IACR Communications in Cryptology
ISSN 3006-5496, Vol. 3, No. 2, 40 pages.

https://doi.org/10.62056/a0ivr-zn4

Paras: Actively Secure Two-Server Private Histograms

Dimitris Mouris[a,1], Lucas Piske[2], Pratik Sarkar[3], Ni Trieu[2] and Mehmet Ugurbil[b,4]

[1] Snap Inc., USA
[2] Arizona State University, USA
[3] Obsidus Labs, India
[4] Google, USA

Abstract. Private histogram computation is a fundamental building block for many data analytics tasks, enabling frequency analysis without revealing individual inputs. Existing protocols achieving robustness against malicious clients and servers typically require three servers with limited adversarial tolerance, restricting practicality.

In this work, we present Paras, the first two-server protocol for private histogram computation that achieves robustness against collusion between a malicious server and arbitrarily many malicious clients. Paras builds upon distributed point function-based approaches and introduces novel consistency checks leveraging vector oblivious linear evaluation (VOLE) to enforce both input correctness and output integrity. To realize these checks, we design two new cryptographic primitives: (1) aBV, an authenticated bit verification protocol that ensures VOLE committed shares correspond to valid bits, and (2) adIPA, an authenticated double inner product argument that enables secure consistency checks across two different VOLE sessions. These primitives may be of independent interest for other secure computation tasks.

We show that Paras is highly efficient and scalable: clients incur minimal cost independent of domain size, while servers achieve low per-client runtime, communication, and storage even at scale. For example, with 8192 clients over a domain of 128 inputs, each server requires only 14 ms runtime and 24 KB communication per client.

Keywords: Function secret sharing · private histograms · multiparty computation

1 Introduction

Privacy-preserving data analytics has become an essential requirement in today's data-driven world. Among the most fundamental statistical operations is the computation of histograms [XZX+13, BS15], which serve as the basis for numerous data analysis tasks across domains, such as healthcare [Ots79, Mun05], social sciences [FD81, CM85], and recommendation systems [ZKL+10, FPE16]. A histogram counts how many clients hold each input value, generating a frequency distribution without exposing individual data. In practice, these tasks involve inputs from well-structured sets: cybersecurity analysts track recurring incident types such as failed logins [Akb23] or denial of service [KB12]; interface designers compute heatmaps over predefined regions [LPGC17]; digital advertising platforms count clicks per ad ID [BKM+20, LPR+21, SNKG23, MMT+24]; and

E-mail: jimouris@udel.edu (Dimitris Mouris), lpiske@asu.edu (Lucas Piske), iampratiksarkar@gmail.com (Pratik Sarkar), nitrieu@asu.edu (Ni Trieu), mugurbil@nyu.edu (Mehmet Ugurbil)

[a] Work completed while at Nillion.
[b] Work mainly conducted at Nillion and completed at Hilbert's AI.

Received: 2026-01-26 Accepted: 2026-07-23

genomics researchers measure the frequency of nucleotides [PYB+06]. Yet, protecting input privacy during the histogram computation remains challenging, particularly in multi-party environments where clients may not fully trust the servers.

In response to this challenge, numerous works have emerged that focus on privacy-preserving statistics in the two/three-server, many-client model [CGB17, BBC+21, AHI+22, JKK+24, MST24, MPD+25, SMX+25]. Boneh et al. [BBC+21] introduced a protocol for private histogram computation in the two-server setting that uses distributed point functions (DPFs) [GI14]. DPFs enable clients to privately encode their inputs such that the servers can jointly compute the histogram without learning individual data points. Concretely, each client secret-shares a point function f_{α_i} built on its input x, encoded as a pair of keys $(\mathsf{key}_{i,0}, \mathsf{key}_{i,1})$ sent to servers $\mathcal{S}_0$ and $\mathcal{S}_1$, respectively. The servers evaluate these keys at different points x to obtain additive shares indicating whether each client's input matches x. By summing these shares, they compute the count for each x; repeating this process over the input domain yields the complete histogram. This approach provides security against malicious clients and hides the clients' input from a malicious server. While the DPF-based protocol is efficient and elegant, it fails to achieve correctness in the presence of malicious servers. In particular, a malicious server can arbitrarily manipulate the final histogram without being detected (and adding a MAC to detect such output manipulation is non-trivial, and we discuss this in the overview section).

Building on [BBC+21], subsequent works on private histogram and heavy hitter computation in the two-server model offer varying guarantees regarding correctness and privacy against malicious adversaries. Many approaches, such as Precio [ACD+24] and DPF-based protocols [BGI15, BGI16, GI14], provide privacy of the client inputs against malicious servers but do not ensure correctness in the presence of active attacks. Similar to the limitations of [BBC+21], a malicious server can alter the output without getting detected. Recent constructions like Mastic [MPD+25] and Doplar [DPRS23] improve security guarantees but still fall short of full robustness (i.e., privacy and correctness) in the two-server model. Notably, this lack of full robustness does not constitute any type of attack, as their functionality descriptions capture the fact that a malicious server can corrupt the final histogram output.

Other solutions, including PLASMA [MST24] and sorting-based protocols [AHI+22, JKK+24], provide full robustness against malicious clients and servers but typically rely on a three-server architecture, assuming at most one malicious server. This is a relatively weak trust model, as compromising one out of three servers is easier than compromising one out of two. There is a separate line of work [ACD+24, BBCC+25, BGG+22, BGR+24] on histograms where the client's inputs are perturbed using differentially private noise. The initial works [ACD+24, BBCC+25, BGG+22] in this line assumed semi-honest servers, and it was later improved by [BGR+24] to achieve security against a malicious server. However, the client's inputs are only hidden with differentially private noise and do not fully hide the client's input.

