbf{y} = \sum_{i\in[n]} b_i \cdot \mathbf{x}_i$ in two stages. First, the server and each $\mathcal{C}_i$ invoke *committed-private multiplication* on the committed input $[\mathbf{x}_i]_p$ and the server-held private bit b_i to produce $\mathbf{y}_i = b_i \cdot \mathbf{x}_i$, with $\mathcal{C}_i$ receiving $[\mathbf{y}_i^{\mathcal{C}}]_p$ and the server receiving $\mathbf{y}_i^{\mathcal{S}}$. Second, the parties sum the $\{\mathbf{y}_i\}_{i\in[n]}$ to obtain $\mathbf{y}$. The server can locally aggregate $\mathbf{y}^{\mathcal{S}} = \sum \mathbf{y}_i^{\mathcal{S}}$, but it must additionally learn $\mathbf{y}^{\mathcal{C}} = \sum \mathbf{y}_i^{\mathcal{C}}$ in a verifiable manner without seeing any individual $\mathbf{y}_i^{\mathcal{C}}$, and tolerating dropouts.

Our main idea is to leverage the additive homomorphism of IT-MACs in conjunction with any private aggregation protocol that securely realizes $\mathcal{F}_{\mathsf{PA}}$ (see Section II-D), such as Bonawitz et al. [1], Bell et al. [7], OPA [29], and Willow [28]. The aggregation proceeds in three steps: (1) the server and clients invoke $\mathcal{F}_{\mathsf{PA}}$ to compute the aggregate share $\mathbf{y}^{\mathcal{C}} = \sum_{i\in\mathcal{U}} \mathbf{y}_i^{\mathcal{C}}$ and the aggregate MAC $\mathbf{M}_y = \sum_{i\in\mathcal{U}} \mathbf{M}_{i,y}$ over the surviving clients $\mathcal{U}$, revealing only the sums; (2) the server locally computes $\mathbf{K}_y = \sum_{i\in\mathcal{U}} \mathbf{K}_{i,y}$ and $\mathbf{y}^{\mathcal{S}} = \sum_{i\in\mathcal{U}} \mathbf{y}_i^{\mathcal{S}}$; (3) the server verifies integrity via the IT-MAC relation $\mathbf{M}_y = \mathbf{K}_y + \mathbf{y}^{\mathcal{C}} \cdot \Delta$, and outputs $\mathbf{y} = \mathbf{y}^{\mathcal{C}} + \mathbf{y}^{\mathcal{S}}$ if the check passes. If dropout-resilience is not required, this step admits a lighter-weight instantiation in which clients mask their shares with a joint sharing of zero, avoiding the complex threshold secret-sharing machinery typically required to realize $\mathcal{F}_{\mathsf{PA}}$.

B. Efficient Norm-based Integrity Checks

Norm checks verify whether the norms of input vectors from clients fall within a specified range [11], [2], [3], which serves both as a stand-alone integrity defense and as a lightweight pre-filter that eliminates clearly malformed inputs before the heavier similarity check. Existing works [3], [4], [17] accomplish norm checks by employing general zero-knowledge proof techniques, such as Bulletproofs [19]. Unfortunately, these approaches suffer from inefficiencies, as discussed in Section I. To efficiently execute norm checks, our initial idea is to leverage computation-efficient VOLE-ZK [42], [43], [48], [46] and combine it with recent range proof techniques [50], [35]. However, the main challenge is that existing range proof techniques for VOLE-ZK [50], [35] cause excessively large overhead, particularly when norm checks must accommodate a wide range. We detail our solutions for addressing L_2 and L_∞ norm checks in the following.

L_2 Norm Check. An L_2 norm check with a bound B enables the server $\mathcal{S}$ to verify $y = \mathbf{x}\cdot\mathbf{x} = \sum_{i\in[d]} x_i^2 < B^2$, given $\mathbf{x} = (x_1, \ldots, x_d) \in \mathbb{F}_p^d$ from the client $\mathcal{C}$. A straightforward solution is that given the committed $\mathbf{x}$ from the client $\mathcal{C}$, the server $\mathcal{S}$ checks the following two relations: (1) *Check the inner product*. The server $\mathcal{S}$ checks $y = \mathbf{x}\cdot\mathbf{x}$, which can be viewed as a degree-2 polynomial proof and proved efficiently by VOLE-ZK such as QuickSilver [43]. (2) *Check the squared range*. The server $\mathcal{S}$ checks $y \in [0, B^2)$ by utilizing range proof techniques in VOLE-ZK [50], [35]. However, this will cause $O(B^2)$ overhead, since the cost of existing range proofs is linear with the range size.

Our solution is to execute the range proof on truncated y to reduce the overhead, while not affecting the correctness of the integrity check. The main observation is that in real-world applications, floating-point values of input vectors are represented as fixed-point numbers with fractional precision s, by scaling them and applying floor operations. After computing the inner product $\mathbf{x}\cdot\mathbf{x}$, the result y contains $2s$-bit fractional precision. However, the last significant s fractional bits influence the result by at most $1/2^s$. With $s = 12$ in our experiments, this impact is negligible and does not affect the integrity check outcome. A similar insight is noted by Garg et al. [52] in the context of verifying floating-point computations. Therefore, we truncate y by s bits and perform the range proof on it using the bound of truncated B^2. Specifically, let m be the bit length of B^2, namely $m = \lceil \log B^2 \rceil$, and y be decomposed as $y = y_1 \| y_0$, where $y_1 \in \{0,1\}^{m-s}$, $y_0 \in \{0,1\}^s$. The client $\mathcal{C}$ first commits to y_0 and y_1. The truncated L_2 norm check can be done in two steps as follows. (1) *Check the truncation*. Following the work [35], the server $\mathcal{S}$ checks that $y_1 \in [0, 2^{m-s}]$, $y_0 \in [0, 2^s]$ and $[y_0]_p + 2^s \cdot [y_1]_p - [y]_p = [0]_p$ to ensure $y = y_1 \| y_0$. (2) *Check the range*. The server $\mathcal{S}$ checks $y_1 \in [0, B')$ by utilizing range proof techniques, where $B' = \lfloor B^2/2^s \rfloor$. Compared with the range proof $y \in [0, B^2)$, the overhead is significantly reduced from $O(2^m)$ to $O(2^{m-s+1} + 2^s)$.

We further reduce the cost of step (2) by replacing the range proof with a sublinear OR proof. We observe that the L_2 norm check requires only a single range proof, which makes it particularly well-suited to the OR-based proof of disjunctions introduced by Chiang et al. [53]: their construction proves an OR over N branches with cost sublinear in N via dim 1-hotvectors of length $\sqrt[dim]{N}$ each, where dim is a public parameter chosen by the client to optimize proof size, leading to $O(dim \cdot \sqrt[dim]{N} + dim \cdot \lambda)$ cost. By instantiating step (2) with the OR proof, we recast the range check $y_1 \in [0, B')$ as a disjunction over the B' candidate values, reducing the dominant cost from $O(2^{m-s+1} + 2^s)$ to $O(dim \cdot 2^{(m-s+1)/dim} + dim \cdot \lambda + 2^s)$. Unfortunately, the OR-based proof does not support batching: for the L_∞ check, where d range proofs must be performed across high-dimensional inputs, this forces the entire proof to be repeated d times, which would be prohibitively expensive.

L_∞ Norm Check. An L_∞ norm check with a bound B enables the server $\mathcal{S}$ to verify that x_i is within the range $(-B, B)$ for $i \in [d]$, given $\mathbf{x} = (x_1, \ldots, x_d) \in \mathbb{F}_p^d$ from the client $\mathcal{C}$. A straightforward solution is to check $x_i \in (-B, B)$ for all $i \in [d]$. As mentioned above, this causes about $2B$ overhead in range proof.

Our solution is to verify the absolute value $\hat{x}_i = |x_i| \in [0, B)$, thereby reducing the overhead by half, while ensuring $\hat{x}_i$ is correctly computed almost for free. Specifically, besides the committed inputs x_i, the client $\mathcal{C}$ additionally commits to $\hat{x}_i$ to the server $\mathcal{S}$ for all $i \in [d]$. The L_∞ norm check can be done in two steps as follows. (1) *Check the range*. The server $\mathcal{S}$ checks $\hat{x}_i \in [0, B)$ by utilizing range proof techniques for VOLE-ZK [50], [35]. Notably, the L_∞ norm check invokes range proofs multiple times, especially for high-dimensional input aggregation. The overhead can be further reduced by amortizing the proof process (refer to Section V-B for more details). (2) *Check the absolute value*. To ensure $\hat{x}_i = |x_i|$, the server $\mathcal{S}$ checks $[\hat{x}_i]_p^2 - [x_i]_p^2 = [0]_p$ for all $i \in [d]$. Our idea is to use a random linear combination defined by d random coefficients $\{c_i\}_{i\in[d]}$ sampled from $\mathbb{F}_p$ by the server after commitment. Instead of checking each $[\hat{x}_i]_p^2 - [x_i]_p^2 = [0]_p$ separately, we check $\sum_{i\in[d]} c_i([\hat{x}_i]_p^2 - [x_i]_p^2) = [0]_p$ as a whole. Since $\{c_i\}_{i\in[d]}$ are sampled uniformly at random, we ensure that all $\hat{x}_i = |x_i|$ as long as the above random linear combination holds. We view the relation as a degree-2 polynomial $f(\mathbf{x}, \hat{\mathbf{x}}) = \sum_{i\in[d]} c_i(\hat{x}_i^2 - x_i^2)$ and prove it by utilizing the polynomial proof in QuickSilver [43]. This reduces the communication overhead from $O(d)$ to $O(1)$, achieving the absolute check almost for free.

C. Server-privacy-motivated Similarity-based Integrity Checks

As previously established, we instantiate similarity-based integrity checks with two widely used metrics: Euclidean distance (EucSim) and cosine similarity (CosSim) [13], [9], [25], [54]. In both checks, the server $\mathcal{S}$ holds a private reference vector $\mathbf{x}_s \in \mathbb{F}_p^d$ and verifies each client's private input $\mathbf{x}_i \in \mathbb{F}_p^d$ against it. Realizing such checks under server privacy is exactly the setting that motivates our *committed-private operations* (Section III-A): the cross-terms between

$[\mathbf{x}_i]_p$ and $\mathbf{x}_s$ are evaluated via committed-private operations, while the client-only terms remain in pure VOLE-ZK. We detail our solutions for EucSim and CosSim in the following.

Euclidean Similarity Check. An EucSim check with a bound τ enables the server $\mathcal{S}$ to verify $\|\mathbf{x}_i - \mathbf{x}_s\|_2 < \tau$, which can be formalized as $\mathbf{x}_i \cdot \mathbf{x}_i - 2\mathbf{x}_i \cdot \mathbf{x}_s + \mathbf{x}_s \cdot \mathbf{x}_s < \tau^2$. The client-only term $\mathbf{x}_i \cdot \mathbf{x}_i$ can be evaluated efficiently within VOLE-ZK as a degree-2 polynomial proof on $[\mathbf{x}_i]_p$. The primary technical challenge lies in the cross-term $\mathbf{x}_i \cdot \mathbf{x}_s$: it involves private inputs from both parties, and is therefore beyond the reach of standard zero-knowledge proofs.

Recall from Section III-A our key observation that an IT-MAC commitment $[\mathbf{x}_i]_p$ also conforms to the structure of a BDOZ-style authentication [51], which lets us formalize the *committed-private hybrid operations*: a primitive that operates over a committed value held by the client and a private value held by the server, ensuring the same $[\mathbf{x}_i]_p$ flows through both VOLE-ZK and 2PC computations. We instantiate this primitive with two concrete operations required by our similarity checks: committed-private multiplication and committed-private comparison, both built on 2PC techniques [36], [37]. Specifically, given $[\mathbf{x}_i]_p$ from $\mathcal{C}_i$ and $\mathbf{x}_s$ from $\mathcal{S}$, committed-private multiplication produces shares of $y = \mathbf{x}_i \cdot \mathbf{x}_s$, with $\mathcal{C}_i$ obtaining a commitment $[y_c]_p$ and $\mathcal{S}$ obtaining a private value y_s such that $y = y_c + y_s$.

The check then becomes $[\mathbf{x}_i]_p \cdot [\mathbf{x}_i]_p - 2([y_c]_p + y_s) + \mathbf{x}_s \cdot \mathbf{x}_s < \tau^2$. By the additive homomorphism of IT-MACs, this rearranges to $[\mathbf{x}_i \cdot \mathbf{x}_i - 2y_c]_p < \tau^2 - \mathbf{x}_s \cdot \mathbf{x}_s + 2y_s$, which is precisely a comparison between a committed value $z = \mathbf{x}_i \cdot \mathbf{x}_i - 2y_c$ held by $\mathcal{C}_i$ and a private value $a = \tau^2 - \mathbf{x}_s \cdot \mathbf{x}_s + 2y_s$ held by $\mathcal{S}$. We invoke committed-private comparison to evaluate $[z]_p < a$, completing the check.

Cosine Similarity Check. A CosSim check with a bound τ enables the server $\mathcal{S}$ to verify $\cos\langle \mathbf{x}_i, \mathbf{x}_s \rangle = \frac{\mathbf{x}_i \cdot \mathbf{x}_s}{\|\mathbf{x}_i\|_2 \cdot \|\mathbf{x}_s\|_2} > \tau$. The direct formulation involves two complex operations: division and reciprocal square root required for L_2 norm, which are prohibitively expensive.

We rewrite CosSim as $\frac{\mathbf{x}_i}{\|\mathbf{x}_i\|_2} \cdot \frac{\mathbf{x}_s}{\|\mathbf{x}_s\|_2} > \tau$ to avoid these complex operations in protocol evaluation. In this formulation, the client locally computes the normalization $\bar{\mathbf{x}}_i = \frac{\mathbf{x}_i}{\|\mathbf{x}_i\|_2}$ and commits it to the server. Accordingly, the server locally computes $\bar{\mathbf{x}}_s = \frac{\mathbf{x}_s}{\|\mathbf{x}_s\|_2}$. Then, the server checks that (1) $[\bar{\mathbf{x}}_i]_p \cdot \bar{\mathbf{x}}_s > \tau$ and (2) $\bar{\mathbf{x}}_i$ is a correct normalization of $\mathbf{x}_i$. The first check can be achieved by invoking our committed-private multiplication and comparison protocols between a commitment from the client and a private value held by the server. For the second check, besides the commitment of $[\mathbf{x}_i]_p$ and $[\bar{\mathbf{x}}_i]_p$, the client commits to $[y]_p$, where $y = \|\mathbf{x}_i\|_2$. The server checks: (a) the multiplicative relation $[\mathbf{x}_i]_p - [\bar{\mathbf{x}}_i]_p \cdot [y]_p = [0]_p$ and (b) the normalization $[\|\bar{\mathbf{x}}_i\|_2]_p = [1]_p$. Note that the latter will be a range check due to the error introduced in computing $\bar{\mathbf{x}}_i$, arising from the local square root and division operations performed by the client. If both checks pass, $\bar{\mathbf{x}}_i$ is verified as the correct normalization of $\mathbf{x}_i$.

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

This functionality is parameterized by a prime $p \geq 2^\lambda$, constants min_1, max_1, min_2, max_2 for norm checks, a set $\mathcal{U} = \phi$, $\mathsf{SimCheck} \in \{\mathsf{EucSim}, \mathsf{CosSim}\}$ and thresholds τ_1 and τ_2 for EucSim and CosSim respectively.
SecAgg: On receiving $\{(\mathsf{SecAgg}, \mathbf{x}_h)\}_{h\in[n]}$ from $\{\mathcal{C}_h\}_{h\in[n]}$ and $(\mathsf{SecAgg}, \mathbf{x}_s)$ from $\mathcal{S}$, where $\mathbf{x}_h, \mathbf{x}_s \in \mathbb{F}_p^d$. For $h \in [n]$, check if $\|\mathbf{x}_h\|_\infty \in [min_1, max_1]$, $\|\mathbf{x}_h\|_2 \in [min_2, max_2]$, and send (fail, h) to $\mathcal{S}$ if any checks fails; otherwise, set $\mathcal{U} = \mathcal{U} \cup \{h\}$ and send $(\mathsf{success}, h)$ to $\mathcal{S}$. If $\mathsf{SimCheck} = \mathsf{EucSim}$, compute $b_h = 1\{\|\mathbf{x}_h - \mathbf{x}_s\|_2 < \tau_1\}$; otherwise, compute $b_h = 1\{\cos\langle \mathbf{x}_h, \mathbf{x}_s \rangle > \tau_2\}$. Store and send $\{b_h\}_{h\in\mathcal{U}}$ to $\mathcal{S}$. Compute and store $\mathbf{x} = \sum_{h\in\mathcal{U}} b_h \mathbf{x}_h$. Wait for an input from the adversary; if it is Deliver then output $\mathbf{x}$ to $\mathcal{S}$, otherwise output abort.

Fig. 5: Ideal functionality for secure aggregation.

IV. Secure Aggregation with Server Privacy

We present our secure aggregation scheme, which integrates norm-based and similarity-based integrity checks to exclude malformed input vectors. The norm checks serve as a lightweight pre-filter that eliminates clearly malformed inputs at low cost, internally invoked as a sub-routine within the similarity check protocol Π_{SimCheck} before the heavier 2PC operations are performed; this layered design retains the fine-grained, server-privacy-preserving validation against the server's reference vector while keeping overall cost low. The ideal functionality, denoted as $\mathcal{F}_{\mathsf{SecAgg}}$, is shown in Figure 5 and the corresponding protocol Π_{SecAgg} is detailed in Figure 6.

In our secure aggregation protocol, each client $\mathcal{C}_h$ holds input vector $\mathbf{x}_h$, $h \in [n]$, and server $\mathcal{S}$ holds reference vector $\mathbf{x}_s$. The entire aggregating process is divided into four steps.

In step (1), denoted as **Initialize**, the initialization generates a uniform global key $\Delta \in \mathbb{F}_p$ for IT-MACs.

In step (2), denoted as **Commit**, each client $\mathcal{C}_h$ commits $\mathbf{x}_h$ to server $\mathcal{S}$, $h \in [n]$. Specifically, $\{\mathcal{C}_h\}_{h\in[n]}$ send $\{(\mathsf{Input}, \mathbf{x}_h)\}_{h\in[n]}$ to functionality $\mathcal{F}_{\mathsf{nZK}}$, while $\mathcal{S}$ sends (Input) to functionality $\mathcal{F}_{\mathsf{nZK}}$. As a result, each client $\mathcal{C}_h$ receives $[\mathbf{x}_h]_p$, while $\mathcal{S}$ receives $\{[\mathbf{x}_h]_p\}_{h\in[n]}$.

In step (3), denoted as **InputCheck**, $\{\mathcal{C}_h\}_{h\in[n]}$ and $\mathcal{S}$ first conduct the norm checks by sending $\{(\ell_2\mathsf{Check}, [\mathbf{x}_h]_p\}_{h\in[n])}$ and $\{(\ell_\infty\mathsf{Check}, [\mathbf{x}_h]_p\}_{h\in[n])}$ to $\mathcal{F}_{\mathsf{nZK}}$. This returns two success sets $\mathcal{U}_1 \subseteq [n]$, $\mathcal{U}_2 \subseteq [n]$ to $\mathcal{S}$, containing the identifier of the client who passes the L_2 and L_∞ norm check. $\mathcal{S}$ then defines $\mathcal{U}_3 = \mathcal{U}_1 \cap \mathcal{U}_2$ as the set of valid clients and only interacts with clients in this set later. After norm checks, $\mathcal{S}$ chooses a similarity-based integrity check protocol, i.e., sets $\mathsf{SimCheck} = \mathsf{EucSim}$ or CosSim, which leads to either EucSim-based aggregation or CosSim-based aggregation. Then, $\{\mathcal{C}_h\}_{h\in\mathcal{U}_3}$ and $\mathcal{S}$ invoke Π_{SimCheck} (i.e. Π_{EucSim} or Π_{CosSim}) with input $\{[\mathbf{x}_h]_p\}_{h\in\mathcal{U}_3}$ and $\{([\mathbf{x}_h]_p, \mathbf{x}_s)\}_{h\in\mathcal{U}_3}$,

Protocol Π_{SecAgg}

Parameters: A finite field $\mathbb{F}_p$, scale s that refers to the fractional bitlength when mapping from reals to this field, $\mathsf{SimCheck} \in \{\mathsf{EucSim}, \mathsf{CosSim}\}$. Two thresholds τ_1, τ_2 for input check.

Input: n clients $\{\mathcal{C}_h\}_{h\in[n]}$ hold $\{\mathbf{x}_h\}_{h\in[n]}$, where $\mathbf{x}_h \in \mathbb{F}_p^d$. Server holds $\mathbf{x}_s \in \mathbb{F}_p^d$.

Protocol execution: n clients $\{\mathcal{C}_h\}_{h\in[n]}$ and server $\mathcal{S}$ compute $\mathbf{y}$ as follows:

1) **Initialize:** $\{\mathcal{C}_h\}_{h\in[n]}$ and $\mathcal{S}$ send (Init) to functionality $\mathcal{F}_{\mathsf{nZK}}$, which returns a global key $\Delta \in \mathbb{F}_p$ to $\mathcal{S}$.
2) **Commit:** $\{\mathcal{C}_h\}_{h\in[n]}$ send $\{(\mathsf{Input}, \mathbf{x}_h)\}_{h\in[n]}$ to functionality $\mathcal{F}_{\mathsf{nZK}}$, and $\mathcal{S}$ sends (Input) to $\mathcal{F}_{\mathsf{nZK}}$, which returns $\{[\mathbf{x}_h]_p\}_{h\in[n]}$ to $\{\mathcal{C}_h\}_{h\in[n]}$ and $\mathcal{S}$.
3) **InputCheck:**
   - $\{\mathcal{C}_h\}_{h\in[n]}$ and $\mathcal{S}$ send $\{(\ell_2\mathsf{Check}, [\mathbf{x}_h]_p\}_{h\in[n])}$ and $\{(\ell_\infty\mathsf{Check}, [\mathbf{x}_h]_p\}_{h\in[n])}$ to functionality $\mathcal{F}_{\mathsf{nZK}}$, which returns two success sets $\mathcal{U}_1 \subseteq [n], \mathcal{U}_2 \subseteq [n]$ to $\mathcal{S}$. $\mathcal{S}$ defines $\mathcal{U}_3 = \mathcal{U}_1 \cap \mathcal{U}_2$ as the set of valid clients.
   - $\mathcal{S}$ sets $\mathsf{SimCheck} = \mathsf{EucSim}$ or CosSim. $\{\mathcal{C}_h\}_{h\in\mathcal{U}_3}$ and $\mathcal{S}$ invoke Π_{SimCheck} with input $\{[\mathbf{x}_h]_p\}_{h\in\mathcal{U}_3}$ and $\{([\mathbf{x}_h]_p, \mathbf{x}_s)\}_{h\in\mathcal{U}_3}$, respectively, which returns $\{b_h\}_{h\in\mathcal{U}_3}$ to $\mathcal{S}$.
4) **Summation:**
   - $\{\mathcal{C}_h\}_{h\in\mathcal{U}_3}$ and $\mathcal{S}$ invoke $\Pi_{\mathsf{BroadProd}}$ with input $\{[\mathbf{x}_h]_p\}_{h\in\mathcal{U}_3}$ and $\{([\mathbf{x}_h]_p, b_h)\}_{h\in\mathcal{U}_3}$, respectively, and obtain $[\mathbf{y}_h^{\mathcal{C}}]_p$ and $\mathbf{y}_h^{\mathcal{S}}$, where $\mathbf{y}_h^{\mathcal{C}} + \mathbf{y}_h^{\mathcal{S}} = b_h\mathbf{x}_h$ and $\mathbf{y}_h^{\mathcal{S}}$ is held by $\mathcal{S}$, $h \in \mathcal{U}_3$.
   - Let $\mathcal{U}_4 \subseteq \mathcal{U}_3$ be the set of surviving clients who remain online throughout this step.
   - $\{\mathcal{C}_h\}_{h\in\mathcal{U}_4}$ and $\mathcal{S}$ invoke the private aggregation functionality $\mathcal{F}_{\mathsf{PA}}$ (see Section II-D), where each $\mathcal{C}_h$ inputs $(\mathbf{y}_h^{\mathcal{C}}, \mathbf{M}_h)$. $\mathcal{S}$ obtains the aggregate sums $\mathbf{y}^{\mathcal{C}} = \sum_{h\in\mathcal{U}_4} \mathbf{y}_h^{\mathcal{C}}$ and $\mathbf{M} = \sum_{h\in\mathcal{U}_4} \mathbf{M}_h$.
   - $\mathcal{S}$ locally computes $\mathbf{y}^{\mathcal{S}} = \sum_{h\in\mathcal{U}_4} \mathbf{y}_h^{\mathcal{S}}$ and $\mathbf{K} = \sum_{h\in\mathcal{U}_4} \mathbf{K}_h$.
   - $\mathcal{S}$ aborts if $\mathbf{M} \neq \mathbf{K} + \Delta\mathbf{y}^{\mathcal{C}}$; otherwise, $\mathcal{S}$ outputs the final result $\mathbf{y} = \mathbf{y}^{\mathcal{C}} + \mathbf{y}^{\mathcal{S}}$.

Fig. 6: Protocol Π_{SecAgg} for secure aggregation.

which returns $\{b_h\}_{h\in\mathcal{U}_3}$ to $\mathcal{S}$. Note that $b_h \in \{0, 1\}$ indicates whether the client $\mathcal{C}_h$'s uploaded input vector is valid or not, and can be regarded as the weight of $\mathbf{x}_h$. Thus, the server $\mathcal{S}$ can perform a weighted summation of the input vectors $\{\mathbf{x}_h\}_{h\in\mathcal{U}_3}$ uploaded by the clients $\{\mathcal{C}_h\}_{h\in\mathcal{U}_3}$ in steps (4).

In step (4), denoted as **Summation**, $\{\mathcal{C}_h\}_{h\in\mathcal{U}_3}$ and $\mathcal{S}$ first execute a committed-private broadcast product, an instance of committed-private multiplication that multiplies a committed vector $[\mathbf{x}_h]_p$ with a private scalar b_h held by $\mathcal{S}$ by invoking $\Pi_{\mathsf{BroadProd}}$ (see Figure 15 in Appendix F3 for details). Note

Functionality $\mathcal{F}_{\mathsf{nZK}}^{\mathsf{NormCheck}}$

This functionality is parameterized by a prime $p \geq 2^\lambda$, constant bounds constants min_1, max_1, min_2, max_2, and can invoke a macro $\mathsf{Auth}()$ defined in Figure 1.

ℓ_2**Check**: For $h \in [n]$, on receiving $(\ell_2\mathsf{Check}, [\mathbf{x}_h]_p)$ from $\mathcal{C}_h$ and $\mathcal{S}$, where $\mathbf{x}_h \in \mathbb{F}_p^d$, check if $[\mathbf{x}_h]_p$ is valid and $\|\mathbf{x}_h\|_2 \in [min, max]$, and output (fail, h) to $\mathcal{S}$ if any check fails; otherwise output $(\mathsf{success}, h)$.

ℓ_∞**Check**: For $h \in [n]$, on receiving $(\ell_\infty\mathsf{Check}, [\mathbf{x}_h]_p)$ from $\{\mathcal{C}_h\}_{h\in[n]}$ and $\mathcal{S}$, where $\mathbf{x}_h \in \mathbb{F}_p^d$, check if $[\mathbf{x}_h]_p$ is valid and $|\mathbf{x}_h[i]| \in [min, max]$ for $h \in [n]$, $i \in [d]$, and output (fail, h) to $\mathcal{S}$ if any check fails; otherwise output $(\mathsf{success}, h)$.

Fig. 7: Ideal functionality for ZK proofs of norm checks with n provers and 1 verifier.

that the output is split as a commitment vector $[\mathbf{y}_h^{\mathcal{C}}]_p$ and a private vector $\mathbf{y}_h^{\mathcal{S}}$ held by $\mathcal{S}$, with $\mathbf{y}_h^{\mathcal{C}} + \mathbf{y}_h^{\mathcal{S}} = b_h\mathbf{x}_h$. The server $\mathcal{S}$ and the surviving clients $\{\mathcal{C}_h\}_{h\in\mathcal{U}_4}$ invoke the private aggregation functionality $\mathcal{F}_{\mathsf{PA}}$, which can be instantiated by any existing single-server scheme such as Bonawitz et al. [1], Bell et al. [7], OPA [29], and Willow [28] (see Section II-D). Specifically, each client $\mathcal{C}_h$ inputs $(\mathbf{y}_h^{\mathcal{C}}, \mathbf{M}_h)$ to $\mathcal{F}_{\mathsf{PA}}$ and $\mathcal{S}$ obtains the aggregate share $\mathbf{y}^{\mathcal{C}} = \sum_{h\in\mathcal{U}_4} \mathbf{y}_h^{\mathcal{C}}$ and the aggregate MAC $\mathbf{M} = \sum_{h\in\mathcal{U}_4} \mathbf{M}_h$. In parallel, the server $\mathcal{S}$ locally aggregates its own shares corresponding to the surviving clients, computing $\mathbf{y}^{\mathcal{S}} = \sum_{h\in\mathcal{U}_4} \mathbf{y}_h^{\mathcal{S}}$ and $\mathbf{K} = \sum_{h\in\mathcal{U}_4} \mathbf{K}_h$. Finally, $\mathcal{S}$ checks if $\mathbf{M} = \mathbf{K} + \Delta\mathbf{y}^{\mathcal{C}}$ and outputs abort if the check fails; otherwise outputs $\mathbf{y} = \mathbf{y}^{\mathcal{C}} + \mathbf{y}^{\mathcal{S}}$. The security is illustrated below and the proof is provided in Appendix G; the formal definition of the norm-check functionality $\mathcal{F}_{\mathsf{nZK}}^{\mathsf{NormCheck}}$ appears in Section V.

Theorem 1. *Protocol Π_{SecAgg} securely realizes the functionality $\mathcal{F}_{\mathsf{SecAgg}}$ in the $(\mathcal{F}_{\mathsf{nZK}}, \mathcal{F}_{\mathsf{nZK}}^{\mathsf{NormCheck}})$-hybrid model against a static adversary that either (i) maliciously corrupts a subset of clients or (ii) semi-honestly corrupts the server.*

V. Efficient Norm Checks

In this section, we present the formal protocols for the L_2 and L_∞ norm checks introduced in Section III-B. As established earlier, these checks serve both as a stand-alone integrity defense and as a lightweight pre-filter inside the similarity check protocols. The corresponding ideal functionalities are illustrated in Figure 7.

A. L_2 Norm Check

The L_2 norm of an input vector $\mathbf{x} \in \mathbb{F}_p^d$ is defined as $\|\mathbf{x}\|_2 = \sqrt{\sum_{i\in[d]} x_i^2}$. As discussed in Section III-B, we check the range of $\sum_{i\in[d]} x_i^2$.

Our $\Pi_{\ell_2\mathsf{Check}}$ protocol is detailed in Figure 8. As discussed in Section III-B, we verify the L_2 norm and maintain con-

sistency throughout the process by three key operations: i) *Compute and check the inner product*. In step (1), $\mathcal{C}_h$ computes the inner product y_h and commits it. In step (2), $\mathcal{C}_h$ and $\mathcal{S}$ check that $[y_h]_p - [\mathbf{x}_h]_p \cdot [\mathbf{x}_h]_p = [0]_p$ to ensure that $y_h = \mathbf{x}_h \cdot \mathbf{x}_h$. ii) *Truncate*. From step (3) to (6), $\{\mathcal{C}_h\}_{h\in[n]}$ and $\mathcal{S}$ truncate y_h by s bits and verify the truncation. $\mathcal{C}_h$ parses $y_h = y_{h,1} \| y_{h,0}$, where $y_{h,0} \in \{0,1\}^s$, $y_{h,1} \in \{0,1\}^{m-s}$, $m = \lceil \log max \rceil$. To ensure $y_{h,0}$ and $y_{h,1}$ are proper decomposition of y_h, $\mathcal{C}_h$ first commits to $y_{h,0}$ and $y_{h,1}$. Then, the parties verify their bitlength by checking their range and that $[y_h]_p = [y_{h,0}]_p + 2^s \cdot [y_{h,1}]_p$. If all checks pass, they obtain $[y_{h,1}]_p$, where $y_{h,1}$ is the proper truncation result. iii) *Check the range of the square*. The parties compute new bounds a_0 and a_1 accordingly and check the range of $y_{h,1}$ by sending RangeCheck with input $[y_{h,1}]_p$ and range $[a_0, a_1]$. The security is illustrated below and the proof is provided in Appendix D.

Theorem 2. *Protocol* $\Pi_{\ell_2\mathsf{Check}}$ *securely realizes the* ℓ_2Check *command of functionality* $\mathcal{F}_{\mathsf{nZK}}^{\mathsf{NormCheck}}$ *against static and malicious adversaries in the* $\mathcal{F}_{\mathsf{nZK}}$*-hybrid model.*

B. L_∞ Norm Check

L_∞ norm of an input vector $\mathbf{x} \in \mathbb{F}_p^d$ is defined as $\|\mathbf{x}\|_\infty = \max(\{|x_i|\}_{i\in[d]})$. As discussed in Section III-B, it's sufficient to perform a range check on each element of the input vector, ensuring it lies within a designated bound.

The detailed protocol $\Pi_{\ell_\infty\mathsf{Check}}$ is illustrated in Figure 9. As discussed in Section III-B, we compute the absolute value $\hat{\mathbf{x}}_h$, and make two checks on them: i) *Check Absolute value*. $\mathcal{C}_h$ and $\mathcal{S}$ check $([\hat{\mathbf{x}}_h]_p + [\mathbf{x}_h]_p) * ([\hat{\mathbf{x}}_h]_p - [\mathbf{x}_h]_p) = [\mathbf{0}]_p$ via a CirEval command to ensure $\hat{\mathbf{x}}_h = |\mathbf{x}_h|$. ii) *Check Range*. $\mathcal{C}_h$ and $\mathcal{S}$ check range by invoking RangeCheck with input $[\hat{x}_{h,i}]_p$ and range $[min, max]$, indicating $x_{h,i} \in [min, max]$ along with the above check. Note that the absolute value check in i) can be viewed d degree-2 polynomials $f(x_{h,i}, \hat{x}_{h,i}) = \hat{x}_{h,i}^2 - x_{h,i}^2 = 0$, which incurs $O(d)$ communication overhead. To reduce this cost, we apply a random linear combination: instead of checking each equality separately, we verify $\sum_{i\in[n]} c_i([\hat{x}_{h,i}]_p^2 - [x_{h,i}]_p^2) = [0]_p$ in a whole, with $\{c_i\}_{i\in[d]}$ uniformly sampled from $\mathbb{F}_p$ by the server after commitment. This combination compresses the checks into a single polynomial proof $f(\mathbf{x}_h, \hat{\mathbf{x}}_h) = \sum_{i\in[n]} c_i(\hat{x}_{h,i}^2 - x_{h,i}^2) = 0$, which can be efficiently verified using a zero-knowledge proof over polynomial sets [43], reducing the overhead to $O(1)$. The security is illustrated below and the proof is provided in Appendix E.

Theorem 3. *Protocol* $\Pi_{\ell_\infty\mathsf{Check}}$ *securely realizes the* ℓ_∞Check *command of functionality* $\mathcal{F}_{\mathsf{nZK}}^{\mathsf{NormCheck}}$ *against static and malicious adversaries in the* $\mathcal{F}_{\mathsf{nZK}}$*-hybrid model.*

Combining L_2 and L_∞ norm checks. The L_2 norm check can be viewed as an extension of the L_∞ norm check, since bounding each individual component via an L_∞ constraint implicitly enforces an upper bound on the overall L_2 norm. However, when working with cryptographic primitives over a finite field $\mathbb{F}_p$, relying solely on an upper bound for the L_2 norm is insufficient to constrain the magnitudes of individual components, due to the wrap-around behavior caused by modular arithmetic [2], [3]. For instance, assuming that in a vector $\mathbf{x} \in \mathbb{F}_p^d$, $d > 16$, the first 16 entries $x_i = \frac{p-1}{4}$, $i \in [16]$ while other entries $x_j = 0$, $j \geq 16$. Then when checking the L_2 norm of $\mathbf{x}$, $\|\mathbf{x}\|_2^2 = \sum_{k\in d} x_k^2 = 16 \cdot (\frac{p-1}{4})^2 = 1 \mod p$. If the bound $B > 1$, then the L_2 norm check passes. But the result we expect is that the check fails. In contrast, with L_∞ norm check, this attack can be easily detected. Thus, L_2 norm should be augmented by combining with L_∞ norm check.

Protocol $\Pi_{\ell_2\mathsf{Check}}$

Parameters: A finite field $\mathbb{F}_p$, fixed scale s, two constant bounds min and max.
Input: n clients $\{\mathcal{C}_h\}_{h\in[n]}$ and the server $\mathcal{S}$ hold $\{[\mathbf{x}_h]_p\}_{h\in[n]}$, where $\mathbf{x}_h \in \mathbb{F}_p^d$.
Protocol execution: n clients $\{\mathcal{C}_h\}_{h\in[n]}$ and the server $\mathcal{S}$ verify that $\|\mathbf{x}_h\|_2 \in [min, max]$ holds for $h \in [n]$ as follows:

1) For $h \in [n]$, $\mathcal{C}_h$ computes $y_h = \mathbf{x}_h \cdot \mathbf{x}_h$ and sends (Input, y_h) to functionality $\mathcal{F}_{\mathsf{nZK}}$, and $\mathcal{S}$ sends (Input) to functionality $\mathcal{F}_{\mathsf{nZK}}$, which returns $[y_h]_p$.
2) $\{\mathcal{C}_h\}_{h\in[n]}$ and $\mathcal{S}$ send $\{(\mathsf{CirEval}, \mathsf{C}_0, [y_h]_p, [\mathbf{x}_h]_p)\}_{h\in[n]}$ to $\mathcal{F}_{\mathsf{nZK}}$, where the circuit C_0 outputs 1 if $y_h = \mathbf{x}_h \cdot \mathbf{x}_h$.
3) For $h \in [n]$, $\mathcal{C}_h$ decomposes y_h into $(y_{h,0}, y_{h,1})$ such that $y_h = y_{h,1} \| y_{h,0}$, where $y_{h,0} \in \{0,1\}^s$, $y_{h,1} \in \{0,1\}^{m-s}$, $m = \lceil \log max^2 \rceil$.
4) For $h \in [n]$, $\mathcal{C}_h$ sends $(\mathsf{Input}, y_{h,0}, y_{h,1})$ to functionality $\mathcal{F}_{\mathsf{nZK}}$, which returns $([y_{h,0}]_p, [y_{h,1}]_p)$.
5) For $h \in [n]$, $i \in \{0,1\}$, $\mathcal{C}_h$ and $\mathcal{S}$ send $(\mathsf{RangeCheck}, R_i, [y_{h,i}]_p)$ to functionality $\mathcal{F}_{\mathsf{nZK}}$, where $R_0 = \{0,1\}^s$ and $R_1 = \{0,1\}^{m-s}$.
6) For $h \in [n]$, $\mathcal{C}_h$ and $\mathcal{S}$ compute $[z_h]_p = [y_{h,0}]_p + 2^s \cdot [y_{h,1}]_p - [y_h]_p$ and send $(\mathsf{CirEval}, \mathsf{C}_1, [z_h]_p)$ to $\mathcal{F}_{\mathsf{nZK}}$, where the circuit C_1 outputs 1 if $z_h = 0$.
7) $\{\mathcal{C}_h\}_{h\in[n]}$ and $\mathcal{S}$ locally compute $a_0 = \lfloor \frac{min^2}{2^s} \rfloor$ and $a_1 = \lfloor \frac{max^2}{2^s} \rfloor$.
8) For $h \in [n]$, $\mathcal{C}_h$ and $\mathcal{S}$ send $\{(\mathsf{RangeCheck}, [a_0, a_1], [y_{h,1}]_p)\}_{h\in[n]}$ to functionality $\mathcal{F}_{\mathsf{nZK}}$.
9) For $h \in [n]$, if any of the above checks fails, $\mathcal{S}$ outputs (fail, h), otherwise $\mathcal{S}$ outputs $(\mathsf{success}, h)$.