We provide a survey of literature in Table 1 and section 1.2. Based on these observations, we ask the following motivating question in this work:

Can we design a two-server private histogram protocol that achieves robustness against collusion between malicious clients and a malicious server?

1.1 Our Contributions

We affirmatively address the above question through the following contributions.

Paras. We introduce Paras, the *first* two-server protocol for privacy-preserving histograms that provides robustness (both in terms of privacy and correctness) against the collusion of an arbitrary number of malicious clients and a malicious server. This improves on

Table 1: Threat model comparisons with related works. Paras is the only MPC solution that guarantees correctness and privacy against malicious corruption of clients and a server in the two-server model. The protocols in the three-server setting tolerate one malicious server, assuming the servers do not collude.

Protocol	Robustness			Malicious Dishonest Majority[‡]	Security Type	Num. Servers
	Clients[*]	Server[†]	Server & Clients			
Precio [ACD+24]	●	○	○	○	DP	2-3
Hash-Prune-Invert [BBCC+25]	●	◐	◐	○	DP	2
Sparse Hist. [BGG+22]	●	○	○	○	DP	2
Mal. Sparse Hist. [BGR+24]	●	●	●	●	DP	2
DPF [BGI15, BGI16, GI14]	●	◐	○	○	MPC	2+
MPC-based [BK21]	◐	◐	○	○	MPC	3
Sorting [AHI+22, JKK+24]	●	●	●	○	MPC	3
Prio [CGB17]	●	◐	○	○	MPC	2+
Poplar [BBC+21]	●	◐	○	○	MPC	2
Doplar [DPRS23]	●	◐	○	○	MPC	2
PLASMA [MST24]	●	●	●	○	MPC	3
Mastic [MPD+25]	●	◐	○	○	MPC	2
Paras	●	●	●	●	MPC	2

[*] Vulnerable to data poisoning attacks by malicious clients or servers. The privacy of honest clients is preserved.

[†] ○ Neither privacy nor correctness against a malicious server. ◐ Privacy is preserved against a malicious server, but not correctness. ● Both privacy and correctness against a malicious server.

[‡] ○ Indicates no *malicious dishonest majority* (i.e., honest majority or dishonest majority under a semi-honest setting), while ● indicates *malicious dishonest majority*, where security holds even if all but one server is maliciously corrupted.

prior works that either need three servers [AHI+22, JKK+24, MST24], or assume semi-honest servers [BBC+21, MPD+25, DPRS23], or provide only DP-based security [BGG+22, BGR+24] to the clients. We provide a qualitative comparison in Table 1.

Consistency Checks. We build novel consistency checks that guarantee robustness against the collusion of malicious clients and a malicious server. Building upon the DPF-based techniques of [BBC+21, MST24, MPD+25], we add mechanisms that enforce input correctness and output integrity, restricting each client to a valid one-hot vector, hence enforcing the client to vote 1 for a single option. However, these techniques only work when both servers are honest. To protect against a malicious server, we introduce novel consistency checks based on Vector Oblivious Linear Evaluation (VOLE) [YWL+20]. The servers evaluate the DPF on the input domain and commit to their DPF evaluation shares using a VOLE. The servers prove that they have committed to the client-provided DPF evaluations, and the committed VOLE values are shares of a shared bit. Then, the servers compute the histogram over the DPF evaluations and the VOLE committed values, and finally generate the output if they are consistent. A malicious server cannot deviate from the protocol since the client inputs are committed by VOLE. To defend against the collusion of a malicious server and malicious clients, we incorporate the VOLE in the one-hot check to ensure that the malicious client is forced to vote correctly. We refer to the technical overview (section 3) for the details.

aBV. We introduce a novel *authenticated bit verification* protocol Π_{aBV} (modeled as $\mathcal{F}_{\mathsf{aBV}}$), which ensures that the inputs from both servers form valid VOLE correlations over binary secrets. This functionality may be of independent interest for other applications that require efficient verification of bits over secret-shared data. We use $\mathcal{F}_{\mathsf{aBV}}$ for designing the

consistency check in Paras.

adIPA. Finally, we introduce an *authenticated double inner product argument* protocol Π_{adIPA} (modeled as $\mathcal{F}_{\mathsf{adIPA}}$) that allows two parties to compute an inner product between two vectors committed via two different VOLEs. This enables inner product computation in the two-party setting where one of them can be malicious, which we leverage for a consistency check in the aBV protocol. We believe this protocol is of independent interest.

Performance Evaluations. We provide concrete runtime, communication, and storage evaluations of the Paras client and server protocols. Each Paras client runs for 81 ms and communicates 0.42 KB over a domain of size 16 inputs. Similarly, each Paras server runs for 84 ms and communicates only 0.63 KB for a domain of 1028 inputs. When running Paras with 8192 clients, each of which has a domain of size 16, each server takes 2 ms, communicates 3 KB, and stores 8 KB for each client. Scaling this up to a domain of size 128, each server's running time increases to 14 ms, communication increases to 24 KB, and finally, storage to 50 KB per client. However, as the number of inputs increases, the client costs remain almost unaffected. Additionally, an increased number of clients results in further improving the amortized server performance. This demonstrates that Paras is highly scalable and can be deployed for a realistic application in the two-server setting. We delve into more experiments in section 7.

1.2 Related Works

In this section, we cover previously published works on private histogram computation that provide functionalities similar to ours. Not all works offer the same functionality; for example, [BBC+21] and [BGG+22] provide support to histograms with super-polynomially sized domains, while other works do not, including ours. We categorize related works into the following three paradigms.