Fig. 8: Protocol for L_2 norm check.

General L_m norm check. In addition to L_2 and L_∞ norm checks, our framework can support a general L_m norm check by employing a truncation-then-proof approach, the same as the method used for the L_2 norm check.

VI. Server-Privacy-Motivated Similarity Checks

In this section, we design two similarity-based defense protocols motivated by server privacy protection: the Euclidean similarity (EucSim) and cosine similarity (CosSim) checks. As

Protocol $\Pi_{\ell_\infty \mathsf{Check}}$

Parameters: A finite field $\mathbb{F}_p$, two constant bounds min and max.
Input: n clients $\{\mathcal{C}_h\}_{h\in[n]}$ and the server $\mathcal{S}$ hold $\{[\mathbf{x}_h]_p\}_{h\in[n]}$, where $\mathbf{x}_h \in \mathbb{F}_p^d$.
Protocol execution: n clients $\{\mathcal{C}_h\}_{h\in[n]}$ and server $\mathcal{S}$ verify that $|\mathbf{x}_h[i]| \in [min, max]$ holds for $h \in [n]$, $i \in [d]$ as follows:

1) For $h \in [n]$, $\mathcal{C}_h$ sends $(\mathsf{Input}, \hat{\mathbf{x}}_h)$ to functionality $\mathcal{F}_{\mathsf{nZK}}$, and $\mathcal{S}$ sends (Input) to functionality $\mathcal{F}_{\mathsf{nZK}}$, which returns $[\hat{\mathbf{x}}_h]_p$, where $\hat{\mathbf{x}}_h = |\mathbf{x}_h|$.
2) For $h \in [n]$, $\mathcal{C}_h$ and $\mathcal{S}$ send $(\mathsf{CirEval}, \mathsf{C}, [\mathbf{x}_h]_p, [\hat{\mathbf{x}}_h]_p)$ to $\mathcal{F}_{\mathsf{nZK}}$, where the circuit C outputs 1 if $(\hat{\mathbf{x}}_h + \mathbf{x}_h) * (\hat{\mathbf{x}}_h - \mathbf{x}_h) = \mathbf{0}$.
3) For $h \in [n]$, $\mathcal{C}_h$ and $\mathcal{S}$ send $(\mathsf{RangeCheck}, [min, max], [\hat{\mathbf{x}}_h]_p)$ to functionality $\mathcal{F}_{\mathsf{nZK}}$.
4) For $h \in [n]$, if any of the above checks fails, $\mathcal{S}$ outputs (fail, h), otherwise $\mathcal{S}$ outputs $(\mathsf{success}, h)$

Fig. 9: Protocol for L_∞ norm check.

established earlier, we integrate norm checks as a pre-filter inside both protocols, eliminating clearly malformed inputs at low cost before invoking the heavier 2PC operations. We detail the protocols for EucSim and CosSim in Sections VI-A and VI-B, respectively.

A. EucSim

Euclidean distance similarity (EucSim) focuses on the difference in magnitudes between the clients' input vectors and the server's reference vector. It sets a threshold based on the norm (magnitude) of the server's reference, and if the difference between a client's input vector and the server's reference exceeds this threshold, the client's input vector is considered suspicious and will be discarded.

The detailed protocol Π_{EucSim} is illustrated in Figure 10. Note that we incorporate only the L_∞ norm check (refer to step (1)) to avoid wrapping around, omitting the L_2 check. This is because L_2 norm check constrains the Euclidean distance between the input vector and the origin, which is a looser constraint compared to EucSim. In contrast, EucSim measures the distance to a server-provided reference vector, enforcing a stricter and more targeted constraint.

As discussed in Section III-C, we choose to check $(\mathbf{x}_h - \mathbf{x}_s) \cdot (\mathbf{x}_h - \mathbf{x}_s) < \tau_1^2$, namely, $\mathbf{x}_h \cdot \mathbf{x}_h - 2 \cdot \mathbf{x}_h \cdot \mathbf{x}_s < \tau_1^2 - \mathbf{x}_s \cdot \mathbf{x}_s$. The parties carry out the check by executing the following three operations: i) *Compute and check the square*. In steps (2) and (3), $\mathcal{C}_h$ and $\mathcal{S}$ compute z_h and check $[z_h]_p = [\mathbf{x}_h]_p \cdot [\mathbf{x}_h]_p$. ii) *Compute and check the crossing item*. In step (4), $\{\mathcal{C}_h\}_{h\in[n]}$ and $\mathcal{S}$ execute a committed-private hybrid inner product between a commitment vector $[\mathbf{x}_h]_p$ and a private server-held vector $\mathbf{x}_s$ by invoking $\Pi_{\mathsf{InnerProd}}$ (refer to Figure 14 in Appendix F3). Note that the output is also a commitment $[y_h^{\mathcal{C}}]_p$ and a private value $y_h^{\mathcal{S}}$ held by $\mathcal{S}$, s.t. $y_h^{\mathcal{C}} + y_h^{\mathcal{S}} = \mathbf{x}_s \cdot \mathbf{x}_h$. iii) *Committed-private Comparison*. After above operations, the inequality to be checked is: $[z_h]_p - 2([y_h^{\mathcal{C}}]_p + y_h^{\mathcal{S}}) < \tau_1^2 - \mathbf{x}_s \cdot \mathbf{x}_s$, namely, $[z_h]_p - 2[y_h^{\mathcal{C}}]_p < \tau_1^2 - \mathbf{x}_s \cdot \mathbf{x}_s + 2y_h^{\mathcal{S}}$. According to the additive homomorphism of IT-MACs, $\mathcal{C}_h$ and $\mathcal{S}$ locally compute $[e_h]_p = [z_h]_p - 2[y_h^{\mathcal{C}}]_p$. Let $\tau' = \tau_1^2 - \mathbf{x}_s \cdot \mathbf{x}_s + 2y_h^{\mathcal{S}}$. $\{\mathcal{C}_h\}_{h\in[n]}$ and $\mathcal{S}$ compare a commitment value $[e_h]_p$ with a private server-held value τ' by invoking Π_{LTZ}, which is a committed-private hybrid comparison protocol and outputs 1 if $e_h < \tau'$ (refer to Figure 17 in Appendix F4). Note that the output is still a commitment $[b_h^{\mathcal{C}}]_p$ and a private value $b_h^{\mathcal{S}}$. Thus, $\mathcal{C}_h$ opens $[b_h^{\mathcal{C}}]_p$ and $\mathcal{S}$ checks it and outputs $b_h = b_h^{\mathcal{C}} + b_h^{\mathcal{S}}$ if all the checks pass.

Subprotocol Π_{EucSim}

Parameters: A finite field $\mathbb{F}_p$, two constant bounds: bd for ℓ_∞ check, τ_1 for EucSim check.
Input: n clients $\{\mathcal{C}_h\}_{h\in[n]}$ and the server $\mathcal{S}$ hold $\{[\mathbf{x}_h]_p\}_{h\in[n]}$, where $\mathbf{x}_h \in \mathbb{F}_p^d$. Server $\mathcal{S}$ holds $\mathbf{x}_s \in \mathbb{F}_p^d$.
Protocol execution: n clients $\{\mathcal{C}_h\}_{h\in[n]}$ and server $\mathcal{S}$ compute $\{b_h\}_{h\in[n]}$, where $b_h = 1\{\|\mathbf{x}_h - \mathbf{x}_s\|_2 < \tau_1\}$, as follows:

1) $\{\mathcal{C}_h\}_{h\in[n]}$ and $\mathcal{S}$ sends $\{(\ell_\infty\mathsf{Check}, [0, bd], [\mathbf{x}_h]_p)\}_{h\in[n]}$ to functionality $\mathcal{F}_{\mathsf{nZK}}$.
2) $\{\mathcal{C}_h\}_{h\in[n]}$ computes $\{z_h = \mathbf{x}_h \cdot \mathbf{x}_h\}_{h\in[n]}$ and sends $\{(\mathsf{Input}, z_h)\}_{h\in[n]}$ to functionality $\mathcal{F}_{\mathsf{nZK}}$, and $\mathcal{S}$ sends (Input) to functionality $\mathcal{F}_{\mathsf{nZK}}$, which returns $\{[z_h]_p\}_{h\in[n]}$.
3) $\{\mathcal{C}_h\}_{h\in[n]}$ and $\mathcal{S}$ send $\{(\mathsf{CirEval}, \mathsf{C}, [z_h]_p, [\mathbf{x}_h]_p\}_{h\in[n]}$ to $\mathcal{F}_{\mathsf{nZK}}$, where the circuit C outputs 1 if $z_h = \mathbf{x}_h \cdot \mathbf{x}_h$.
4) $\{\mathcal{C}_h\}_{h\in[n]}$ and $\mathcal{S}$ invoke $\Pi_{\mathsf{InnerProd}}$ with input $\{[\mathbf{x}_h]_p\}_{h\in[n]}$ and $\{([\mathbf{x}_h]_p, \mathbf{x}_s)\}_{h\in[n]}$, respectively, and obtain $[y_h^{\mathcal{C}}]_p$ and $y_h^{\mathcal{S}}$, where $y_h^{\mathcal{C}} + y_h^{\mathcal{S}} = \mathbf{x}_s \cdot \mathbf{x}_h$ and $y_h^{\mathcal{S}}$ is held by $\mathcal{S}$, $h \in [n]$.
5) For $h \in [n]$, $\mathcal{C}_h$ and $\mathcal{S}$ locally compute $[e_h]_p = [z_h]_p - 2[y_h^{\mathcal{C}}]_p$. $\mathcal{S}$ locally computes $\tau_h' = \tau_1^2 - \mathbf{x}_s \cdot \mathbf{x}_s + 2y_h^{\mathcal{S}}$.
6) $\{\mathcal{C}_h\}_{h\in[n]}$ and $\mathcal{S}$ invoke Π_{LTZ} with input $\{[e_h]_p\}_{h\in[n]}$ and $\{([e_h]_p, \tau_h')\}_{h\in[n]}$ respectively, and obtain $[b_h^{\mathcal{C}}]_p$ and $b_h^{\mathcal{S}}$, where $b_h^{\mathcal{S}}$ is held by $\mathcal{S}$.
7) For $h \in [n]$, $\mathcal{C}_h$ opens $[b_h^{\mathcal{C}}]_p$ to $\mathcal{S}$ and $\mathcal{S}$ checks if $[b_h^{\mathcal{C}}]_p$ is valid.
8) If any check fails, $\mathcal{S}$ aborts. Otherwise, $\mathcal{S}$ outputs $\{b_h\}_{h\in[n]}$, where $b_h = b_h^{\mathcal{C}} + b_h^{\mathcal{S}}$.

Fig. 10: Subprotocol for Euclidean distance similarity.

B. CosSim

Cosine similarity (CosSim) compares the direction of the client's input vector to that of the server's reference vector. A low cosine similarity score suggests that the input vector

Subprotocol Π_{CosSim}

Parameters: A finite field $\mathbb{F}_p$, scale s that refers to the fractional bitlength, four constant bounds: bd_1 for ℓ_∞ check, bd_2 and error e for ℓ_2 check, τ_2 for CosSim check.
Input: n clients $\{\mathcal{C}_h\}_{h\in[n]}$ and the server $\mathcal{S}$ hold $\{[\mathbf{x}_h]_p\}_{h\in[n]}$, where $\mathbf{x}_h \in \mathbb{F}_p^d$. Server $\mathcal{S}$ holds $\mathbf{x}_s \in \mathbb{F}_p^d$.
Protocol execution: n clients $\{\mathcal{C}_h\}_{h\in[n]}$ and server $\mathcal{S}$ compute $\{b_h\}_{h\in[n]}$, where $b_h = 1\{\cos\langle \mathbf{x}_h, \mathbf{x}_s\rangle > \tau_2\}$:

1) $\{\mathcal{C}_h\}_{h\in[n]}$ and $\mathcal{S}$ sends $\{(\ell_\infty\mathsf{Check}, [0, bd_1], [\mathbf{x}_h]_p)\}_{h\in[n]}$ to functionality $\mathcal{F}_{\mathsf{nZK}}$.
2) For $h \in [n]$, $\mathcal{C}_h$ normalizes $\mathbf{x}_h$ to $\bar{\mathbf{x}}_h = \frac{\mathbf{x}_h}{\|\mathbf{x}_h\|_2}$.
3) $\mathcal{S}$ normalizes $\mathbf{x}_s$ to unit length $\bar{\mathbf{x}}_s = \frac{\mathbf{x}_s}{\|\mathbf{x}_s\|_2}$.
4) $\{\mathcal{C}_h\}_{h\in[n]}$ sends $\{(\mathsf{Input}, \bar{\mathbf{x}}_h, \|\mathbf{x}_h\|_2)\}_{h\in[n]}$ to functionality $\mathcal{F}_{\mathsf{nZK}}$, and $\mathcal{S}$ sends (Input) to functionality $\mathcal{F}_{\mathsf{nZK}}$, which returns $\{([\bar{\mathbf{x}}_h]_p, [\|\mathbf{x}_h\|_2]_p)\}_{h\in[n]}$.
5) $\{\mathcal{C}_h\}_{h\in[n]}$ and $\mathcal{S}$ sends $\{(\mathsf{RangeCheck}, [0, bd_2], [\|\mathbf{x}_h\|_2]_p)\}_{h\in[n]}$ to functionality $\mathcal{F}_{\mathsf{nZK}}$.
6) $\{\mathcal{C}_h\}_{h\in[n]}$ and $\mathcal{S}$ sends $\{(\ell_2\mathsf{Check}, [2^s - e, 2^s + e], [\bar{\mathbf{x}}_h]_p)\}_{h\in[n]}$ to functionality $\mathcal{F}_{\mathsf{nZK}}$.
7) $\{\mathcal{C}_h\}_{h\in[n]}$ and $\mathcal{S}$ send $\{(\mathsf{CirEval}, \mathsf{C}, [\mathbf{x}_h]_p, [\bar{\mathbf{x}}_h]_p, [\|\mathbf{x}_h\|_2]_p\}_{h\in[n]}$ to $\mathcal{F}_{\mathsf{nZK}}$, where the circuit C outputs 1 if $\mathbf{x}_h = \bar{\mathbf{x}}_h * \|\mathbf{x}_h\|_2$.
8) $\{\mathcal{C}_h\}_{h\in[n]}$ and $\mathcal{S}$ invoke $\Pi_{\mathsf{InnerProd}}$ with input $\{[\bar{\mathbf{x}}_h]_p\}_{h\in[n]}$ and $\{([\bar{\mathbf{x}}_h]_p, \bar{\mathbf{x}}_s)\}_{h\in[n]}$, respectively, and obtain $[y_h^{\mathcal{C}}]_p$ and $y_h^{\mathcal{S}}$, where $y_h^{\mathcal{C}} + y_h^{\mathcal{S}} = \bar{\mathbf{x}}_h \cdot \bar{\mathbf{x}}_s$ and $y_h^{\mathcal{S}}$ is held by $\mathcal{S}$, $h \in [n]$.
9) $\{\mathcal{C}_h\}_{h\in[n]}$ and $\mathcal{S}$ invoke Π_{LTZ} with input $\{-[y_h^{\mathcal{C}}]_p\}_{h\in[n]}$ and $\{(-[y_h^{\mathcal{C}}]_p, y_h^{\mathcal{S}} - \tau_2)\}_{h\in[n]}$, respectively, and obtain $[b_h^{\mathcal{C}}]_p$ and $b_h^{\mathcal{S}}$, where $b_h^{\mathcal{S}}$ is held by $\mathcal{S}$.
10) For $h \in [n]$, $\mathcal{C}_h$ opens $[b_h^{\mathcal{C}}]_p$ to $\mathcal{S}$ and $\mathcal{S}$ checks if $[b_h^{\mathcal{C}}]_p$ is valid.
11) If any check fails, $\mathcal{S}$ aborts. Otherwise, $\mathcal{S}$ outputs $\{b_h\}_{h\in[n]}$, where $b_h = b_h^{\mathcal{C}} + b_h^{\mathcal{S}}$.

Fig. 11: Subprotocol for cosine similarity.

is likely to be an outlier or potentially malicious, and thus, it will be discarded.

The detailed protocol Π_{CosSim} is illustrated in Figure 11. We incorporate both L_∞ and L_2 norm checks into Π_{CosSim}, but the method to check the norms differs. We check the L_∞ norm by directly sending $\ell_\infty\mathsf{Check}$ command to $\mathcal{F}_{\mathsf{nZK}}^{\mathsf{NormCheck}}$ (refer to step (1)). However, for L_2 norm check, we observe that the clients need to commit $\|\mathbf{x}_h\|_2$, so we check $\|\mathbf{x}_h\|_2 \in [0, bd_2]$ by invoking $\mathsf{RangeCheck}$ with input $[\|\mathbf{x}_h\|]_2]_p$ and range $[0, bd_2]$ (refer to step (5)). In this way, we reduce the communication of L_2 norm check in Π_{CosSim} to $O(bd_2)$.

As discussed in Section III-C, $\mathcal{C}_h$ and $\mathcal{S}$ normalize their input $\bar{\mathbf{x}}_h = \frac{\mathbf{x}_h}{\|\mathbf{x}_h\|_2}$, $\bar{\mathbf{x}}_s = \frac{\mathbf{x}_s}{\|\mathbf{x}_s\|_2}$ and check $\bar{\mathbf{x}}_h \cdot \bar{\mathbf{x}}_s > \tau_2$. The parties implement this check by following three operations: i) *Compute and check the normalization*. In steps (2) and (4), $\mathcal{C}_h$ computes $\bar{\mathbf{x}}_h$, $\|\mathbf{x}_h\|_2$ and commits them. We divide the check of the normalization into two parts: the first is checking the multiplication relation $[\mathbf{x}_h]_p = [\bar{\mathbf{x}}_h]_p * [\|\mathbf{x}_h\|_2]_p$ and the other is to check that $\bar{\mathbf{x}}_h$ is a normalized vector. However, for the latter, $\|\bar{\mathbf{x}}_h\|_2$ is not exactly equal to 1 because of the errors caused by the rounding down ($x = \lfloor \hat{x} \cdot 2^s \rfloor$, Section II-A). Thus, we set e as the error allowed and verify that $\|\bar{\mathbf{x}}_h\|_2 \in [2^s - e, 2^s + e]$ by invoking $\ell_2\mathsf{Check}$ with input $[\bar{\mathbf{x}}_h]_p$ and range $[2^s - e, 2^s + e]$, ensuring that $\|\bar{\mathbf{x}}_h\|_2 \approx 1$. ii) *Compute and check the crossing item*. In step (8), it follows the same method for computing the crossing term as in Π_{EucSim}. $\{\mathcal{C}_h\}_{h\in[n]}$ and $\mathcal{S}$ execute a committed-private hybrid inner product between $[\bar{\mathbf{x}}_h]_p$ and $\bar{\mathbf{x}}_s$ by invoking $\Pi_{\mathsf{InnerProd}}$ (refer to Figure 14 in Appendix F3). The output is a commitment $[y_h^{\mathcal{C}}]_p$ and a private value $y_h^{\mathcal{S}}$ held by $\mathcal{S}$, s.t. $y_h^{\mathcal{C}} + y_h^{\mathcal{S}} = \bar{\mathbf{x}}_h \cdot \bar{\mathbf{x}}_s$. iii) *Committed-private Comparison*. After above operations, the inequality that needs to be checked becomes: $[y_h^{\mathcal{C}}]_p + y_h^{\mathcal{S}} > \tau_2$, namely, $-[y_h^{\mathcal{C}}]_p < y_h^{\mathcal{S}} - \tau_2$. $\{\mathcal{C}_h\}_{h\in[n]}$ and $\mathcal{S}$ execute a committed-private comparison between $-[y_h^{\mathcal{C}}]_p$ and $y_h^{\mathcal{S}} - \tau_2$ by invoking Π_{LTZ} (refer to Figure 17 in Appendix F4). Note that the output is still a commitment $[b_h^{\mathcal{C}}]_p$ and a private value $b_h^{\mathcal{S}}$. Thus, $\mathcal{C}_h$ opens $[b_h^{\mathcal{C}}]_p$ and $\mathcal{S}$ checks it and outputs $b_h = b_h^{\mathcal{C}} + b_h^{\mathcal{S}}$ if all checks pass.

VII. Evaluation

In this section, we present our experimental evaluation. We organize the evaluation into two parts. Section VII-A compares Vespa against prior single-server works (RoFL [3], ACORN [4], Armadillo [17]) at two levels: the underlying norm-check building blocks (Section VII-A1) and the end-to-end aggregation pipeline (Section VII-A2). We use norm checks as the basis for this comparison, since prior works primarily focus on norm-based defenses. Section VII-B then evaluates Vespa's similarity-based protocols, which includes the similarity checks themselves (Section VII-B1) and the end-to-end aggregation (Section VII-B2). The two-server scheme ELSA [2] operates under a different trust model and is therefore compared in Appendix A.

Experimental Setup. We implement our cryptographic protocols[4] in C++ using the EMP-toolkit [55] and SIMC [36]. All experiments are conducted on a machine equipped with four 20-core Intel Xeon Gold 5218R CPUs (2.10GHz) and 256GiB of RAM. Following RoFL [3], we simulate the network with a 0.5ms latency. The reported overhead for norm and similarity checks in Sections VII-A1 and VII-B1 measures the cost of a single client's check against the server, isolating per-client cost from aggregation overhead.

Implementation Details. To ensure a fair comparison with prior single-server works [3], [4], [17], we align our cryptographic and system parameters with theirs. Following [33], [43], [42], [35], we set the computational security parameter $\kappa = 128$ and the statistical security parameter $\lambda \geq 40$. To meet the requirements of both VOLE-ZK and secure computation protocols, we work over a 59-bit finite field $\mathbb{F}_p$ with $p = 2^{59} - 2^{28} + 1$. By default, we use 48 clients (including

[4]https://github.com/Jiahu235/SecAgg-voleZK-mpc.git

TABLE II: L_2 and L_∞ norm checks comparison with RoFL [3], ACORN-detect [4] and Armadillo [17] across different dimensions. NA is due to the unavailability of the source code and communication cost.

Protocol		Runtime (s)			Communication (MB)		
		32k	131k	524k	32k	131k	524k
L_2	RoFL	5.83	19.19	74.23	7.38	29.50	118.00
	Ours	0.06	0.11	0.19	0.25	1.00	4.00
L_∞	RoFL	5.43	17.71	69.15	2.13	8.50	34.00
	Ours	0.14	0.30	1.11	1.31	5.25	21.00
		19k	273k	818k	19k	273k	818k
L_2	ACORN	5.46	79.42	235.28	0.0017	0.0019	0.0025
	Ours	0.04	0.18	0.26	0.15	2.08	6.24
L_∞	ACORN	5.40	75.44	225.78	0.0016	0.0019	0.0020
	Ours	0.05	0.50	1.64	0.76	10.94	32.78
		4k	8k	16k	4k	8k	16k
L_2	Armadillo	1.74	2.11	3.35	NA	NA	NA
	Ours	0.014	0.033	0.035	0.03	0.06	0.13
L_∞	Armadillo	6.19	8.51	16.21	NA	NA	NA
	Ours	0.010	0.020	0.039	0.17	0.33	0.66

TABLE III: End-to-end performance comparison with RoFL [3], ACORN-detect [4] and Armadillo [17]. Runtime and communication are reported per client. NA is due to the unavailability of the source code and communication cost. *Values are estimated from the original paper.

Protocol	Client runtime (s)		Client comm (MB)	
	L_2	L_∞	L_2	L_∞
	19k entries vector			
RoFL	5.46	5.22	4.50	1.40
Acorn	5.48	5.42	0.04	0.04
Armadillo	4.00*	19.27*	NA	NA
Ours	0.11	0.12	0.30	0.92
	273k entries vector			
RoFL	62.69	59.80	62.40	18.70
Acorn	79.58	75.80	0.49	0.49
Armadillo	57.30*	276.73*	NA	NA
Ours	0.26	0.58	4.17	13.01
	818k entries vector			
RoFL	888.88	880.95	186.60	55.70
Acorn	235.76	226.26	1.46	1.46
Armadillo	171.70*	829.17*	NA	NA
Ours	0.38	1.75	12.49	39.01

20% malicious)[5] with a dropout rate of 0, and adopt the same input dimensions as in prior works.

A. Comparison with Prior Works based on Norm Checks

1) Norm Checks: We evaluate our L_2 and L_∞ norm check protocols against three representative single-server baselines: RoFL [3], ACORN [4], and Armadillo [17]. The results are summarized in Table II.

Runtime. Vespa delivers substantial and consistent runtime advantages over all three baselines, with the gap widening as input dimensions grow. Against RoFL, our protocols achieve up to two orders of magnitude speedup for L_2 checks (around $390\times$ at 524k) and over $60\times$ for L_∞ checks. The advantage is even more pronounced against ACORN, reaching nearly three orders of magnitude for L_2 (around $900\times$ at 818k) and over $150\times$ for L_∞. Our protocols also consistently surpass Armadillo, with L_∞ speedups exceeding $400\times$ across all tested dimensions. These gains stem from two complementary design choices: optimized range proofs with reduced range sizes, and batch proof processing that amortizes cryptographic overhead across high-dimensional vectors.

Communication. The communication cost is more nuanced and reflects a deliberate design trade-off. Against RoFL, our protocols reduce communication by roughly $30\times$ in L_2 checks across all dimensions, while remaining competitive in L_∞ checks. This efficiency stems from the polynomial sets proof optimization [43] and the OR proof technique [53], which allow the client to send only a constant 608 bytes beyond the initial commitment phase. Against ACORN, our protocols exhibit higher communication, as ACORN employs Bulletproofs [19] to achieve logarithmic communication at the cost of heavy group exponentiations, while our scheme builds on VOLE-ZK to trade this logarithmic factor for orders-of-magnitude faster computation. The resulting communication remains practical (e.g., 6.24 MB for L_2 on an 818k-dimensional vector).

2) End-to-end Comparison: We further compare the end-to-end aggregation performance of Vespa against RoFL [3], ACORN-detect [4], and Armadillo [17] across three benchmarks with varying input dimensions, with results summarized in Table III. For a fair comparison, Vespa is restricted to use norm-only validation in this comparison, matching the validation scope of the baselines.

Runtime. Vespa maintains its runtime advantage at the end-to-end level: across all input dimensions, Vespa achieves $1 \sim 3$ orders of magnitude speedup over RoFL, ACORN-detect, and Armadillo for both L_2 and L_∞ norm checks.

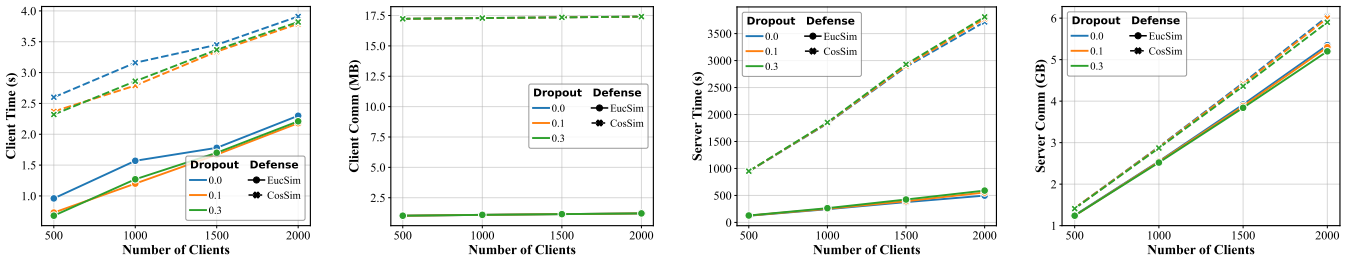
Communication. Vespa also achieves practical communication overhead: on an 818k-entry vector, the total cost is roughly 39 MB. Against RoFL, this represents a reduction of over $10\times$ for L_2 and around $1.4\times$ for L_∞. Vespa incurs higher communication than ACORN-detect, but this trade-off favors typical deployments, where computation on resource-limited clients dominates the end-to-end cost.

B. Evaluation based on Similarity Checks

Having restricted Vespa to norm-only validation in the prior-work comparison, we now evaluate Vespa with its full validation pipeline active, where the server-privacy-preserving similarity check internally invokes the norm check as a pre-filter. We evaluate at two levels: the similarity check protocols themselves (Section VII-B1) and the end-to-end aggregation pipeline (Section VII-B2).

1) Similarity Checks: We report the runtime and communication costs of our similarity check protocols, EucSim and CosSim, in Table IV. Both protocols incorporate the norm check as a pre-filter, so the reported costs reflect the complete validation flow performed for each client. As

[5]This client count matches that of prior works [3]. The number of malicious clients affects neither the performance nor the security of our protocols; security proofs are provided in Appendices C1 and G.



(a) Per-client Runtime (b) Per-client Communication (c) Server Runtime (d) Server Communication

Fig. 12: Computation and communication overhead comparison between EucSim-based and CosSim-based aggregation with varying numbers of clients and dropout rates.

TABLE IV: Runtime (sec) and communication cost (MB) of our similarity check protocols, i.e. EucSim and CosSim.

Protocol		19k		273k		818k	
		Server	Client	Server	Client	Server	Client
EucSim	Time	3.68	1.18	47.18	30.98	141.07	93.77
	Comm.	3.72	2.17	42.08	27.06	123.74	80.24
CosSim	Time.	3.49	2.02	33.25	32.15	97.00	94.72
	Comm.	3.72	18.61	42.08	49.30	123.74	114.97

expected, the overhead grows nearly linearly with the input dimension, owing to the protocols' reliance on efficient vector product operations. We further compare our protocols against EIFFeL [5], the only prior single-server scheme supporting general validation predicates including similarity-based ones. Note that EIFFeL does not protect server privacy: it can only validate against a public reference, whereas our protocols pay additional cryptographic cost to protect a private server-held reference vector. Since EIFFeL's source code is unavailable, we conservatively estimate its overhead at 100 clients and 19k parameters by extrapolating from the measurements in their paper, yielding approximately 21 seconds of client runtime and 250 MB of client communication. Even under this unfavorable setting, our protocols achieve 17× (EucSim) and 10× (CosSim) runtime speedups, with client communication reduced by over 100× for EucSim and over 13× for CosSim. The advantage is expected to grow with deployment scale, since EIFFeL's per-client overhead scales linearly with the number of clients whereas ours is independent of it.

2) End-to-end Aggregation: This section evaluates the end-to-end performance of Vespa with its full similarity-based validation pipeline active. We first assess its efficiency under typical configurations with 48 to 200 clients and input dimensions ranging from 19k to 818k [3], [4], [5]. We then stress-test scalability by scaling the client population up to 2000 under dropout rates of 0, 0.1, and 0.3.

Overhead across client counts and input dimensions. Tables V and VI report the communication and runtime costs of Vespa in a single aggregation iteration with a dropout rate of 0. The most notable observation is that the per-client overhead remains stable as the client population grows: for instance, under EucSim with 818k parameters, per-client communication stays at roughly 147 MB and per-client runtime at roughly 101 seconds, regardless of whether 48, 100, or 200 clients participate. This stability is a desirable property for large-scale deployments, where a client's cost should not be amplified by the size of the cohort it participates in.

TABLE V: The communication cost of Vespa in a single secure aggregation iteration with different numbers of clients. The Server and Client columns are the total overhead (GB) while the Average column is the per-client overhead (MB).

#Client	19k			273k			818k		
	Server	Client	Average	Server	Client	Average	Server	Client	Average
	Aggregation based on EucSim								
48	0.36	0.19	3.99	4.05	2.32	49.51	11.93	6.88	146.80
100	0.74	0.39	4.01	8.44	4.84	49.53	24.85	14.34	146.81
200	1.49	0.79	4.02	16.89	9.68	49.54	49.70	28.68	146.83
	Aggregation based on CosSim								
48	0.37	0.97	20.61	4.08	3.37	71.99	11.96	8.53	181.91
100	0.78	2.01	20.62	8.49	7.03	72.01	24.92	17.77	181.92
200	1.56	4.03	20.64	16.99	14.07	72.02	49.85	35.53	181.93

TABLE VI: The runtime of Vespa in a single secure aggregation iteration with different numbers of clients. The Server and Client columns are the total overhead (min) while the Average column is the per-client overhead (sec).

#Client	19k			273k			818k		
	Server	Client	Average	Server	Client	Average	Server	Client	Average
	Aggregation based on EucSim								
48	2.67	1.11	1.39	38.25	25.52	31.91	119.46	81.16	101.45
100	5.53	2.39	1.43	79.23	53.10	31.86	247.91	169.31	101.59
200	11.01	5.12	1.54	158.25	106.75	32.02	492.14	336.61	100.98
	Aggregation based on CosSim								
48	2.54	1.76	2.20	27.00	26.37	32.96	87.57	85.30	106.62
100	5.36	3.83	2.30	56.04	55.16	33.10	183.26	180.00	108.00
200	10.74	8.02	2.41	111.71	110.76	33.23	365.28	360.91	108.27

Scalability and dropout-resilience. We then stress-test scalability by scaling the client population from 500 to 2000 with the input dimension fixed at 1k, under dropout rates of 0, 0.1, and 0.3. Since the per-client cost of our integrity checks is independent of both the cohort size and the dropout rate, the scaling behavior observed here is governed by the underlying private aggregation protocol Π_{PA}, which we instantiate with

Bonawitz et al. [1]. As shown in Figure 12, the client side scales gracefully: runtime grows near-linearly with the cohort size while communication remains nearly constant and entirely independent of the dropout rate. The server side exhibits linear growth in both runtime and communication relative to the client population. Higher dropout rates slightly increase server runtime due to the additional unmasking operations required for dropped clients, but the impact on server communication is negligible, as unmasking costs little compared to the similarity checks.

Ethical Considerations

Vespa addresses privacy and integrity risks in secure aggregation. For end-users, it protects individual inputs while filtering malicious contributions that could manipulate aggregate outputs. For service providers, it enables validation against a private reference vector without revealing that vector, making it practical to jointly protect privacy and integrity at scale. More broadly, our work demonstrates that stronger input validation can be efficiently integrated into privacy-preserving aggregation systems, including federated learning and private telemetry. Like other validation systems, Vespa does not eliminate all integrity risks: malicious inputs satisfying its checks may still be accepted, while poorly chosen thresholds may reject benign inputs. These limitations should be considered when configuring the system. We publish our protocol and evaluation to support scrutiny, reproducibility, and further research.

Generative AI Usage

During the preparation of this work, the authors utilized Gemini and Claude exclusively for linguistic improvements, including grammar checking, spelling corrections, and stylistic polishing. The authors meticulously reviewed and verified all AI-suggested revisions to ensure accuracy and coherence. We confirm that no scientific concepts, or experimental data were generated by AI tools.

Acknowledgment

We would like to express our deepest gratitude for all the reviewers' constructive comments. This work is supported by the National Natural Science Foundation of China under Grant U25B2079, Grant 62502079 and Grant 62402090, the Sichuan Natural Science Foundation 2026NSFSC1463, and the Singapore Ministry of Education (MOE) Academic Research Fund (AcRF) Tier 1 grant (Proposal ID: 24-SIS-SMU-052).

References

[1] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth, "Practical secure aggregation for privacy-preserving machine learning," in *Proceedings of ACM CCS*, 2017, pp. 1175–1191.

[2] M. Rathee, C. Shen, S. Wagh, and R. A. Popa, "Elsa: Secure aggregation for federated learning with malicious actors," in *Proceedings of IEEE S&P*, 2023, pp. 1961–1979.

[3] H. Lycklama, L. Burkhalter, A. Viand, N. Küchler, and A. Hithnawi, "Rofl: Robustness of secure federated learning," in *Proceedings of IEEE S&P*, 2023, pp. 453–476.

[4] J. Bell, A. Gascón, T. Lepoint, B. Li, S. Meiklejohn, M. Raykova, and C. Yun, "Acorn: input validation for secure aggregation," in *Proceedings of USENIX Security*, 2023, pp. 4805–4822.

[5] A. Roy Chowdhury, C. Guo, S. Jha, and L. van der Maaten, "Eiffel: Ensuring integrity for federated learning," in *Proceedings of ACM CCS*, 2022, pp. 2535–2549.

[6] J. So, R. E. Ali, B. Güler, J. Jiao, and A. S. Avestimehr, "Securing secure aggregation: Mitigating multi-round privacy leakage in federated learning," in *Proceedings of AAAI*, vol. 37, no. 8, 2023, pp. 9864–9873.

[7] J. H. Bell, K. A. Bonawitz, A. Gascón, T. Lepoint, and M. Raykova, "Secure single-server aggregation with (poly)logarithmic overhead," in *Proceedings of ACM CCS*, 2020, p. 1253–1269.

[8] R. Xu, B. Li, C. Li, J. B. Joshi, S. Ma, and J. Li, "Tapfed: Threshold secure aggregation for privacy-preserving federated learning," *IEEE Transactions on Dependable and Secure Computing*, vol. 21, no. 5, pp. 4309–4323, 2024.

[9] Y. Dong, X. Chen, K. Li, D. Wang, and S. Zeng, "Flod: Oblivious defender for private byzantine-robust federated learning with dishonest-majority," in *Proceedings of ESORICS*, 2021.

[10] H. Corrigan-Gibbs and D. Boneh, "Prio: Private, robust, and scalable computation of aggregate statistics," in *Proceedings of USENIX NSDI*, 2017, pp. 259–282.

[11] S. Addanki, K. Garbe, E. Jaffe, R. Ostrovsky, and A. Polychroniadou, "Prio+: Privacy preserving aggregate statistics via boolean shares," in *Proceedings of SCN*, 2022, pp. 516–539.

[12] M. Rathee, Y. Zhang, H. Corrigan-Gibbs, and R. A. Popa, "Private analytics via streaming, sketching, and silently verifiable proofs," in *Proceedings of IEEE S&P*. IEEE, 2024, pp. 3072–3090.