FSS-based Protocols. Function secret sharing (FSS) schemes for point functions [GI14] provide a natural solution for computing histograms with a polynomial-sized support set $\mathcal{S}$ in the 2-server setting. Each client C_i secret shares a unit vector $\boldsymbol{v}$ of length $|\mathbf{X}|$, where each component corresponds to an element of the set $\mathbf{X}$, and the component associated with the client's input α_i is set to 1. To compute the histogram, the two servers homomorphically aggregate all the vectors received from the clients and reveal the resulting vector. FSS allows the unit vectors to be secret shared in a compressed form, enhancing efficiency.

Poplar [BBC+21] leverages FSS along with a malicious-secure sketching technique to protect against malicious clients while ensuring that a malicious server cannot learn the clients' inputs. However, its approach is vulnerable to a malicious server corrupting the final histogram output. Building on this foundation, PLASMA [MST24] extended the idea to a 3-server setting, achieving stronger security against a malicious server. This is accomplished by having each pair of servers replicate FSS and verifying if their outputs match. Given the assumption that at least two servers behave honestly, at least one pair will produce the correct output. Recently, Mastic [MPD+25] addressed the weighted heavy-hitters problem in the 2-server setting. Mastic achieves the same security guarantees as Poplar's heavy-hitters protocol while incorporating the constructions from PLASMA and employs zero-knowledge proofs (ZKP) to allow for more elaborate (weighted) statistics.

DP-based Protocols. A substantial body of research has focused on histogram protocols that rely exclusively on differential privacy (DP) techniques. These approaches offer the potential for greater efficiency compared to cryptography-based methods; however, the introduction of DP noise to protect the privacy of clients' inputs often reduces the utility of the resulting computations. DP-based histogram protocols can be broadly categorized based on their underlying privacy model. In the local DP model, as seen in [BS15, CG24, WHT+15], clients independently add noise to their data before sending it to an untrusted server, which aggregates the noisy inputs to compute the histogram. The

shuffle model explored in [CSU+19, CZ22], builds on the local model by routing client messages through a secure channel that shuffles them, breaking the link between clients and their inputs. In contrast, the central DP model [KTM+21, CB22, GKKM22] assumes a trusted server that collects raw client data, computes the histogram, and adds DP noise to the result before sharing it. While the central model offers higher utility, its reliance on a trusted server departs from the objectives of this work. Replacing it with MPC would remove trust but introduce significant overhead.

MPC with DP-leakage. [BGG+22] introduces a protocol for weighted histograms in the non-colluding 2-server setting, combining DP and public-key cryptography techniques. The protocol leverages the ElGamal cryptosystem for its homomorphic properties, with an ElGamal-based OPRF as a core construction. It assumes semi-honest clients and servers and incorporates DP leakage, allowing limited information about users' inputs to be revealed, as long as it adheres to DP guarantees. However, this approach leaks more information than traditional MPC, compromising the privacy of users' inputs. [BGR+24] improves the security guarantees of the [BGG+22] to protect against an adversary who may corrupt one server and any number of clients. It analyzes the vulnerabilities of the semi-honest protocol under malicious adversaries and proposes modifications to mitigate these risks, including replacing the semi-honest OPRF with an actively secure one.

The work of [GKK+22] is in a similar setting, where multiple parties hold information about their clients and aim to compute a histogram. Their proposed solution integrates Bloom filter-based sketching with homomorphic encryption, MPC, and DP to achieve this goal. Also in the MPC & DP setting, [BBCC+25] is a generic framework for heavy-hitters and DP in the large domain setting that ensures computational differential privacy against any number of malicious clients. Unfortunately, neither of these works offers correctness against a malicious server. Nebula [SSG+25] is a system that uses an aggregator server and a randomness generation server, alongside clients responsible for generating histogram data. The servers are assumed to be non-colluding but potentially malicious, while the clients are considered semi-honest. Nebula leverages a combination of sample-and-threshold DP techniques and an OPRF-based verifiable client-side thresholding approach to ensure DP guarantees without relying on trusted third parties.

In all the aforementioned DP-based works, client inputs are protected by adding DP noise rather than being cryptographically hidden, ensuring that the output does not reveal individual inputs beyond DP guarantees. This results in a different security model from ours, where the output does not necessarily protect the input information. However, assuming at least one honest server, the inputs are otherwise perfectly hidden except for what is implied by the final output. In contrast, in DP-based approaches, even with fully honest servers, client inputs are never fully hidden and are protected only up to the chosen DP parameters. As a result, the two models are inherently incomparable and reflect different trust and leakage trade-offs. Finally, we compare with histograms protocols that can be derived from secure aggregation-based protocols.

Secure aggregation-based Protocols. The protocols in [BGL+23, CGJvdM22, LBV+23, BIK+17, BBG+20] do not naturally support histograms. In histograms, each client selects one out of many options, and this choice must remain hidden, while aggregation protocols assume clients submit encrypted numeric values that are summed. Simulating a histogram with aggregation would require running a separate aggregation instance for every domain element and forcing each client to input a non-zero value in exactly one instance. This would make the client-server communication proportional to the input domain size. Moreover, the single server aggregation protocols, like ACORN [BGL+23], EIFFEL [CGJvdM22], RoFL [LBV+23], and others [BIK+17], do not provide robustness, i.e., output correctness, against a malicious server. In the multi-server setting, ELSA [RSWP23] fails to provide robustness against a malicious server, while MARIO [NLT25] requires an honest majority among the servers to provide robustness, similar to [MST24, AHI+22, JKK+24]. Con-

sequently, these protocols neither achieve the functionality nor the security guarantees provided by Paras.