[13] X. Cao, M. Fang, J. Liu, and N. Z. Gong, "Fltrust: Byzantine-robust federated learning via trust bootstrapping," in *Proceedings of NDSS*, 2021.

[14] M. Fang, X. Cao, J. Jia, and N. Gong, "Local model poisoning attacks to byzantine-robust federated learning," in *Proceedings of USENIX Security*, 2020, pp. 1605–1622.

[15] J.-y. Sohn, D.-J. Han, B. Choi, and J. Moon, "Election coding for distributed learning: Protecting signsgd against byzantine attacks," in *Proceedings of NeurIPS*, 2020, pp. 14 615–14 625.

[16] N. Franzese, A. Dziedzic, C. A. Choquette-Choo, M. R. Thomas, M. A. Kaleem, S. Rabanser, C. Fang, S. Jha, N. Papernot, and X. Wang, "Robust and actively secure serverless collaborative learning," in *Proceedings of NeurIPS*, 2024.

[17] Y. Ma, Y. Guo, H. Karthikeyan, and A. Polychroniadou, "Armadillo: Robust single-server secure aggregation for federated learning with input validation," in *Proceedings of ACM CCS*, 2025, pp. 2219–2233.

[18] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE Transactions on Information Theory*, vol. 31, no. 4, pp. 469–472, 1985.

[19] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, "Bulletproofs: Short proofs for confidential transactions and more," in *Proceedings of IEEE S&P*, 2018, pp. 315–334.

[20] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," *Journal of the ACM (JACM)*, vol. 60, no. 6, pp. 1–35, 2013.

[21] M. O. Rabin and J. O. Shallit, "Randomized algorithms in number theory," *Communications on Pure and Applied Mathematics*, vol. 39, no. S1, pp. S239–S256, 1986.

[22] Y. Liu, Z. Jia, X. Chen, S. Bian, R. Xu, D. Li, and Y. Lu, "Aion: Robust and efficient {Multi-Round}{Single-Mask} secure aggregation against malicious participants," in *Proceedings of USENIX Security*, 2025, pp. 3025–3044.

[23] G. Damaskinos, E. M. El Mhamdi, R. Guerraoui, R. Patra, and M. Taziki, "Asynchronous byzantine machine learning (the case of sgd)," in *Proceedings of ICML*, 2018, pp. 1145–1154.

[24] J. Bernstein, J. Zhao, K. Azizzadenesheli, and A. Anandkumar, "Signsgd with majority vote is communication efficient and fault tolerant," in *Proceedings of ICLR*, 2019.

[25] C. Xie, S. Koyejo, and I. Gupta, "Zeno++: Robust fully asynchronous SGD," in *Proceedings of ICML*, 2020, pp. 10 495–10 503.

[26] Y. Ma, J. Woods, S. Angel, A. Polychroniadou, and T. Rabin, "Flamingo: Multi-round single-server secure aggregation with applications to private federated learning," in *Proceedings of IEEE S&P*, 2023, pp. 477–496.

[27] T. S. Nguyen, T. Lepoint, and N. Trieu, "Mario: multi-round multiple-aggregator secure aggregation with robustness against malicious actors," in *Proceedings of EuroS&P*. IEEE, 2025, pp. 1140–1164.

[28] J. Bell-Clark, A. Gascón, B. Li, M. Raykova, and P. Schoppmann, "Willow: secure aggregation with one-shot clients," in *Proceedings of CRYPTO*. Springer, 2025, pp. 285–318.
[29] H. Karthikeyan and A. Polychroniadou, "Opa: one-shot private aggregation with single client interaction and its applications to federated learning," in *Proceedings of CRYPTO*. Springer, 2025, pp. 319–353.
[30] R. Lehmkuhl, H. Corrigan-Gibbs, E. Dauterman, and D. J. Wu, "Heli: Heavy-light private aggregation," *Proceedings of USENIX Security*, 2026.
[31] Z. Lu and S. Lu, "Lzksa: Lattice-based special zero-knowledge proofs for secure aggregation's input verification," *Cryptology ePrint Archive*, 2025.
[32] H. Karthikeyan and A. Polychroniadou, "TAPAS: Efficient two-server asymmetric private aggregation beyond prio(+)," Cryptology ePrint Archive, Paper 2026/564, 2026. [Online]. Available: https://eprint.iacr.org/2026/564
[33] C. Weng, K. Yang, X. Xie, J. Katz, and X. Wang, "Mystique: Efficient conversions for zero-knowledge proofs with applications to machine learning," in *Proceedings of USENIX Security*, 2021.
[34] T. Liu, X. Xie, and Y. Zhang, "Zkcnn: Zero knowledge proofs for convolutional neural network predictions and accuracy," in *Proceedings of ACM CCS*, 2021, pp. 2968–2985.
[35] M. Hao, H. Chen, H. Li, C. Weng, Y. Zhang, H. Yang, and T. Zhang, "Scalable zero-knowledge proofs for non-linear functions in machine learning," in *Proceedings of USENIX Security*, 2024.
[36] N. Chandran, D. Gupta, S. L. B. Obbattu, and A. Shah, "SIMC: ML inference secure against malicious clients at Semi-Honest cost," in *Proceedings of USENIX Security*, 2022, pp. 1361–1378.
[37] R. Lehmkuhl, P. Mishra, A. Srinivasan, and R. A. Popa, "Muse: Secure inference resilient to malicious clients," in *Proceedings of USENIX Security*, 2021.
[38] P. Mohassel, M. Rosulek, and N. Trieu, "Practical privacy-preserving k-means clustering," in *Proceedings of PETS*, vol. 4, 2020, pp. 414–433.
[39] D. Kaviani, S. Tan, P. G. Kannan, and R. A. Popa, "Flock: A framework for deploying {On-Demand} distributed trust," in *Proceedings of USENIX OSDI*, 2024, pp. 721–743.
[40] I. Damgård, V. Pastro, N. Smart, and S. Zakarias, "Multiparty computation from somewhat homomorphic encryption," in *Proceedings of CRYPTO*, 2012, pp. 643–662.
[41] J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra, "A new approach to practical active-secure two-party computation," in *Proceedings of CRYPTO*, 2012, pp. 681–700.
[42] C. Weng, K. Yang, J. Katz, and X. Wang, "Wolverine: fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits," in *Proceedings of IEEE S&P*, 2021, pp. 1074–1091.
[43] K. Yang, P. Sarkar, C. Weng, and X. Wang, "Quicksilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field," in *Proceedings of the 2021 ACM CCS*, 2021, pp. 2986–3001.
[44] I. Damgård, J. B. Nielsen, M. Nielsen, and S. Ranellucci, "The tinytable protocol for 2-party secure computation, or: Gate-scrambling revisited," in *Proceedings of CRYPTO*, 2017, pp. 167–187.
[45] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, P. Rindal, and P. Scholl, "Efficient two-round ot extension and silent non-interactive secure computation," in *Proceedings of ACM CCS*, 2019, pp. 291–308.
[46] S. Dittmer, Y. Ishai, and R. Ostrovsky, "Line-point zero knowledge and its applications," *Cryptology ePrint Archive*, 2020.
[47] C. Baum, A. J. Malozemoff, M. B. Rosen, and P. Scholl, "Mac'n'cheese: Zero-knowledge proofs for boolean and arithmetic circuits with nested disjunctions," in *Proceedings of CRYPTO*, 2021, pp. 92–122.
[48] S. Dittmer, Y. Ishai, S. Lu, and R. Ostrovsky, "Improving line-point zero knowledge: two multiplications for the price of one," in *Proceedings of ACM CCS*, 2022, pp. 829–841.
[49] C. Weng, K. Yang, Z. Yang, X. Xie, and X. Wang, "Antman: Interactive zero-knowledge proofs with sublinear communication," in *Proceedings of ACM CCS*, 2022.
[50] Y. Yang and D. Heath, "Two shuffles make a ram: Improved constant overhead zero knowledge ram," in *Proceedings of USENIX Security*, 2024.
[51] R. Bendlin, I. Damgård, C. Orlandi, and S. Zakarias, "Semi-homomorphic encryption and multiparty computation," in *Proceedings of EUROCRYPT*, 2011, pp. 169–188.
[52] S. Garg, A. Jain, Z. Jin, and Y. Zhang, "Succinct zero knowledge for floating point computations," in *Proceedings of the ACM CCS*, 2022, pp. 1203–1216.

TABLE VII: Runtime (sec) comparison of ELSA [2] and Vespa across different dimensions and norm check protocols.

Protocol		32k		131k		262k		524k	
		Server	Client	Server	Client	Server	Client	Server	Client
L_2	ELSA	3.08	2.04	10.84	9.38	19.98	17.50	48.35	32.79
	Ours	0.06	0.01	0.11	0.04	0.18	0.07	0.19	0.15
L_∞	ELSA	2.21	1.23	10.04	5.11	18.10	9.80	41.39	21.88
	Ours	0.14	0.07	0.30	0.19	0.49	0.40	1.11	1.05

[53] J. H.-Y. Chiang, I. Damgård, W. R. Duro, S. Engan, S. Kolby, and P. Scholl, "Post-quantum threshold ring signature applications from vole-in-the-head," in *Proceedings of ACM CCS*, 2025, pp. 4664–4678.
[54] J. Steinhardt, P. W. W. Koh, and P. S. Liang, "Certified defenses for data poisoning attacks," in *Proceedings of NeurIPS*, 2017.
[55] X. Wang, A. J. Malozemoff, and J. Katz, "EMP-toolkit: Efficient MultiParty computation toolkit," https://github.com/emp-toolkit, 2016.
[56] T.-M. H. Hsu, H. Qi, and M. Brown, "Measuring the effects of non-identical data distribution for federated visual classification," *arXiv:1909.06335*, 2019.
[57] A. N. Bhagoji, S. Chakraborty, P. Mittal, and S. Calo, "Analyzing federated learning through an adversarial lens," in *Proceedings of ICML*, 2019, pp. 634–643.
[58] M. Keller, "Mp-spdz: A versatile framework for multi-party computation," in *Proceedings of ACM CCS*, 2020, pp. 1575–1590.
[59] X. Wang, S. Ranellucci, and J. Katz, "Authenticated garbling and efficient maliciously secure two-party computation," in *Proceedings of ACM CCS*, 2017, pp. 21–37.
[60] O. Goldreich, *Foundations of Cryptography: Volume 1, Basic Tools, 2001*. Cambridge university press Cambridge, 2001.
[61] ——, *Foundations of Cryptography: Volume 2, Basic Applications, 2004*. Cambridge university press Cambridge, 2004.
[62] N. Franzese, J. Katz, S. Lu, R. Ostrovsky, X. Wang, and C. Weng, "Constant-overhead zero-knowledge for ram programs," in *Proceedings of ACM CCS*, 2021, pp. 178–191.
[63] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Proceedings of EUROCRYPT*, 1999, pp. 223–238.
[64] M. S. Riazi, M. Samragh, H. Chen, K. Laine, K. Lauter, and F. Koushanfar, "Xonn: Xnor-based oblivious deep neural network inference," in *Proceedings of USENIX Security*, 2019, pp. 1501–1518.
[65] Z. Huang, W.-j. Lu, C. Hong, and J. Ding, "Cheetah: Lean and fast secure two-party deep neural network inference," in *Proceedings of the USENIX Security*, 2022.
[66] R. Cramer, I. B. Damgård, and J. B. Nielsen, *Secure multiparty computation and secret sharing*. Cambridge University Press, 2015.
[67] Q. Zhang, C. Xin, and H. Wu, "Gala: Greedy computation for linear algebra in privacy-preserved neural networks," in *Proceedings of NDSS*, 2021.
[68] V. Lyubashevsky and G. Seiler, "Short, invertible elements in partially splitting cyclotomic rings and applications to lattice-based zero-knowledge proofs," in *Proceedings of EUROCRYPT*. Springer, 2018, pp. 204–224.
[69] C. Baum, I. Damgård, V. Lyubashevsky, S. Oechsner, and C. Peikert, "More efficient commitments from structured lattice assumptions," in *Proceedings of SCN*. Springer, 2018, pp. 368–385.

APPENDIX

A. Additional Experiments

1) Comparison with two-server work: We compare the runtime and communication cost of our norm checks with two-server work ELSA [2]. Tables VII and VIII demonstrate Vespa's substantial runtime and communication performance advantages. Vespa outperforms ELSA in both norm checks, achieving up to a $250\times$ server-side speedup for L_2 and $37\times$ for L_∞. This challenges the conventional assumption that single-server models are inherently less efficient than two-server architectures. These extensive performance gains are

TABLE VIII: Communication (MB) comparison of ELSA [2] and Vespa across different dimensions and norm check protocols.

Protocol		32k		131k		262k		524k	
		Server	Client	Server	Client	Server	Client	Server	Client
L_2	ELSA	10.50	17.13	42.00	68.50	84.00	137.00	168.00	274.00
	Ours	0.00	0.25	0.00	1.00	0.00	2.00	0.00	4.00
L_∞	ELSA	8.00	16.13	32.00	64.50	64.00	129.00	128.00	258.00
	Ours	0.00	1.31	0.00	5.25	0.00	10.50	0.00	21.00

primarily attributed to our design of highly efficient protocols that facilitate batch proof processing, which significantly amortizes the computational overhead across large-scale input vectors. Besides, Vespa outperforms ELSA by around $68\times$ for L_2 and $12\times$ for L_∞ in communication, which stems from the polynomial sets proof optimization [43] and the OR proof technique [53]. Notably, Vespa incurs no additional server-side communication, as validation is performed on data already received (accounting for ELSA's inter-server communication would further increase our total gains).

2) Efficacy of integrity defense: We evaluate the defense efficacy of our secure aggregation protocols, instantiated with both EucSim and CosSim. This evaluation has two main goals: first, to assess the protocols' robustness against typical input integrity attacks; and second, to identify the optimal scale parameter s at which model accuracy is comparable to the full-precision baseline. In our experiments, the default scale for fixed-point representation is set to $s = 12$, a value estimated from updates and validated experimentally (see Figure 13).

We simulate the non-iid setting by utilizing the Dirichlet distribution $Dir(\alpha)$ in [56] with $\alpha = 0.9$, given that $\alpha \in [0, +\infty)$. The 5 attacks considered are: (1) *Stochastic sign attack* [24] randomizes the sign of the local update; (2) *Sign flipping attack* [23], [24] flips sign of the local update; (3) *Stochastic label flipping (LF)* [14], [13] flips training samples with label i to a random label j; (4) *Selective label flipping (LF)* [14], [13] flips training samples with label i to a designated label j; (5) *Scaling attack* [57] scales up the local updates.

The results are presented in Figure 13, where the colored lines show the model accuracy under these attacks for different scale values. These are compared against the black solid line, which represents the baseline accuracy of standard FedAvg in plaintext without any attacks. The results in Figure 13 lead to two key conclusions. First, our similarity-based integrity defenses are highly effective. While model accuracy is low under all five attacks with small scale values (e.g., $s \leq 6$), it becomes nearly comparable to the attack-free, full-precision baseline (the black solid line) once s reaches a sufficient value. Second, the required precision is model- and dataset-dependent. For the CIFAR-10 dataset, a scale of $s = 12$ is required to achieve accuracy comparable to the baseline. For the Shakespeare dataset, however, a smaller scale of $s = 8$ is sufficient. Based on these findings, we select $s = 12$ as our default parameter to ensure robust performance across different scenarios, as higher values would significantly increase the overhead of the norm checks.

B. Detailed Threat Model and Comparison Setting

This appendix provides the detailed threat model for Vespa and discusses how it relates to the threat models of prior single-server secure aggregation works.

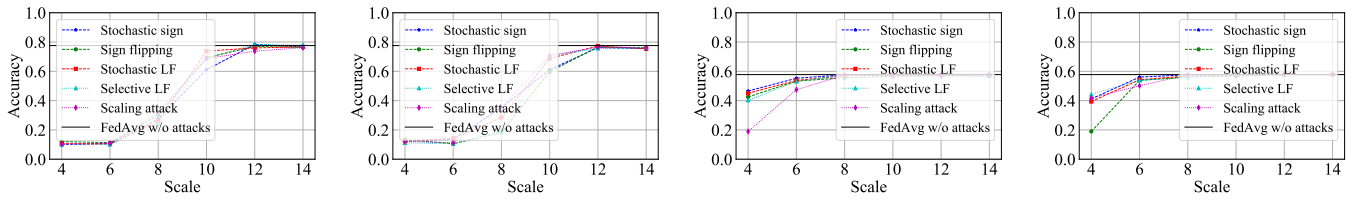
1) Threat Model per Component: Vespa consists of three protocol layers, each with its own threat model:

- **Norm check protocols (L_2 and L_∞).** Both protocols achieve malicious security against the client and the server. Soundness against a malicious client is inherited from the soundness of the underlying VOLE-based zero-knowledge proof system [42], [43], [33]; zero-knowledge against a malicious server is inherited from the malicious-verifier zero-knowledge property of the same system. Formal proofs are given in Appendix D and E.
- **Similarity check protocols (EucSim and CosSim).** Both protocols achieve malicious-client security and semi-honest-server security in our current instantiation.[6] We follow SIMC [36] to obtain malicious-client security at semi-honest cost: linear operations use AHE with MAC tags that authenticate the client's inputs, and non-linear operations use GC whose inputs are bound to the same MAC tags. The semi-honest-server assumption stems from the underlying multiplication and comparison protocols, which are themselves secure against a semi-honest server; replacing them with malicious-server-secure counterparts (e.g., [58], [59]) would lift the similarity check protocols to full malicious security on both sides, at the cost of additional cryptographic overhead.
- **Aggregation.** The aggregation inherits the threat model of the underlying private aggregation protocol Π_{PA} (see Section II-D). For instance, instantiations such as Bonawitz et al. [1] provide input privacy of honest clients against a malicious server.

2) Fair Comparison with Prior Single-Server Works: Our prior-work comparison (Section VII-A) is conducted at the malicious-client and malicious-server threat model, where Vespa uses only norm-based validation to match the input validation scope of RoFL [3], ACORN-detect [4], and Armadillo [17]. At this threat model and validation scope, Vespa and the baselines are directly comparable.[7] For similarity checks, we provide no comparison since no prior single-server scheme provides this functionality with server privacy.

[6] As large service providers like Amazon Web Services, Google Cloud, and Microsoft Azure are unlikely to intentionally violate the protocol, a semi-honest server assumption is also reasonable and widely accepted [10], [3], [4], [11], [2]. These companies are driven by reputation concerns and legal constraints, so they would act as semi-honest servers, seeking to extract more information through normal protocol execution.

[7] Recent analysis in Armadillo [17] Remark 3 identifies a tension between input validation and privacy against a malicious server in proofs that rely on the Programmable Random Oracle Model, which underlies certain prior schemes. Vespa is unaffected by this issue: VOLE-ZK proof systems do not rely on programmable random oracles for malicious-server zero-knowledge, and our IT-MAC commitments are information-theoretic, sidestepping the binding-violation pattern identified in Armadillo [17].



(a) EucSim on CIFAR-10 (b) CosSim on CIFAR-10 (c) EucSim on Shakespeare (d) CosSim on Shakespeare

Fig. 13: Model accuracy of Vespa in FL with different fractional scales against various integrity attacks.

C. Security model

We use the sequential composability (SC) framework [60], [61] to prove security in the presence of a probabilistic polynomial time (PPT), static, adversary $\mathcal{A}$. $\mathcal{A}$ may either maliciously corrupt a subset of clients or semi-honestly corrupt the server, but not both[8]. We use the standard simulation-based security definition. Generally, a protocol is said to realize an ideal functionality if executing the protocol effectively emulates the ideal process associated with that functionality.