In comparison, Paras is the first protocol that works in the two-server setting and provides both robustness and privacy in the dishonest majority setting while maintaining a polylogarithmic (in input domain size) client-server communication.

2 Preliminaries

Notation. We use κ and μ for the computational and statistical security parameters, respectively. Our protocols consider a field $\mathbb{F}$ of size $\mathcal{O}(2^\mu)$. We denote the set of integers $\{0, \ldots, n-1\}$ as $[n]$. Let $\mathcal{S}_b$ represent the b^{th} server, where $b \in \{0,1\}$, and let $\mathcal{C}_i$ represent the i^{th} client, where $i \in [N]$. Let $[\![x]\!]_b$ to represent a secret share of x for $\mathcal{S}_b$. We use $\coloneqq$ for assignment, $\leftarrow \mathcal{D}$ for sampling from a distribution $\mathcal{D}$, and $=$ for checking equality.

We denote a vector of ℓ elements as $\vec{a} \coloneqq (a_1, \ldots, a_\ell)$. For two vectors $\vec{a}, \vec{b} \in \mathbb{F}^\ell$, we define their dot product $\vec{a} \cdot \vec{b} \in \mathbb{F}$ computed as $\vec{a} \cdot \vec{b} \coloneqq \sum_{i \in [\ell]} a_i \cdot b_i$, their component-wise addition $\vec{a} + \vec{b} \in \mathbb{F}^\ell$ computed as $\vec{a} + \vec{b} \coloneqq (a_1 + b_1, \ldots, a_\ell + b_\ell)$, and their component-wise multiplication $\vec{a} \odot \vec{b} \in \mathbb{F}^\ell$ computed as $\vec{a} \odot \vec{b} \coloneqq (a_1 \cdot b_1, \ldots, a_\ell \cdot b_\ell)$. For a scalar $c \in \mathbb{F}$, we denote component-wise scalar addition $\vec{a} + c \in \mathbb{F}^\ell$ computed as $\vec{a} + c \coloneqq (a_1 + c, \ldots, a_\ell + c)$, and component-wise scalar multiplication $c \cdot \vec{a} = \vec{a} \cdot c \in \mathbb{F}^\ell$ computed as $\vec{a} \cdot c \coloneqq (a_1 \cdot c, \ldots, a_\ell \cdot c)$. When working with secret-shared vectors, these same operations apply analogously.

2.1 Verifiable Distributed Point Function

A 2-*out-of*-2 *Distributed Point Function (DPF) scheme* allows a dealer to split a point function $f_{\alpha,\beta} \colon \mathcal{D} \to \mathcal{R}$ into two function shares k_0 and k_1, where $f_{\alpha,\beta}$ is 0 everywhere except at point α, where its value is β. These shares allow the parties to locally evaluate the function at any point $x \in \mathcal{D}$ in a non-interactive way, producing 2-out-of-2 secret shares of $f(x)$. Although there exist DPF schemes that generalize to more parties [GI14], our protocol only requires the 2-party variant. Standard DPFs are vulnerable to malicious dealers, who can completely compromise the result by sending corrupted DPF keys. Even worse, malicious clients can craft keys to manipulate the output without the servers' knowledge. In this work, we require the scheme to be secure against malicious dealers, which is known as a Verifiable Distributed Point Function (VDPF). VDPFs address this by ensuring that the DPF keys are well-formed using efficient hashing-based techniques. We describe the VDPF scheme [dCP22], which consists of the following three algorithms:

- $\mathsf{VDPF.Gen}(1^\kappa, \alpha, \beta) \to (\mathsf{k}_0, \mathsf{k}_1)$: Given a security parameter κ and a point function $f_{\alpha,\beta}$ that maps n-bit strings to an additive group $\mathbb{G}$, the key generation algorithm outputs two VDPF shares $(\mathsf{k}_0, \mathsf{k}_1)$.
- $\mathsf{VDPF.BatchEval}(b, \mathsf{k}_b, X) \to ([\![\vec{y}]\!]_b, \pi_b)$: This algorithm takes as input the key k_b (with $b \in \{0,1\}$) and a set of t inputs $X = \{x_0, \ldots, x_{t-1}\}$, where each x_i is an n-bit string and produces a pair $([\![\vec{y}]\!]_b, \pi_b)$. Here, $[\![\vec{y}]\!]_b$ is a vector of additive shares over $\mathbb{G}$ such that the shares from both parties sum to $f_{\alpha,\beta}(x_i)$ for each x_i. The proof π_b attests to the well-formedness of this output.
- $\mathsf{VDPF.Verify}(\pi_0, \pi_1)$: Given the pair of proofs (π_0, π_1) from the batch evaluations, the verification algorithm outputs Accept if the combined shares $[\![\vec{y}]\!]_0 + [\![\vec{y}]\!]_1$ define a valid point function (i.e., a function that is nonzero in at most one location), and Reject otherwise.

Existing VDPF constructions instantiate the underlying pseudorandom generator (PRG) using a length-doubling PRG. Throughout this work, we follow the standard approach and

instantiate this PRG using AES in PRG-expansion mode, where each invocation expands one seed into the two child seeds required during VDPF evaluation.

Initialize: Upon receiving $(\mathsf{init}, \mathsf{sid})$ from S and R, sample $\Delta \leftarrow \mathbb{F}_p$ if R is honest, otherwise, receive Δ from the adversary. Store and send Δ to R. Ignore all subsequent $(\mathsf{init}, \mathsf{sid})$ commands.

Extend: This procedure can be run multiple times. Upon receiving (sid, ℓ) from S and R:

1. If R is honest, the functionality samples $\vec{k} \leftarrow \mathbb{F}_p^\ell$. Otherwise, the functionality receives $\vec{k} \in \mathbb{F}_p^\ell$ from the adversary.
2. If S is honest, the functionality samples $\vec{x} \leftarrow \mathbb{F}_p^\ell$ and computes $\vec{m} \coloneqq \vec{k} - \Delta \cdot \vec{x} \in \mathbb{F}_p^\ell$. Otherwise, the functionality receives $\vec{x} \in \mathbb{F}_p^\ell$ and $\vec{m} \in \mathbb{F}_p^\ell$ from the adversary, and then recomputes $\vec{k} \coloneqq \vec{m} + \Delta \cdot \vec{x} \in \mathbb{F}_p^\ell$.
3. Send $(\vec{x}, \vec{m})$ to S and $\vec{k}$ to R.

Figure 1: Functionality $\mathcal{F}_{\mathsf{VOLE}}^p$ for vector oblivious linear evaluation (VOLE).

2.2 Vector Oblivious Linear Evaluation

Intuitively speaking, in this work, we define the vector oblivious linear evaluation (VOLE) functionally as a two-party functionality that uniformly samples two vectors $\vec{x}, \vec{k}, \leftarrow \mathbb{F}_p^\ell$ and an element $\Delta \leftarrow \mathbb{F}_p$, and outputs the tuples $(\vec{x}, \vec{m})$ and $(\vec{k}, \Delta)$ to the two parties S and R, respectively, where $\vec{m} = \vec{k} - \Delta \cdot \vec{x} \in \mathbb{F}_p^\ell$. Note that uncorrupted parties taking part in the protocol do not arbitrarily choose inputs to this function, as in some definitions of the VOLE functionality [YWL+20, YSWW21]. However, when a party is corrupt, they get to choose its inputs. We provide a formal definition for the VOLE functionality in Fig. 1 and use it in our protocols to generate correlated randomness for the parties to "commit to" values using an information-theoretic message authentication code (MAC) in the commit-and-prove paradigm [CD97].

We note that the two cases in Fig. 1 follow the standard ideal-functionality treatment for malicious corruption. When the sender is honest, the functionality samples uniformly random values and derives the receiver's output so that the prescribed VOLE correlation holds. Conversely, when the sender is corrupted, the adversary specifies the sender's view, and the functionality reconstructs the corresponding receiver output to preserve the same VOLE relation. This guarantees that an honest party always receives outputs that are consistent with the functionality, while allowing a corrupted party to choose its own inputs, as is standard in ideal-world functionality definitions.

2.3 Exponential ElGamal

We describe the exponential variant [CGS97] ElGamal encryption scheme [ElG84] that supports additive homomorphism over encrypted plaintexts, a feature essential to our use case. We define the following algorithms:

- $\mathsf{EG.Gen}(1^\kappa)$: Let $\mathbb{G}$ be a cyclic group of order q with generator g, where q is a prime and has bit-length κ. Sample $x \leftarrow \mathbb{Z}_q$ and compute $h \coloneqq g^x$. Output the public-private key pair $(\mathsf{pk}, \mathsf{sk})$, where $\mathsf{pk} \coloneqq (\mathbb{G}, q, g, h)$ and $\mathsf{sk} \coloneqq (\mathbb{G}, q, g, x)$.
- $\mathsf{EG.Enc}(\mathsf{pk}, m)$: Using pk from $\mathsf{EG.Gen}()$, and a plaintext $m \in \mathbb{Z}_q$, output the ciphertext $\mathsf{ct} \coloneqq (g^r, g^m \cdot h^r)$, where $r \leftarrow \mathbb{Z}_q$. We use $\mathsf{EG.Enc}(\mathsf{pk}, m, r)$ in case we want to pass a fixed randomness r.
- $\mathsf{EG.Dec}(\mathsf{sk}, \mathsf{ct})$: Let sk be a secret key from $\mathsf{EG.Gen}()$ (where $h = g^x$) and ct be a ciphertext encrypted under the corresponding public key. Output $\hat{m}$ as the discrete logarithm of $h^r \cdot g^m/(g^r)^x$ as the decrypted message.

For clarity, we occasionally abuse notation by presenting certain operations, such as ciphertext-ciphertext addition and plaintext-ciphertext multiplication, in a simplified,

white-box form. For instance, in the IPA protocol shown in Fig. 5, we write $e_1 \coloneqq \mathsf{emask}_0 \cdot \prod_{i\in[\ell]}(\mathsf{eprod}_{0,i})^{\eta_{0,i}}$ to indicate that each ciphertext $\mathsf{eprod}_{0,i}$ is multiplied with the corresponding scalar $\eta_{0,i}$. This notation reflects the underlying operation in the ElGamal scheme, where exponentiation is used to perform scalar multiplication on the encrypted plaintext. The product of these exponentiated ciphertexts corresponds to the homomorphic addition of the plaintexts.

2.4 Chaum-Pedersen Protocol

The Chaum-Pedersen protocol [CP93] allows a prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$ that they know two discrete logarithms x_1 and x_2 such that $x_1 = x_2$, without revealing any other information to $\mathcal{V}$. Specifically, $\mathcal{P}$ wants to prove knowledge of the discrete logarithms of $y_1 = g^{x_1}$ and $y_2 = h^{x_2}$, with the relation $x_1 = x_2$, while ensuring that $\mathcal{V}$ does not learn anything beyond this equivalence. Both parties share the generators g and h, and these generate cyclic groups of prime order q.