Definition 1. Let $\mathcal{F}$ be an ideal functionality and let Π be a protocol. We say that Π securely realizes $\mathcal{F}$ if for any adversary $\mathcal{A}$ there exists an ideal-process adversary $\mathcal{S}$ such that

$$\mathsf{IDEAL}_{\mathcal{F},\mathcal{S}} \approx_c \mathsf{REAL}_{\Pi,\mathcal{A}}. \tag{1}$$

Here, $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S}}$ denotes the ensemble $\{\mathsf{IDEAL}_{\mathcal{F},\mathcal{S}}(\kappa, x)\}_{\kappa,x}$, which is the view and output of the parties in the ideal process on security parameter κ and input x. $\mathsf{REAL}_{\Pi,\mathcal{A}}$ denotes the ensemble $\{\mathsf{REAL}_{\Pi,\mathcal{A}}(\kappa, x)\}_{\kappa,x}$, which is the view and output of the parties when running protocol Π on security parameter κ and input x. Besides, we prove the security of our protocols in the $\mathcal{G}$-hybrid model, where the parties execute a protocol with real messages and also have access to an ideal functionality $\mathcal{G}$. Note that in our protocols, the clients act as provers, while the server serves as the verifier.

1) Proofs of ZKP protocols: We provide the security proof for our ZK proofs of building blocks. In our framework, the malicious server acts as the verifier, similar to the previous ZK works [62], [50], [35], the verifier in these protocols does not have any input and only receives messages from the functionalities. Therefore, it is straightforward to prove security for a corrupted verifier, and hence we only focus on the case of a corrupted prover below.

D. Proof of L_2 Norm Check

Malicious prover. Note that the clients act as provers in our protocols, while the server is the verifier. Thus, we assume that adversary $\mathcal{A}$ corrupts $n-2$ clients. Without loss of generality, assume that $\mathcal{A}$ corrupts provers $\mathcal{C}_0, \ldots, \mathcal{C}_{n-3}$. Note that simulator $\mathcal{S}$ extracts $\mathbf{x}_h$ and the corresponding MAC tags in the procedure of generating authenticated values $[\mathbf{x}_h]_p$ for $h \in [n-2]$, and can send them to $\mathcal{F}_{\mathsf{nZK}}$. $\mathcal{S}$ interacts with adversary $\mathcal{A}$ as follows:

1. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{nZK}}$, and receives $\{y_h\}_{h\in[n-2]}$ and the corresponding MAC tags from $\mathcal{A}$.
2. For $h \in [n-2]$, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{nZK}}$ and receives C_0, $\mathbf{x}_h$, y_h and their corresponding MAC tags, and then compute the MAC tag on $[y_h]_p - [\mathbf{x}_h]_p \cdot [\mathbf{x}_h]_p$ and use it to check correctness of circuit, i.e., the CheckZero procedure in $\mathsf{C}_0([y_h]_p, [\mathbf{x}_h]_p)$. $\mathcal{S}$ sends (fail, h) to $\mathcal{F}_{\mathsf{nZK}}$ if the check fails.
3. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{nZK}}$, and receives $\{(y_{h,0}, y_{h,1})\}_{h\in[n-2]}$ and the corresponding MAC tags from $\mathcal{A}$.
4. For $h \in [n-2]$, $i \in \{0,1\}$, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{nZK}}$ and receives R_i, $y_{h,i}$ and the corresponding MAC tags, and then check if $y_{h,i}$ is consistent with step 3, and compute the MAC tags required in RangeCheck and use it to check correctness of CheckZero procedure in range check. $\mathcal{S}$ sends (fail, h) to $\mathcal{F}_{\mathsf{nZK}}$ if any check fails.
5. For $h \in [n-2]$, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{nZK}}$ and receives C_1, z_h and the corresponding MAC tags, and then compute the MAC tag on $[y_{h,0}]_p + 2^s \cdot [y_{h,1}]_p - [y_h]_p$ and use it to check correctness of circuit, i.e., the CheckZero procedure in $\mathsf{C}_1([z_h]_p)$. $\mathcal{S}$ sends (fail, h) to $\mathcal{F}_{\mathsf{nZK}}$ if the check fails.
6. For $h \in [n-2]$, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{nZK}}$ and receives $[a_0, a_1]$, $y_{h,1}$) and the corresponding MAC tags, and then check if $y_{h,i}$ is consistent with step 3, and compute the MAC tags required in RangeCheck and use it to check correctness of CheckZero procedure in range check. $\mathcal{S}$ sends (fail, h) to $\mathcal{F}_{\mathsf{nZK}}$ if any check fails; otherwise $\mathcal{S}$ sends $\mathbf{x}_h$ and the corresponding MAC tags to $\mathcal{F}_{\mathsf{nZK}}$.

The view of adversary $\mathcal{A}$ simulated by $\mathcal{S}$ is perfect, which is computationally indistinguishable from the view of $\mathcal{A}$ in the real protocol execution.

E. Proof of L_∞ Norm Check

Malicious prover. Note that the clients act as provers in our protocols, while the server is the verifier. Thus, we assume that adversary $\mathcal{A}$ corrupts $n-2$ clients. Without loss of generality, assume that $\mathcal{A}$ corrupts clients $\mathcal{C}_0, \ldots, \mathcal{C}_{n-3}$. Note that simulator $\mathcal{S}$ extracts $\mathbf{x}_h$ and their corresponding MAC

[8]Our norm-check protocols achieve security against a malicious adversary in both client and server corruption cases, while our similarity-check protocols achieve security against a malicious adversary in the client-corruption case and a semi-honest adversary in the server-corruption case. Thus, the composed protocol Π_{SecAgg} is secure against a malicious adversary in the client-corruption case and a semi-honest adversary in the server-corruption case.

tags in the procedure of generating authenticated values $[\mathbf{x}_h]_p$ for $h \in [n-2]$, and can send them to $\mathcal{F}_{\mathsf{nZK}}$. $\mathcal{S}$ interacts with adversary $\mathcal{A}$ as follows:

1. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{nZK}}$ and receives $\{\hat{\mathbf{x}}_h\}_{h\in[n-2]}$ and the corresponding MAC tags from $\mathcal{A}$.
2. For $h \in [n-2]$, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{nZK}}$ and receives C, $\mathbf{x}_h$, $\hat{\mathbf{x}}_h$ and their corresponding MAC tags, and then check if $\mathbf{x}_h$ and $\hat{\mathbf{x}}_h$) are consistent with previous steps, and compute the MAC tags on $([\hat{\mathbf{x}}_h]_p + [\mathbf{x}_h]_p) * ([\hat{\mathbf{x}}_h]_p - [\mathbf{x}_h]_p)$ and use it to check correctness of circuit, i.e., the CheckZero procedure in $\mathsf{C}([\mathbf{x}_h]_p, [\hat{\mathbf{x}}_h]_p)$. $\mathcal{S}$ sends (fail, h) to $\mathcal{F}_{\mathsf{nZK}}$ if any check fails.
3. For $h \in [n-2]$, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{nZK}}$ and receives $[min, max]$, $\hat{\mathbf{x}}_h$ and the corresponding MAC tags, and then check if $\hat{\mathbf{x}}_h$ is consistent with previous steps, and compute the MAC tags required in RangeCheck and use it to check correctness of CheckZero procedure in range check. $\mathcal{S}$ sends (fail, h) to $\mathcal{F}_{\mathsf{nZK}}$ if any check fails; otherwise $\mathcal{S}$ sends $\mathbf{x}_h$ and the corresponding MAC tags to $\mathcal{F}_{\mathsf{nZK}}$.

The view of adversary $\mathcal{A}$ simulated by $\mathcal{S}$ is perfect, which is computationally indistinguishable from the view of $\mathcal{A}$ in the real protocol execution.

F. Committed-private Protocols

1) Additive Homomorphic Encryption: An additive homomorphic encryption scheme [63], [64], [65] $\mathsf{AHE} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ is a public key encryption scheme that additionally supports linear homomorphic operations on ciphertexts. We use $\mathbb{F}_p$ as the plaintext space.

- $\mathsf{KeyGen} \to (pk, sk)$. KeyGen is a randomized algorithm that samples a public key pk and corresponding secret key sk.
- $\mathsf{Enc}_{pk}(m) \to c$. Enc denotes the encryption operation that takes the public key pk and message $m \in \mathbb{F}_p$ as input, and outputs a ciphertext c.
- $\mathsf{Dec}_{sk}(c) \to m$. Dec denotes the decryption operation that takes the secret key sk and ciphertext c as input and outputs the message m.
- $\mathsf{Eval}_{pk}(c_1, c_2, L) \to c_3$. Eval refers to operations on ciphertexts that take the public key pk, two ciphertexts c_1, c_2 (corresponding to messages m_1 and m_2), and a linear function L as input, and outputs ciphertext c_3 (corresponding to $m_3 = L(m_1, m_2)$).

2) Additive Secret Sharing: We utilize an additive secret sharing scheme over field $\mathbb{F}_p$. A secret $x \in \mathbb{F}_p$ is shared as two random values $\langle x\rangle_0, \langle x\rangle_1 \in \mathbb{F}_p$ held by client $\mathcal{C}$ and server $\mathcal{S}$, respectively, which satisfy $x = \langle x\rangle_0 + \langle x\rangle_1 \mod p$. To simplify the notation, we sometimes use x_c and x_s to represent the share of x, held by the client and server, respectively. The security guarantees that given a share $\langle x\rangle_0$ or $\langle x\rangle_1$, the value x is perfectly hidden [66].

3) Vector Product: Firstly, it's important to clarify a point: the broad product protocol is simpler than the inner product protocol. This is because the multiplication in the broad product protocol involves multiplying each element of a vector by a scalar, whereas the inner product protocol entails multiplying the corresponding elements of two vectors and then summing these products. As a result, we only detail the inner product protocol.

Inner product. The detailed protocol $\Pi_{\mathsf{InnerProd}}$ is illustrated in Figure 14. We utilize the row-encoding-share-RaS multiplication to speed up our InnerProd protocol (for more details please refer to GALA [67]). In the beginning, the client $\mathcal{C}$ holds $[\mathbf{x}]_p$, a pair of keys (pk, sk) for AHE and the server $\mathcal{S}$ holds $[\mathbf{x}]_p$, $\mathbf{y}$, public key pk. Note that $\mathbf{x}, \mathbf{y} \in \mathbb{F}_p^d$. The client $\mathcal{C}$ computes $c_1 = \mathsf{Enc}_{pk_h}(\mathbf{x})$ and sends c_1 to the server $\mathcal{S}$. Note that the ZK proof along with c_1 aims to prove that c_1 is a valid sample from some x and x is known by the client (more details can be referred to [58], [37]). The server S randomly samples $r_1, r_2 \in \mathbb{F}_p$, $\mathbf{r_3} \in \mathbb{F}_p^d$, and sets $\langle z\rangle_1 = r_1$, $\langle \Delta z\rangle_1 = r_2$, $\mathbf{v_1} = \mathbf{r_3}$. In this case, r_1, r_2 are the secret sharing of output while $\mathbf{r_3}$ is a mask used for verifying the input $\mathbf{x}$. After receiving c_1, S computes $c_2 = \mathsf{Enc}_{pk_h}(\mathbf{x}\cdot\mathbf{y} - r_1)$, $c_3 = \mathsf{Enc}_{pk_h}(\Delta\cdot\mathbf{x}\cdot\mathbf{y} - r_2)$, $c_4 = \mathsf{Enc}_{pk_h}(\Delta\cdot\mathbf{x} - \mathbf{r_3})$ by invoking AHE.Eval and send c_2, c_3 and c_4 to $\mathcal{C}$. Then $\mathcal{C}$ decrypts c_2, c_3, c_4 and obtain $\langle z\rangle_0$, $\langle \Delta z\rangle_0$ and $\mathbf{v_0}$. At last, $\mathcal{C}$ and $\mathcal{S}$ check if $\mathsf{H}(\mathbf{K_x} + \mathbf{v_1}) = \mathsf{H}(\mathbf{M_x} - \mathbf{v_0})$, where $\mathsf{H}(\cdot)$ is a hash function modeled as a random oracle. If the check fails, then $\mathcal{S}$ aborts; otherwise $\mathcal{C}_h$ outputs $[z_{h,c}]_p$, and $\mathcal{S}$ outputs $[z_{h,c}]_p$ and $z_{h,s}$. Note that $M_h = \langle \Delta z_h\rangle_0 = \Delta z_{h,s} - \langle \Delta z_h\rangle_1 + \Delta z_{h,c} = K_h + \Delta z_{h,c}$.

Correctness: It is obvious that $z = \langle z\rangle_0 + \langle z\rangle_1 = (\mathbf{x}\cdot\mathbf{y} - r_1) + r_1 = \mathbf{x}\cdot\mathbf{y}$ and $\Delta z = \langle \Delta z\rangle_0 + \langle \Delta z\rangle_1 = (\Delta\mathbf{x}\cdot\mathbf{y} - r_2) + r_2 = \Delta\cdot\mathbf{x}\cdot\mathbf{y}$. The verification of $[\mathbf{x}]_p$ is to check $\mathbf{M_x} = \mathbf{K_x} + \Delta\cdot\mathbf{x}$. In the protocol, $\mathcal{C}$ computes $\mathsf{H}(\mathbf{M_x} - \mathbf{v_0}) = \mathsf{H}(\mathbf{M_x} - \mathbf{r_3})$, while $\mathcal{S}$ computes $\mathsf{H}(\mathbf{K_x} + \mathbf{v_1}) = \mathsf{H}(\mathbf{K_x} + \Delta\cdot\mathbf{x} - \mathbf{r_3})$. Due to the hash function $\mathsf{H}(\cdot)$, $\mathsf{H}(\mathbf{K_x} + \mathbf{v_1}) = \mathsf{H}(\mathbf{M_x} - \mathbf{v_0})$ can prove $\mathbf{M_x} = \mathbf{K_x} + \Delta\cdot\mathbf{x}$, i.e. $[\mathbf{x}]_p$ is valid.

Broad product. Our BroadProd protocol is shown in Figure 15, which is used to multiply each element of a vector $\mathbf{x}$ by a scalar b. This protocol is a simplified version of $\Pi_{\mathsf{InnerProd}}$ and its correctness can be proven readily in the same way.

4) LTZ Protocol: Protocol Π_{LTZ} is shown in Figure 17, which is based on $\mathcal{F}_{\mathsf{Non\text{-}lin}}$ shown in Figure 16 referred to SIMC [36] and is actually a comparison protocol based on GC. (Security proof of $\mathcal{F}_{\mathsf{Non\text{-}lin}}$ has been provided in SIMC [36])

G. Proof of Secure Aggregation

We provide the security proof for our secure aggregation protocol. It is straightforward to prove security for a semi-honest server. Hence, we only focus on the case of malicious clients, i.e. an arbitrary malicious adversary $\mathcal{A}$ controlling at most m clients (or provers in $\mathcal{F}_{\mathsf{nZK}}$), where m is constrained by Π_{PA}. We construct a simulator $\mathcal{S}$ given access to $\mathcal{F}_{\mathsf{nZK}}$, and run an adversary $\mathcal{A}$ as a subroutine.

Proof. **Malicious prover.** Assume that adversary $\mathcal{A}$ corrupts m clients, i.e. $\mathcal{C}_0, \ldots, \mathcal{C}_{m-1}$. Simulator $\mathcal{S}$ interacts with adversary $\mathcal{A}$ as follows.

Real execution Real: This is the real-world execution.

Initialize:

Subprotocol $\Pi_{\mathsf{InnerProd}}$

Parameters: A finite field $\mathbb{F}_p$, dimension of each vector d, a hash function $\mathsf{H} : \{0,1\}^* \to \{0,1\}^\kappa$ modeled as a random oracle.

Input: n clients $\{\mathcal{C}_h\}_{h\in[n]}$ hold $\{[\mathbf{x}_h]_p\}_{h\in[n]}$, n pairs of keys $\{(pk_h, sk_h)\}_{h\in[n]}$ for AHE. Server $\mathcal{S}$ holds $\{[\mathbf{x}_h]_p\}_{h\in[n]}$, $\mathbf{y}$, public keys $\{pk_h\}_{h\in[n]}$. ($\mathbf{x}_h \in \mathbb{F}_p^d$)

Protocol execution: n clients $\{\mathcal{C}_h\}_{h\in[n]}$ and server $\mathcal{S}$ compute $\langle z_h\rangle$ and $\langle \Delta z_h\rangle$ for $h \in [n]$ such that $z_h = \mathbf{x}_h \cdot \mathbf{y}$ as follows:

1) For $h \in [n]$, $\mathcal{C}_h$ computes the encryption $c_{h,1} = \mathsf{Enc}_{pk_h}(\mathbf{x}_h)$ and sends $c_{h,1}$ to $\mathcal{S}$ along with a zero-knowledge proof of plaintext knowledge of the ciphertext.
2) For $h \in [n]$, $\mathcal{S}$ samples $r_{h,1}, r_{h,2} \in \mathbb{F}_p$, $\mathbf{r}_{h,3} \in \mathbb{F}_p^d$ randomly, and sets $\langle z_h\rangle_1 = r_{h,1}$, $\langle \Delta z_h\rangle_1 = r_{h,2}$, $\mathbf{v}_{h,1} = \mathbf{r}_{h,3}$.
3) $\mathcal{S}$ receives $\{c_{h,1}\}_{h\in[n]}$. For $h \in [n]$, $\mathcal{S}$ evaluates $c_{h,2} = \mathsf{Enc}_{pk_h}(\mathbf{x}_h \cdot \mathbf{y} - r_{h,1})$, $c_{h,3} = \mathsf{Enc}_{pk_h}(\Delta \cdot \mathbf{x}_h \cdot \mathbf{y} - r_{h,2})$, $c_{h,4} = \mathsf{Enc}_{pk_h}(\Delta \cdot \mathbf{x}_h - \mathbf{r}_{h,3})$ by $\mathsf{AHE.Eval}$.
4) For $h \in [n]$, $\mathcal{S}$ sends $c_{h,2}$, $c_{h,3}$ and $c_{h,4}$ to $\mathcal{C}_h$.
5) For $h \in [n]$, $\mathcal{C}_h$ decrypts $\langle z_h\rangle_0 = \mathsf{Dec}_{sk_h}(c_{h,2})$, $\langle \Delta z_h\rangle_0 = \mathsf{Dec}_{sk_h}(c_{h,3})$, $\mathbf{v}_{h,0} = \mathsf{Dec}_{sk_h}(c_{h,4})$.
6) For $h \in [n]$, $\mathcal{C}_h$ computes $a_h = \mathsf{H}(\mathbf{M}_h - \mathbf{v}_{h,0})$ and sends a_h to $\mathcal{S}$.
7) For $h \in [n]$, $\mathcal{S}$ checks that $\mathsf{H}(\mathbf{K}_h + \mathbf{v}_{h,1}) = a_h$. If any check fails, $\mathcal{S}$ aborts.
8) For $h \in [n]$, $\mathcal{C}_h$ outputs $z_{h,c} = \langle z_h\rangle_0$ and $M_h = \langle \Delta z_h\rangle_0$, and $\mathcal{S}$ outputs $z_{h,s} = \langle z_h\rangle_1$ and $K_h = \Delta z_{h,s} - \langle \Delta z_h\rangle_1$. (i.e., $\mathcal{C}_h$ outputs $[z_{h,c}]_p$, and $\mathcal{S}$ outputs $[z_{h,c}]_p$ and $z_{h,s}$)

Fig. 14: Subprotocol for inner product.

Subprotocol $\Pi_{\mathsf{BroadProd}}$

Parameters: A finite field $\mathbb{F}_p$, dimension of each vector d, a hash function $\mathsf{H} : \{0,1\}^* \to \{0,1\}^\kappa$ modeled as a random oracle.