The protocol starts with $\mathcal{P}$ uniformly sampling a nonce $k \leftarrow \mathbb{Z}_q$, and sending $r_1 \coloneqq g^k$ and $r_2 \coloneqq h^k$ to $\mathcal{V}$. After receiving r_1 and r_2 from $\mathcal{P}$, the verifier sends a uniformly sampled challenge $c \leftarrow \mathbb{Z}_q$ to $\mathcal{P}$. With the challenge in hand, $\mathcal{P}$ sends $s \coloneqq k - c \cdot x_1$ to $\mathcal{V}$. Finally, $\mathcal{V}$ decides whether to accept the proof from $\mathcal{P}$ if and only if $r_1 = g^s \cdot y_1^c \wedge r_2 = h^s \cdot y_2^c$ is true. The zero-knowledge property comes from the uniform distribution of the nonce k and the CDH assumption, while the completeness of this protocol can be checked by expanding the equation used by the verifier in the last step of the protocol in the following way:

$$r_1 = g^s \cdot y_1^c = g^{k-c\cdot x_1} \cdot g^{c\cdot x_1} = g^k$$

$$r_2 = h^s \cdot y_2^c = h^{k-c\cdot x_1} \cdot h^{c\cdot x_2} = h^k$$

Special soundness comes from the distribution of the challenge c and the fact that the prover commits to the nonces before learning c by sending r_1 and r_2 to $\mathcal{V}$. Note that we are only interested in the case where the $\mathcal{V}$ is honest, so we only consider special soundness in this section.

We present the Chaum-Pedersen protocol [CP93] for proving DDH tuple as follows:

- **Input:** (g_1, g_2, X, Y).
- **Witness:** $x \in \mathbb{Z}_q$ s.t. $X = g_1^x$.
- **Statement:** Output 1 if (g_1, g_2, X, Y) is a DDH tuple of the form (g_1, g_2, g_1^x, g_2^x).
- Prove: To compute the proof, perform the following:
  - Sample $u \leftarrow \mathbb{Z}_q$ and compute $p_1 = g_1^u$ and $p_2 = g_2^u$.
  - Compute challenge $\mathsf{chall} = H(g_1, g_2, X, Y, p_1, p_2)$.
  - Output opening $z = u + \mathsf{chall} \cdot x$.

  Output proof as (p_1, p_2, z).
- **Verify:** To verify the proof, perform the following:
  - Compute challenge $\mathsf{chall} = H(g_1, g_2, X, Y, p_1, p_2)$.
  - Output 1 if $g_1^z \stackrel{?}{=} p_1 \cdot X^{\mathsf{chall}}$ and $g_2^z \stackrel{?}{=} p_2 \cdot Y^{\mathsf{chall}}$, else output 0.

The above scheme is sound and satisfies zero-knowledge assuming Discrete Log is hard. We refer to [CP93] for the formal proof.

Public Parameters
1. N clients $\mathcal{C}_i$ for $i \in [\mathsf{N}]$ and two servers $\mathcal{S}_0$ and $\mathcal{S}_1$.
2. $\mathcal{S}_0$ and $\mathcal{S}_1$ agree on the set of histogram inputs $\mathbf{X}$.

Input Phase
1. *Servers* $\mathcal{S}_0$ and $\mathcal{S}_1$ do not have any input.
2. Each *client* $\mathcal{C}_i$ for $i \in [\mathsf{N}]$ has input $\alpha_i \in \mathbf{X}$.

Corruption Phase
1. Initialize the corruption list $\widehat{\text{Ł}} \coloneqq \emptyset$ and a discard list $\text{Ł} \coloneqq \emptyset$.
2. If $\mathcal{A}$ instructs $\mathcal{F}_{\mathsf{HIST}}$ to corrupt a list $\widehat{\text{Ł}}$ of clients then mark them as corrupt. Additionally, if $\mathcal{A}$ corrupts one of the servers, then mark the server as corrupt. $\mathcal{A}$ also provides the histogram $\widehat{\mathsf{HIST}}$ computed on the inputs of the corrupt clients in $\widehat{\text{Ł}}$. This step is executed only once; $\mathcal{F}_{\mathsf{HIST}}$ ignores future requests.
3. If $\mathcal{A}$ submits a list $\text{Ł} \subseteq [\mathsf{N}]$ of clients whose inputs will be discarded, then store Ł. This step is executed only once; $\mathcal{F}_{\mathsf{HIST}}$ ignores future requests.
4. If $\mathcal{A}$ instructs $\mathcal{F}_{\mathsf{HIST}}$ to abort, then send $\perp$ to all the uncorrupted parties.

Output Phase
1. Initialize the output as $\mathsf{HIST} \coloneqq ([\![\mathsf{cnt}_1]\!], [\![\mathsf{cnt}_2]\!], \ldots, [\![\mathsf{cnt}_{|\mathbf{X}|}]\!]) \coloneqq (0, 0, \ldots, 0)$.
2. Update $[\![\mathsf{cnt}_{\alpha_i}]\!] \coloneqq [\![\mathsf{cnt}_{\alpha_i}]\!] + 1$ for each uncorrupt and undiscarded client $\mathcal{C}_i$ (i.e., for $i \in [\mathsf{N}] \setminus (\widehat{\text{Ł}} \cup \text{Ł})$).
3. Once $\mathcal{A}$ completes steps 2 and 3 in "Corruption Phase", aggregate the corrupt clients to the histogram by adding component-wise as $\mathsf{HIST} \coloneqq \mathsf{HIST} + \widehat{\mathsf{HIST}}$.
4. Send HIST to $\mathcal{A}$ and wait until $\mathcal{A}$ responds.
   a. If $\mathcal{A}$ sends $\mathsf{continue}$, then send HIST to all the uncorrupt parties.
   b. Otherwise, send $\perp$ to all the uncorrupt parties.

Figure 2: The ideal $\mathcal{F}_{\mathsf{HIST}}$ functionality for histogram.

3 Technical Overview

3.1 Private Histogram Computation

Let $\mathbf{X} \subseteq \{0,1\}^\star$ be a polynomial-sized set known a priori by all the parties involved in the protocol. There are N clients, $\mathcal{C}_i$, each holding an input $\alpha_i \in \mathbf{X}$. Our goal is to compute a histogram that counts, for every element $x \in \mathbf{X}$, how many clients have x as their input. Crucially, this must be done in a way that reveals no more information than the final histogram itself.

Threat Model. In our setting, two servers receive inputs from the clients and produce the final histogram once all inputs have been collected. As long as at least one server honestly follows the protocol and the servers do not collude, Paras guarantees both the correctness of the histogram and the privacy of the clients' inputs. Neither server learns the clients' individual inputs; only the final histogram is revealed. Moreover, Paras maintains these guarantees even if one server behaves maliciously and colludes with malicious clients. Fig. 2 presents a formal description of the ideal functionality for histogram computation.

In this work, we consider a strong security definition for histogram computation that consists of two key properties: *Completeness* and *Robustness*.

Completeness. This property ensures that if all clients and servers execute the Paras protocol honestly, the servers will produce the correct histogram as output.

Robustness. Robustness means that the final histogram remains both *correct* and *private* despite malicious behavior by clients or servers. Clients may alter or omit their inputs, and a malicious server may drop some honest inputs or block output altogether. However, the adversary cannot cause an honest server to output an arbitrary histogram, nor can they learn more than what is revealed by the final histogram itself. Paras guarantees that: 1) a set of malicious clients cannot do more than change their own inputs or refuse to send them, and 2) a malicious server cannot forge arbitrary results or break the privacy of honest clients.

Interaction Model. Motivated by scenarios with a large number of clients, minimizing communication between clients and servers is crucial [BBC+21]. We require that clients not interact with each other and that communication between clients and servers is kept to a minimum. We define three phases:

- *Setup Phase:* The servers publish data accessible to all clients.
- *Upload Phase:* Each $\mathcal{C}_i$ receives at most one message from each server and sends at most one message to each server. $\mathcal{C}_i$'s message may depend on the server's message from the Setup phase.
- *Aggregation Phase:* The servers engage in an interactive protocol to compute a histogram based on the clients' inputs. The servers do not communicate with clients during this phase.

Our Paras protocol, as well as related works from Table 1, adhere to the interaction restrictions outlined above.

Motivation for the Two-Server Model. Our goal is to guarantee both input privacy and output correctness even when one of the servers colludes with an arbitrary number of malicious clients. This security objective cannot be achieved in the corresponding single-server model. A single server necessarily receives all client messages and is solely responsible for computing the output. Consequently, a malicious server can arbitrarily modify the histogram or deviate from the protocol without an independent party capable of detecting or proving such deviations. While cryptographic commitments or zero-knowledge proofs may restrict client behavior, they do not prevent a single malicious evaluator from producing an incorrect output unless additional trust assumptions, such as trusted hardware, public verifiability, or an external verifier, are introduced.

Similarly, our communication model requires each client to participate only once during the upload phase and remain offline thereafter. Because clients do not engage in further interaction, they cannot participate in later consistency checks or dispute resolution. Therefore, all verification of the computation must be performed by independent servers using only the information uploaded during the client phase. This motivates the use of two non-colluding servers that can jointly verify each other's computation while preserving client privacy.

Compared with three-server robust protocols, Paras achieves the same guarantees while requiring less trust. Existing three-server constructions rely on an honest majority among the servers, whereas Paras operates in the dishonest majority setting, requiring only that one of the two servers remains honest and non-colluding, while tolerating collusion between the other server and an arbitrary number of malicious clients.

3.2 Paras Overview

We summarize the histogram protocol of [BBC+21] that Paras uses as a foundation, and describe how we improve upon it to provide security against malicious adversaries.

A semi-honest solution. In [BBC+21], Boneh et al. proposed a protocol for semi-honest secure private histogram computation in the 2-server setting that uses as its main building block a distributed point function DPF. In a DPF, a client $\mathcal{C}_i$ with input string $\alpha_i \in \{0,1\}^*$ secret shares the function $f_{\alpha_i} = 1$ { if $x = \alpha_i$ } else { 0 } by encoding α_i as a pair of keys $(\mathsf{key}_{i,0}, \mathsf{key}_{i,1})$. The client sends $\mathsf{key}_{i,0}$ to server $\mathcal{S}_0$ and $\mathsf{key}_{i,1}$ to $\mathcal{S}_1$. After receiving the function keys from all clients, the two servers run the histogram computation phase. To count the number of client inputs equal to x, each server $\mathcal{S}_{i\in\{0,1\}}$ starts by evaluating all their function secret shares $\mathsf{key}_{j,i}$ at position x for every $i \in [\mathsf{N}]$, where N is the number of clients. As a result of evaluating $\mathsf{key}_{i,0}$ and $\mathsf{key}_{i,1}$ at x, the servers receive 2-out-of-2 additive shares of y_i, where $y_i = 1$ if $\mathcal{C}_i$ holds $\alpha_i = x$, and $y_i = 0$ otherwise. Then, the servers non-interactively compute shares of $\mathsf{cnt}_x = \sum_{i\in[\mathsf{N}]} y_i$ and open them, where cnt_x is the desired count. To compute the full histogram on the set $\mathbf{X}$, the servers repeat this process for all different $x \in \mathbf{X}$. Even though this solution is secure against a semi-honest adversary, it leaves multiple attack vectors open for malicious ones. Our work modifies this protocol to close these attack vectors.