Input: n clients $\{\mathcal{C}_h\}_{h\in[n]}$ hold $\{[\mathbf{x}_h]_p\}_{h\in[n]}$, n pairs of keys $\{(pk_h, sk_h)\}_{h\in[n]}$ for AHE. Server $\mathcal{S}$ holds $\{[\mathbf{x}_h]_p\}_{h\in[n]}$, $\{b_h\}_{h\in[n]}$, public keys $\{pk_h\}_{h\in[n]}$. ($\mathbf{x}_h \in \mathbb{F}_p^d$)

Protocol execution: n clients $\{\mathcal{C}_h\}_{h\in[n]}$ and server $\mathcal{S}$ compute $\langle \mathbf{z}_h\rangle$ and $\langle \Delta \mathbf{z}_h\rangle$ for $h \in [n]$ such that $\mathbf{z}_h = b_h \cdot \mathbf{x}_h$ as follows:

1) For $h \in [n]$, $\mathcal{C}_h$ computes the encryption $c_{h,1} = \mathsf{Enc}_{pk_h}(\mathbf{x}_h)$ and sends $c_{h,1}$ to $\mathcal{S}$ along with a zero-knowledge proof of plaintext knowledge of the ciphertext.
2) For $h \in [n]$, $\mathcal{S}$ samples $\mathbf{r}_{h,1}, \mathbf{r}_{h,2}, \mathbf{r}_{h,3} \in \mathbb{F}_p^d$ randomly, and sets $\langle \mathbf{z}_h\rangle_1 = r_{h,1}$, $\langle \Delta \mathbf{z}_h\rangle_1 = r_{h,2}$, $\mathbf{v}_{h,1} = \mathbf{r}_{h,3}$.
3) $\mathcal{S}$ receives $\{c_{h,1}\}_{h\in[n]}$. For $h \in [n]$, $\mathcal{S}$ evaluates $c_{h,2} = \mathsf{Enc}_{pk_h}(b_h \cdot \mathbf{x}_h - \mathbf{r}_{h,1})$, $c_{h,3} = \mathsf{Enc}_{pk_h}(\Delta \cdot b_h \cdot \mathbf{x}_h - \mathbf{r}_{h,2})$, $c_{h,4} = \mathsf{Enc}_{pk_h}(\Delta \cdot \mathbf{x}_h - \mathbf{r}_{h,3})$ by $\mathsf{AHE.Eval}$.
4) For $h \in [n]$, $\mathcal{S}$ sends $c_{h,2}$, $c_{h,3}$ and $c_{h,4}$ to $\mathcal{C}_h$.
5) For $h \in [n]$, $\mathcal{C}_h$ decrypts $\langle \mathbf{z}_h\rangle_0 = \mathsf{Dec}_{sk_h}(c_{h,2})$, $\langle \Delta \mathbf{z}_h\rangle_0 = \mathsf{Dec}_{sk_h}(c_{h,3})$, $\mathbf{v}_{h,0} = \mathsf{Dec}_{sk_h}(c_{h,4})$.
6) For $h \in [n]$, $\mathcal{C}_h$ computes $a_h = \mathsf{H}(\mathbf{M}_h - \mathbf{v}_{h,0})$ and sends a_h to $\mathcal{S}$.
7) For $h \in [n]$, $\mathcal{S}$ checks that $\mathsf{H}(\mathbf{K}_h + \mathbf{v}_{h,1}) = a_h$. If any check fails, $\mathcal{S}$ aborts.
8) For $h \in [n]$, $\mathcal{C}_h$ outputs $\mathbf{z}_{h,c} = \langle \mathbf{z}_h\rangle_0$ and $\mathbf{M}_h = \langle \Delta \mathbf{z}_h\rangle_0$, and $\mathcal{S}$ outputs $\mathbf{z}_{h,s} = \langle \mathbf{z}_h\rangle_1$ and $\mathbf{K}_h = \Delta \mathbf{z}_{h,s} - \langle \Delta \mathbf{z}_h\rangle_1$. (i.e., $\mathcal{C}_h$ outputs $[\mathbf{z}_{h,c}]_p$, and $\mathcal{S}$ outputs $[\mathbf{z}_{h,c}]_p$ and $\mathbf{z}_{h,s}$)

Fig. 15: Subprotocol for broad product.

- $\mathcal{S}$ samples $\Delta \xleftarrow{\$} \mathbb{F}_p$.
- $\mathcal{S}$ initializes thresholds τ_1 for Π_{EucSim} and τ_2 for Π_{CosSim}.

Input:

- $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{nZK}}$ and receives $\{\mathbf{x}_h\}_{h\in[m]}$ and their corresponding MAC tags from $\mathcal{A}$, and naturally define the corresponding keys.

InputCheck:

There are two different types of input integrity check protocols, i.e., EucSim and CosSim. Thus, we divide the proof of this part into two cases:

Case I: EucSim. The detailed subprotocol is shown in Figure 10.

- $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{nZK}}$ and receives $\{[\mathbf{x}_h]\}_{h\in[m]}$ from $\mathcal{A}$. $\mathcal{S}$ check if $[\mathbf{x}_h]_p$ is valid and $\mathbf{x}_h[i] \in [min, max]$ for $h \in [m]$, $i \in [d]$, and sends (fail, h) to $\mathcal{F}_{\mathsf{SecAgg}}$ if any check fails.

  Hybrid execution Hyb_1: This is the same as Real except the above execution of simulator $\mathcal{S}$.

Claim 1. Real *and* Hyb_1 *are indistinguishable.*

Proof. The distinguishability follows the security proof of $\Pi_{\ell_\infty \mathsf{Check}}$ in Appendix E. □

- $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{nZK}}$ and receives $\{z_h\}_{h\in[m]}$ and their corresponding MAC tags from $\mathcal{A}$, and naturally define the corresponding keys.
- $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{nZK}}$ and receives $\{(\mathsf{C}, [z_h]_p, [\mathbf{x}_h]_p)\}_{h\in[m]}$, and then check if $\mathsf{C}([z_h]_p, [\mathbf{x}_h]_p) = 1$ for $h \in [m]$. $\mathcal{S}$ sends (fail, h) to $\mathcal{F}_{\mathsf{SecAgg}}$ if the check fails.

  Hybrid execution Hyb_2: This is the same as Hyb_1 except the above execution of simulator $\mathcal{S}$.

Claim 2. Hyb_1 *and* Hyb_2 *are indistinguishable.*

Proof. The distinguishability follows the security of the CirEval in functionality $\mathcal{F}_{\mathsf{nZK}}$ (refer to Theorem 2 and 3 in Quicksilver [43]). □

Functionality $\mathcal{F}_{\mathsf{Non\text{-}lin}}$

This functionality is parameterized by a prime $p \geq 2^\lambda$.
Non-lin: On receiving $(\mathsf{C}, \langle x \rangle_0)$ from clients $\mathcal{C}$ and $(\mathsf{C}, \langle x \rangle_1, \Delta)$ from a server $\mathcal{S}$, where $x \in \mathbb{F}_p$ and C is a boolean circuit for a function f. Compute Δx, $y = f(x)$ and Δy. Then, uniformly sample $(\langle \Delta x \rangle_0, \langle y \rangle_0, \langle \Delta y \rangle_0) \xleftarrow{\$} \mathbb{F}_p^3$ and set $\langle \Delta x \rangle_1 = \Delta x - \langle \Delta x \rangle_0$, $\langle y \rangle_1 = y - \langle y \rangle_0$, $\langle \Delta y \rangle_1 = \Delta y - \langle \Delta y \rangle_0$. Finally, send $(\langle \Delta x \rangle, \langle y \rangle, \langle \Delta y \rangle)$ to $\mathcal{C}$ and $\mathcal{S}$.

Fig. 16: Ideal functionality for Non-lin.

Subprotocol Π_{LTZ}

Parameters: A finite field $\mathbb{F}_p$. Two hash functions $\mathsf{H}_\ell : \{0,1\}^* \to \{0,1\}^\ell$ and $\mathsf{H}_{2\ell} : \{0,1\}^* \to \{0,1\}^{2\ell}$, modeled as random oracles. A boolean circuit Comp^f that takes additive shares of $x \in \mathbb{F}_p$, i.e., $\langle x \rangle_0$, $\langle x \rangle_1$, as input and outputs $(x, f(x))$, where $f(x) = 1\{x < 0\}$.
Input: n clients $\{\mathcal{C}_h\}_{h\in[n]}$ hold $\{[e_h]_p\}_{h\in[n]}$. Server $\mathcal{S}$ holds $\{[e_h]_p\}_{h\in[n]}$, $\{\tau_h\}_{h\in[n]}$. $\{\mathcal{C}_h\}_{h\in[n]}$ set $\langle x_h \rangle_0 = e_h$, $\langle \Delta x_h \rangle_0 = M_h$, and $\mathcal{S}$ sets $\langle x_h \rangle_1 = -\tau_h$, $\langle \Delta x_h \rangle_1 = -\Delta\tau_h - K_h$.
Protocol execution: n clients $\{\mathcal{C}_h\}_{h\in[n]}$ and server $\mathcal{S}$ compute $\langle b_h \rangle$, $\langle \Delta b_h \rangle$ such that $b_h = f(x_h) = f(e_h - \tau_h) = 1\{e_h - \tau_h < 0\}$ for $h \in [n]$ as follows:

1) For $h \in [n]$, $\mathcal{C}_h$ sends $(\mathsf{Comp}^f, \langle x_h \rangle_0)$ and $\mathcal{S}$ sends $(\mathsf{Comp}^f, \langle x_h \rangle_1, \Delta)$ to $\mathcal{F}_{\mathsf{Non\text{-}lin}}$, which returns $(\langle \Delta x_h \rangle', \langle b_h \rangle, \langle \Delta b_h \rangle)$.
2) $\mathcal{C}_h$ sends $\langle \Delta x_h \rangle_0 - \langle \Delta x_h \rangle'_0$ to $\mathcal{S}$.
3) $\mathcal{S}$ checks that $\langle \Delta x_h \rangle'_1 - \langle \Delta x_h \rangle_1 = \langle \Delta x_h \rangle_0 - \langle \Delta x_h \rangle'_0$ for $h \in [n]$. If any check fails, $\mathcal{S}$ aborts.
4) For $h \in [n]$, $\mathcal{C}_h$ outputs $b_h^{\mathcal{C}} = \langle b_h \rangle_0$ and $M_h = \langle \Delta b_h \rangle_0$, and $\mathcal{S}$ outputs $b_h^{\mathcal{S}} = \langle b_h \rangle_1$ and $K_h = \Delta b_h^{\mathcal{S}} - \langle \Delta b_h \rangle_1$. (i.e., $\mathcal{C}_h$ outputs $[b_h^{\mathcal{C}}]_p$, and $\mathcal{S}$ outputs $[b_h^{\mathcal{C}}]_p$ and $b_h^{\mathcal{S}}$).

Fig. 17: Less than zero subprotocol

- $\mathcal{S}$ uniformly samples $\mathbf{x}'_s \xleftarrow{\$} \mathbb{F}_p^d$, and executes $\Pi_{\mathsf{InnerProd}}$ (see Figure 14) as an honest server with $\mathcal{A}$, using $[\mathbf{x}_h]_p$ (i.e., Δ and the keys defined above), $\mathbf{x}'_s$ and $\{pk_h\}_{h\in[m]}$. (Note that $\mathcal{S}$ can obtain $\langle y_h \rangle_1$ and $\langle \Delta y_h \rangle_1$ as $\mathcal{S}$ samples them uniformly, and thus $\mathcal{S}$ can obtain $[y_h^{\mathcal{C}}]_p$ and $y_h^{\mathcal{S}}$)
  Hybrid execution Hyb_3: This is the same as Hyb_2 except the above execution of simulator $\mathcal{S}$.

  Claim 3. *If* H *is a random oracle (collision-resistant),* Hyb_2 *and* Hyb_3 *are indistinguishable.*

  Proof. Hyb_3 is identical to Hyb_2, except the hash function H, as the $r_{h,1}$, $r_{h,2}$ and $\mathbf{r}_{h,3}$ are sampled uniformly at random, which makes $\langle z_h \rangle_0$, $\langle \Delta z_h \rangle_0$ and $\mathbf{v}_{h,0}$ distributed uniformly. First, we consider the probability that $\mathcal{A}$ distinguishes between the Hyb_2 and Hyb_3 for a corrupted client $\mathcal{C}_h$. In Hyb_2, if $\mathcal{C}_h$ computes $a'_h = \mathsf{H}(\mathbf{M}_h - \mathbf{v}_{h,0} + \mathbf{e}_h)$, where $\mathbf{e}_h$ is the error chosen by $\mathcal{A}$. The probability that $a'_h = a_h$ is $\mathsf{negl}(\kappa)$. In Hyb_3, $\mathcal{S}$ outputs would abort once $\mathbf{e}_h \neq 0$. Therefore, adversary $\mathcal{A}$ can distinguish between Hyb_2 and Hyb_3 with probability $\mathsf{negl}(\kappa)$ if $\mathbf{e}_h \neq 0$. Further, for (m) corrupted provers, Hyb_2 and Hyb_3 are indistinguishable. □
- $\mathcal{S}$ executes Π_{LTZ} with $\mathcal{A}$, where $\mathcal{S}$ acts as an honest verifier with input $\{([e_h]_p, \tau'_h)\}_{h\in[m]}$ ($\mathcal{S}$ can do this because it knows $[z_h]_p$, $[y_{h,c}]_p$ and $\tau'_h = \tau_1^2 - \mathbf{x}'_s \cdot \mathbf{x}'_s + 2y_h^{\mathcal{S}}$). In the execution of Π_{LTZ} (shown in Figure 17), whose input is denoted as $\{[e_h]_p, \tau_h\}_{h\in[m]}$, $\mathcal{S}$ does as follows:
  a. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{Non\text{-}lin}}$ and receives $\{e_h\}_{h\in[m]}$ from $\mathcal{A}$.
  b. $\mathcal{S}$ reconstructs e_h and computes $b_h = 1\{e_h < 0\}$, $h \in [m]$.
  c. For $h \in [m]$, $\mathcal{S}$ sends $\langle \Delta e_h \rangle$ to $\mathcal{F}_{\mathsf{Non-lin}}$ and receives back $(\langle \Delta e_h \rangle'_0, \langle b_h \rangle_0, \langle \Delta b_h \rangle_0)$.
  d. For $h \in [m]$, $\mathcal{S}$ computes $\langle \Delta e_h \rangle'_1 = \Delta e_h - \langle \Delta e_h \rangle'_0$, $\langle b_h \rangle_1 = b_h - \langle b_h \rangle_0$, $\langle \Delta b_h \rangle_1 = \Delta b_h - \langle \Delta b_h \rangle_0$.
  e. $\mathcal{S}$ sends $\{(\langle \Delta e_h \rangle'_0, \langle b_h \rangle_0, \langle \Delta b_h \rangle_0)\}_{h\in[m]}$ to $\mathcal{A}$.
  f. $\mathcal{S}$ executes the rest of the protocol as an honest server.

  Hybrid execution Hyb_4: This is the same as Hyb_3 except the above execution of simulator $\mathcal{S}$.

  Claim 4. Hyb_3 *and* Hyb_4 *are indistinguishable.*

  Proof. The distinguishability follows the security proof of $\Pi_{\mathsf{Non\text{-}lin}}$ referred to [36], except the probability that $\mathcal{A}$ knows Δ and use $(\langle e_h \rangle_0 + e, \langle \Delta e_h \rangle_0 + \Delta e)$ as input for Π_{LTZ}, where e is an error chosen by $\mathcal{A}$. The probability that $\mathcal{A}$ knows Δ is at most $1/p$, which is $\mathsf{negl}(\lambda)$. Thus, Hyb_3 and Hyb_4 are indistinguishable. □
- $\mathcal{S}$ executes the rest of the protocol as an honest server.
  Hybrid execution Hyb_5: This is the same as Hyb_4 except the above execution of simulator $\mathcal{S}$.

  Claim 5. Hyb_4 *and* Hyb_5 *are indistinguishable.*

  Proof. Hyb_4 and Hyb_5 are identical, except the probability that $\mathcal{A}$ knows Δ. The probability that $\mathcal{A}$ knows Δ is at most $1/p$, which is $\mathsf{negl}(\lambda)$. Thus, Hyb_4 and Hyb_5 are indistinguishable. □

Case II: CosSim. The detailed subprotocol is shown in Figure 11.

- $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{nZK}}$ and receives $\{[\mathbf{x}_h]_p\}_{h\in[m]}$from $\mathcal{A}$. $\mathcal{S}$ check if $[\mathbf{x}_h]_p$ is valid and $\mathbf{x}_h[i] \in [min, max]$ for $h \in [m]$, $i \in [d]$. $\mathcal{S}$ sends (fail, h) to $\mathcal{F}_{\mathsf{SecAgg}}$ if any check fails.
  Hybrid execution Hyb_1: This is the same Real except the above execution of simulator $\mathcal{S}$.

  Claim 6. Real *and* Hyb_1 *are indistinguishable.*

  Proof. The distinguishability follows the security proof of $\Pi_{\ell_\infty \mathsf{Check}}$ in Appendix E. □
- $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{nZK}}$ and receives $\{\bar{\mathbf{x}}_h, ||\mathbf{x}_h||_2\}_{h\in[m]}$ and their corresponding MAC tags from $\mathcal{A}$, and naturally define the corresponding keys.

- $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{nZK}}$ and receives $\{[||\mathbf{x}_h||_2]_p\}_{h\in[m]}$ from $\mathcal{A}$. Then, $\mathcal{S}$ checks if $[||\mathbf{x}_h||_2]_p$ is valid and $||\mathbf{x}_h||_2 \in [0, bd_2]$. $\mathcal{S}$ sends (fail, h) to $\mathcal{F}_{\mathsf{SecAgg}}$ if any check fails.
  Hybrid execution Hyb_6: This is the same as Hyb_1 except the above execution of simulator $\mathcal{S}$.

  Claim 7. Hyb_1 *and* Hyb_6 *are indistinguishable.*

  Proof. The distinguishability follows the security of the RangeCheck in functionality $\mathcal{F}_{\mathsf{nZK}}$ (refer to Theorem 1 in [50]). □
- $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{nZK}}$ and receives $\{[\bar{\mathbf{x}}_h]_p\}_{h\in[m]}$ from $\mathcal{A}$. $\mathcal{S}$ check if $[\bar{\mathbf{x}}_h]_p$ is valid for $h \in [m]$ and $||\bar{\mathbf{x}}_h||_2 \in [2^s - e, 2^s + e]$. $\mathcal{S}$ sends (fail, h) to $\mathcal{F}_{\mathsf{SecAgg}}$ if any check fails.
  Hybrid execution Hyb_7: This is the same as Hyb_6 except the above execution of simulator $\mathcal{S}$.

  Claim 8. Hyb_6 *and* Hyb_7 *are indistinguishable.*

  Proof. The distinguishability follows the security proof of $\Pi_{\ell_2\mathsf{Check}}$ in Appendix D. □
- $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{nZK}}$ and receives $\{(\mathsf{C}, [\mathbf{x}_h]_p, [\bar{\mathbf{x}}_h]_p, [||\mathbf{x}_h||_2]_p)\}_{h\in[m]}$, and then check if $\mathsf{C}(\{[\mathbf{x}_h]_p, [\bar{\mathbf{x}}_h]_p, [||\mathbf{x}_h||_2]_p\}_{h\in[m]}) = 1$. $\mathcal{S}$ sends (fail, h) to $\mathcal{F}_{\mathsf{SecAgg}}$ if the check fails.
  Hybrid execution Hyb_8: This is the same as Hyb_7 except the above execution of simulator $\mathcal{S}$.

  Claim 9. Hyb_7 *and* Hyb_8 *are indistinguishable.*

  Proof. The distinguishability follows the security of the CirEval in functionality $\mathcal{F}_{\mathsf{nZK}}$ (refer to Theorem 2 and 3 in Quicksilver [43]) □
- $\mathcal{S}$ uniformly samples $\mathbf{x}'_s \xleftarrow{\$} \mathbb{F}_p^d$, normalizes $\mathbf{x}'_s$ to unit length $\bar{\mathbf{x}}'_s = \frac{\mathbf{x}'_s}{||\mathbf{x}'_s||_2}$, and executes $\Pi_{\mathsf{InnerProd}}$ as an honest server with $\mathcal{A}$, using $[\bar{\mathbf{x}}_h]_p$ (i.e., Δ and the keys defined above), $\mathbf{x}'_s$ and $\{pk_h\}_{h\in[m]}$. (Note that $\mathcal{S}$ can obtain $\langle y_h\rangle_1$ and $\langle \Delta y_h\rangle_1$ as $\mathcal{S}$ samples them uniformly, and thus $\mathcal{S}$ can obtain $[y_h^{\mathcal{C}}]_p$ and $y_h^{\mathcal{S}}$)
  Hybrid execution Hyb_9: This is the same as Hyb_8 except the above execution of simulator $\mathcal{S}$.

  Claim 10. *If* H *is a random oracle (collision-resistant),* Hyb_8 *and* Hyb_9 *are indistinguishable.*

  Proof. Hyb_9 is identical to Hyb_8, except the hash function H, as the $r_{h,1}$, $r_{h,2}$ and $\mathbf{r}_{h,3}$ are sampled uniformly at random, which makes $\langle z_h\rangle_0$, $\langle \Delta z_h\rangle_0$ and $\mathbf{v}_{h,0}$ distributed uniformly. First, we consider the probability that $\mathcal{A}$ distinguishes between the Hyb_2 and Hyb_3 for a corrupted client $\mathcal{C}_h$. In Hyb_8, if $\mathcal{C}_h$ computes $a'_h = \mathsf{H}(\mathbf{M}_h - \mathbf{v}_{h,0} + \mathbf{e}_h)$, where $\mathbf{e}_h$ is the error chosen by $\mathcal{A}$. The probability that $a'_h = a_h$ is $\mathsf{negl}(\kappa)$. In Hyb_9, $\mathcal{S}$ outputs would abort once $\mathbf{e}_h \neq 0$. Therefore, adversary $\mathcal{A}$ can distinguish between Hyb_8 and Hyb_9 with probability $\mathsf{negl}(\kappa)$ if $\mathbf{e}_h \neq 0$. Further, for (m) corrupted provers, Hyb_8 and Hyb_9 are indistinguishable. □
- $\mathcal{S}$ executes Π_{LTZ} with $\mathcal{A}$, where $\mathcal{S}$ acts as an honest verifier with input $\{(-[y_h^{\mathcal{C}}]_p, y_h^{\mathcal{S}} - \tau_2)\}_{h\in[m]}$. In the execution of Π_{LTZ} (shown in Figure 17), whose input is denoted as $\{[e_h]_p, \tau_h\}_{h\in[m]}$, $\mathcal{S}$ does as follows:
  a. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{Non\text{-}lin}}$ and receives $\{e_h\}_{h\in[m]}$ from $\mathcal{A}$.
  b. $\mathcal{S}$ reconstructs e_h and computes $b_h = 1\{x_h < 0\}$, $h \in [m]$.
  c. For $h \in [m]$, $\mathcal{S}$ sends $\langle \Delta e_h\rangle$ to $\mathcal{F}_{\mathsf{Non-lin}}$ and receives back $(\langle \Delta e_h\rangle'_0, \langle b_h\rangle_0, \langle \Delta b_h\rangle_0)$.
  d. For $h \in [m]$, $\mathcal{S}$ computes $\langle \Delta e_h\rangle'_1 = \Delta e_h - \langle \Delta e_h\rangle'_0$, $\langle b_h\rangle_1 = b_h - \langle b_h\rangle_0$, $\langle \Delta b_h\rangle_1 = \Delta b_h - \langle \Delta b_h\rangle_0$.
  e. $\mathcal{S}$ sends $\{(\langle \Delta e_h\rangle'_0, \langle b_h\rangle_0, \langle \Delta b_h\rangle_0)\}_{h\in[m]}$ to $\mathcal{A}$.
  f. $\mathcal{S}$ executes the rest of the protocol as an honest server.

  Hybrid execution Hyb_{10}: This is the same as Hyb_9 except the above execution of simulator $\mathcal{S}$.

  Claim 11. Hyb_9 *and* Hyb_{10} *are indistinguishable.*

  Proof. The distinguishability follows the security proof of $\Pi_{\mathsf{Non\text{-}lin}}$ referred to [36], except the probability that $\mathcal{A}$ knows Δ and use $(\langle e_h\rangle_0 + e, \langle \Delta e_h\rangle_0 + \Delta e)$ as input for Π_{LTZ}, where e is an error chosen by $\mathcal{A}$. The probability that $\mathcal{A}$ knows Δ is at most $1/p$, which is $\mathsf{negl}(\lambda)$. Thus, Hyb_9 and Hyb_{10} are indistinguishable. □
- $\mathcal{S}$ executes the rest of the protocol as an honest server.
  Hybrid execution Hyb_{11}: This is the same as Hyb_{10} except the above execution of simulator $\mathcal{S}$.

  Claim 12. Hyb_{10} *and* Hyb_{11} *are indistinguishable.*

  Proof. Hyb_{10} and Hyb_{11} are identical, except the probability that $\mathcal{A}$ knows Δ and sends $(\langle b_h\rangle_0 + e, \langle \Delta b_h\rangle_0 + \Delta e)$, where e is an error chosen by $\mathcal{A}$. The probability that $\mathcal{A}$ knows Δ is at most $1/p$, which is $\mathsf{negl}(\lambda)$. Thus, Hyb_{10} and Hyb_{11} are indistinguishable. □

BroadProd:

- Note that $\mathcal{S}$ knows b_h in the execution of Π_{LTZ} in either EucSim or CosSim. $\mathcal{S}$ executes $\Pi_{\mathsf{BroadProd}}$ (see Figure 15) as an honest server with $\mathcal{A}$, using $[\mathbf{x}_h]_p$ (i.e., Δ and the keys defined above), $\mathbf{x}'_s$ and $\{pk_h\}_{h\in[m]}$. (Note that $\mathcal{S}$ can obtain $\langle \mathbf{y}_h\rangle_1$ and $\langle \Delta \mathbf{y}_h\rangle_1$ as $\mathcal{S}$ samples them uniformly, and thus $\mathcal{S}$ can obtain $[\mathbf{y}_h^{\mathcal{C}}]_p$ and $\mathbf{y}_h^{\mathcal{S}}$)
  Hybrid execution Hyb_{12}: This is the same as Hyb_5 (or Hyb_{11}) except the above execution of simulator $\mathcal{S}$.

  Claim 13. *If* H *is a random oracle (collision-resistant),* Hyb_5 *and* Hyb_{12} *are indistinguishable (or* Hyb_{11} *and* Hyb_{12} *are indistinguishable).*

  Proof. Hyb_{12} is identical to Hyb_5 (or Hyb_{11}), except the hash function H, as the $\mathbf{r}_{h,1}$, $\mathbf{r}_{h,2}$ and $\mathbf{r}_{h,3}$ are sampled uniformly at random, which makes $\langle \mathbf{z}_h\rangle_0$, $\langle \Delta \mathbf{z}_h\rangle_0$ and $\mathbf{v}_{h,0}$ distributed uniformly. First, we consider the probability that $\mathcal{A}$ distinguishes between the Hyb_5 and Hyb_{12} (or Hyb_{11} and Hyb_{12}) for a corrupted client $\mathcal{C}_h$. In Hyb_5 (or Hyb_{11}), if $\mathcal{C}_h$ computes $a'_h = \mathsf{H}(\mathbf{M}_h - \mathbf{v}_{h,0} + \mathbf{e}_h)$, where $\mathbf{e}_h$ is the error chosen by $\mathcal{A}$. The probability that $a'_h = a_h$ is $\mathsf{negl}(\kappa)$. In Hyb_{12}, $\mathcal{S}$ outputs would abort once $\mathbf{e}_h \neq 0$. Therefore, adversary $\mathcal{A}$ can distinguish between Hyb_5 and Hyb_{12} (or Hyb_{11} and Hyb_{12}) with probability $\mathsf{negl}(\kappa)$ if $\mathbf{e}_h \neq 0$. Further, for (m) corrupted provers, Hyb_5 and Hyb_{12} (or Hyb_{11} and Hyb_{12}) are indistinguishable. □

Summation:

- The simulator $\mathcal{S}$ simulates the summation step. The privacy and dropout-resilience of the underlying secure aggregation protocol Π_{PA} against actively adversarial users are established in [1] (Theorem 6.4). If the simulated honest server aborts, $\mathcal{S}$ sends $\perp$ to $\mathcal{F}_{\mathsf{SecAgg}}$ and aborts. If the honest server outputs success, $\mathcal{S}$ sends the extracted inputs $\{\mathbf{x}_h\}_{h\in[m]}$ to $\mathcal{F}_{\mathsf{SecAgg}}$.
  Hybrid execution Hyb_{13}: This is the same as Hyb_{12} except the above execution of simulator $\mathcal{S}$.

  Claim 14. Hyb_{13} *and* Hyb_{12} *are indistinguishable.*

  Proof. The indistinguishability follows from the security of Π_{PA} and the unforgeability of IT-MACs. According to Theorem 6.4 in [1], the protocol ensures input privacy even in the presence of actively adversarial users. The only remaining case for a distinguisher is a successful forgery of the IT-MAC by a malicious client.
  Suppose an adversary $\mathcal{A}$ attempts to inject an error $\mathbf{e} \neq \mathbf{0}$ by sending $(\mathbf{y}'_{h,c} + \mathbf{e}, \mathbf{M}'_h + \mathbf{e}')$ as inputs to Π_{PA}. To pass the integrity check $\mathbf{M} = \mathbf{K} + \Delta\mathbf{y}^{\mathcal{C}}$ at the server, the adversary must provide a modified MAC such that $\mathbf{e}' = \Delta\mathbf{e}$. Since Δ is chosen uniformly at random from $\mathbb{F}_p$, the probability that $\mathcal{A}$ correctly guesses Δ is $1/p$, which is $\mathsf{negl}(\lambda)$. Therefore, Hyb_{12} and Hyb_{13} are indistinguishable. □

$\mathcal{A}$'s view in Hyb_{13} is identical to its view in the ideal-world execution, i.e. Sim. Besides, the outputs of the honest parties (i.e. the only honest client $\mathcal{C}_m$ and server $\mathcal{S}$) in the ideal-world execution and in the real-world execution are the same.

This concludes the proof. □

H. Security Flaws in the LZKSA Scheme

This chapter provides a security analysis of the RLWE-based commitment scheme and its accompanying zero-knowledge proof protocols as proposed in LZKSA [31]. Our analysis reveals that the scheme suffers from a fundamental vulnerability in its core commitment mechanism, namely a **failure of the binding property**. Furthermore, the zero-knowledge proof protocols designed for data validation exhibit critical design flaws.

1) Failure of the Binding Property in the Core Commitment Scheme: The foundation of the LZKSA protocol [31] is its proposed RLWE-based commitment scheme in Section 4. However, this scheme fails to satisfy the binding property, which is one of the most critical security requirements for a commitment.

a) Scheme Review and Flawed Security Proof: We first provide a detailed review of the commitment scheme as defined in LZKSA [31], followed by an analysis of its flawed security proof.

Commitment Scheme Definition. The scheme is defined by the following three algorithms: Gen, Commit, and open.

- `Gen`: The key generation algorithm produces a public commitment key pk, which consists of randomly chosen $a \in R_q^d$, where R_q is a polynomial ring $\mathbb{Z}_q[X]/\langle X^n+1\rangle$. The plaintext domain is the ring R_t^d.
- `Com`: To commit to a message $x \in R_t^d$, the committer draws a secret polynomial $s \in R_q^d$ and a noise polynomial e that is conditioned on its infinity norm being bounded, i.e., $\|e\|_\infty \leq n$. The commitment c is then computed as:
$$c := \mathrm{Com}(s, x, e) = as + x + te \in R_q^d$$
  The information required to open the commitment is the tuple (x, s, e).
- `Ver`: To verify an opening, a verifier takes the commitment c, a claimed message x', randomness s', and noise e'. The verifier accepts if and only if the equation $as' + x' + te' = c$ holds true.

Analysis of the Flawed Proof: It is a common and natural practice to construct a commitment scheme from a public-key encryption scheme [3], but this practice is not applicable to symmetric-key schemes, like the one-time pad, which inherently lacks the binding property. The commitment scheme in LZKSA [31] originates from such a symmetric-key encryption scheme, drawing from an RLWE-based symmetric encryption scheme presented in ACORN [4]. This reliance on a symmetric paradigm strongly suggests a potential failure of the binding property, which our analysis confirms.

The security proof for the binding property presented in Section 4 of LZKSA [31] contains a fundamental logical fallacy, which can be broken down as follows:

- **The Collision Equation**: The proof correctly establishes that an adversary capable of breaking the binding property must be able to find two distinct openings, (s_0, x_0, e_0) and (s_1, x_1, e_1) (where $x_0 \neq x_1$), for the same commitment c. This leads to the core equation:
$$a(s_0 - s_1) + (x_0 - x_1) + t(e_0 - e_1) \equiv 0 \pmod q$$
- **LZKSA's Core Argument**: The paper then claims that finding a non-zero solution $(\Delta s, \Delta x, \Delta e) = (s_0 - s_1, x_0 - x_1, e_0 - e_1)$, to the above equation, is equivalent to solving the RLWE problem.
- **The Logical Fallacy**: This reduction is invalid. The decisional RLWE problem is hard because it requires distinguishing a challenge sample (a, b'), provided by an external party, as either pseudorandom or truly random. The adversary in the LZKSA proof, however, does not solve such a challenge. Instead, the adversary merely constructs a specific instance $(a, 0)$ that satisfies a particular algebraic relation. The adversary demonstrates no such general distinguishing capability.

b) Formal Algebraic Analysis of the Attack: We can demonstrate the binding failure through a formal algebraic analysis. This analysis shows that an attacker can construct a collision for any commitment in deterministic polynomial time.

- **Attacker's Goal and Constraints**: An attacker, having committed to a message x with opening (s, x, e), wishes to find a second valid opening (s', x', e') for the same commitment c, where $x' \neq x$. The attacker is constrained by two factors: the new message x' must be meaningful

(so $\Delta x = x - x'$ is chosen by the attacker), and the new error term e' must be small (so $\Delta e = e - e'$ must also be small). The secret value s has no such norm constraints.

- **Attack Strategy**: The attacker's task is to find a tuple $(\Delta s, \Delta x, \Delta e)$ satisfying the core equation:

$$a \cdot \Delta s + \Delta x + t \cdot \Delta e \equiv 0 \pmod{q} \tag{2}$$

The optimal strategy is to solve for the unconstrained variable, Δs.

1) The attacker first chooses a desired, non-zero message difference Δx. This could be any polynomial in R_t, allowing the forged message $x' = x - \Delta x$ to have any desired semantic meaning.
2) The attacker chooses a small, non-zero error difference Δe (e.g., $\Delta e = 1$) to ensure the resulting error term $e' = e - \Delta e$ remains small and valid.
3) The attacker then algebraically solves for Δs. By rearranging Equation 2, we get:

$$a \cdot \Delta s \equiv -\Delta x - t \cdot \Delta e \pmod{q}$$

To find Δs, the attacker computes the inverse of the public polynomial a in the ring R_q, denoted a^{-1}.

$$\Delta s \equiv a^{-1} \cdot (-\Delta x - t \cdot \Delta e) \pmod{q}$$

Ring parameters in such schemes are deliberately chosen to ensure that elements required for zero-knowledge proofs are invertible, a condition necessary for soundness [68]. These parameter choices can guarantee that all non-zero polynomials with small norms are invertible [68], [69], and thus a randomly chosen public polynomial a is invertible with overwhelming probability.

According to the principles outlined by Lyubashevsky et al. [68], by selecting appropriate parameters, the ring $\mathbb{Z}_q[X]/\langle X^n + 1\rangle$ can be decomposed via the Chinese Remainder Theorem into a Cartesian product of k smaller fields. An element in this product structure is invertible if and only if all of its k components are non-zero. The probability of a uniformly random element being invertible can therefore be expressed as:

$$P(a \text{ is not invertible}) = 1 - (1 - \frac{1}{q^{n/k}})^k < \frac{k}{q^{n/k}},$$

which is negligible [68].

- **Complexity Analysis**: The computational complexity of this attack is dominated by two steps: computing the polynomial inverse a^{-1} (e.g., via the Extended Euclidean Algorithm, complexity roughly $O(N \log^2 N)$), and performing one polynomial multiplication (complexity $O(N \log N)$ with NTT). The entire attack is therefore a low-degree polynomial-time algorithm. It is not a computationally hard problem and succeeds with non-negligible (in fact, overwhelming) probability.

This analysis demonstrates that the binding property fails because the scheme's algebraic structure allows an attacker to find a solution to the underlying homogeneous equation in polynomial time.

2) Design Flaws in the ZKP Protocols for Norm Verification: In addition to the vulnerability in the core commitment scheme, the zero-knowledge proof protocols (Protocols 3 and 4 in Section 5 in LZKSA [31]) designed for data norm verification also contain two severe design flaws (in Proposition 1 and Proposition 2 in Section 5).

a) Flaw 1: Missing Proof for the Norm of the Blinding Vector μ: This is a fundamental security vulnerability that renders the protocol unsound in LZKSA [31]:

- **Root Cause**: The security of the norm proofs relies on Proposition 1 (for the L_∞ norm) and Proposition 2 (for the L_2 norm). The logic of both propositions is conditional: if the random blinding vector μ has a bounded norm (e.g., $\|\mu\|_\infty \leq B_\infty$), then verifying an inequality allows one to infer that the secret input $\mathbf{x}$ also has a bounded norm.
- **Protocol Vulnerability**: In Protocol 3 (L_∞) and Protocol 4 (L_2), the Prover only sends a commitment "t" of μ to the Verifier. The Verifier never receives any proof that the vector μ committed to in "t" actually satisfies the required norm bound, which is the crucial premise for the entire proof.
- **Security Impact**: A malicious Prover can select a μ with an unbounded norm to cancel out the effect of a norm-violating input $\mathbf{x}$, thereby deceiving the Verifier into accepting an invalid input. This completely breaks the soundness of the norm verification mechanism.

b) Flaw 2: Impractical Positive-Input Restriction: Both of the protocol's core propositions, Proposition 1 and Proposition 2, explicitly assume that the input vector "$\mathbf{x}$" must only contain positive values. This restriction makes the protocol unsuitable for its primary stated application, Federated Learning, where model gradients inherently contain both positive and negative values. This reveals a severe practical limitation in the protocol's design.

3) Conclusion: In summary, the LZKSA scheme exhibits multiple, critical flaws at different layers:

1) Its **core commitment scheme** is **not binding** due to a flawed design and an incorrect security proof.
2) Its **ZKP protocols for norm verification** are **unsound** because they fail to prove a crucial premise of their security argument.

These flaws collectively demonstrate that the LZKSA scheme fails to meet its design objectives and cannot provide secure and reliable input validation in a practical setting.