Protecting against malicious clients. To protect against malicious adversaries corrupting clients, we replace the DPF with a verifiable DPF scheme [dCP22] and incorporate additional steps to ensure that each client's input is a well-formed one-hot vector suitable for the histogram computation. These steps also guarantee that a malicious server cannot alter the client's input during the computation. Our client input phase begins with each $\mathcal{C}_i$ generating a pair of VDPF keys encoding the function f_{α_i}, where $\alpha_i \in \{0,1\}^*$ is $\mathcal{C}_i$'s input. At the end of this phase, the key pair is distributed to the two servers. The servers then perform a batched DPF evaluation over the input domain as $(\llbracket \vec{y_i} \rrbracket_b, \pi_{b,i}) \coloneqq \mathsf{VDPF.BatchEval}(b, \mathsf{key}_{b,i}, \mathbf{X})$. To ensure that the non-zero component of each vector $\vec{y_i}$ is exactly 1, each $\mathcal{S}_b$ locally computes the sum $\sum_{j \in \mathbf{X}} \llbracket y_{i,j} \rrbracket_b$. The servers then combine their shares to compute the sum $\sum_{j \in \mathbf{X}} y_{i,j} = \sum_{j \in \mathbf{X}} \llbracket y_{i,j} \rrbracket_0 + \sum_{j \in \mathbf{X}} \llbracket y_{i,j} \rrbracket_1$ and verify that the result equals 1, guaranteeing that the one-hot vector has a single entry set to 1.

However, the above description overlooks an important issue: a malicious server may provide an incorrect sum, or a colluding client and server may work together to bypass the verification. To address this, we incorporate a VOLE-based (section 2.2) MAC mechanism. Specifically, each $\mathcal{S}_b$ selects a random MAC key Δ_b and sends it to the client in encrypted form. We use a variant of ElGamal encryption that supports additive homomorphism to enable this. Each $\mathcal{C}_i$ samples a random mask $\mathsf{inppad}_{b,i}$ and computes the ciphertext $\mathsf{inpenc}_{b,i}$ of $\alpha_i \cdot \Delta_b + \mathsf{inppad}_{b,i}$. This ciphertext represents a MAC tag on α_i, authenticated under key Δ_b and masked with $\mathsf{inppad}_{b,i}$. After computing the ciphertexts, each client $\mathcal{C}_i$ sends the tuple $(\mathsf{key}_{b,i}, \mathsf{inpenc}_{b,i}, \mathsf{inppad}_{1-b,i})$ to server $\mathcal{S}_{b \in \{0,1\}}$. Each server $\mathcal{S}_b$ can then decrypt $\mathsf{inpenc}_{b,i}$ to obtain the MAC tag on α_i, and no longer has a need for the $\mathsf{inpenc}_{b,i}$ ciphertext. We introduce an additional step in the server initialization phase to prevent malicious servers from sending incorrect or inconsistent encryptions of their MAC keys Δ_b (see section 5.1).

In the next paragraphs, we describe how to leverage client-generated MACs to enable a robust histogram protocol. For now, if $\mathcal{C}_i$ fails any of the validation checks, its input is discarded, and its identifier i is added to the exclusion list Ł. This ensures that $\mathcal{C}_i$'s input will be ignored in all subsequent steps of the protocol and will not contribute to the final histogram.

Protecting against a malicious server. To prevent a malicious server from manipulating the output, we generate a MAC over the histogram result. However, integrating MACs into a DPF-based histogram computation is non-trivial, particularly because the DPF evaluation protocol is non-linear and inherently incompatible with standard MAC techniques. To address this challenge, we construct Paras step by step as follows.

A naive approach to this problem is to have each server evaluate every client-provided VDPF key-value pair over the entire domain $\mathbf{X}$ and then commit to the results. However, this simply shifts the problem elsewhere: we still cannot ensure that the servers are committing to the correct VDPF evaluation outputs rather than to arbitrary values chosen by a malicious adversary. In this work, we use this idea as a foundation but introduce additional steps, leveraging client-generated MACs, to ensure that both servers commit to the correct VDPF evaluations.

More specifically, we use a VOLE-based MAC as the commitment scheme, where the random VOLE correlations required to instantiate these commitments are pre-processed during the server initialization phase. This step can be performed entirely in the preprocessing stage. Once each $\mathcal{S}_{b \in \{0,1\}}$ evaluates the full domain of every VDPF key provided by each $\mathcal{C}_i$ and obtains the corresponding shares $\llbracket \vec{y_i} \rrbracket_b$, it consumes a portion of the preprocessed VOLE correlations to commit to its own shares. As a result, each $\mathcal{S}_b$ holds a tuple, where the values highlighted in yellow are known only to server $\mathcal{S}_{1-b}$ and the value

highlighted in grey is unknown to both servers:

$$\vec{k}_{b,i} \coloneqq \vec{m}_{b,i} + \Delta_b \cdot [\![\vec{y}_i]\!]_{1-b}; \quad \vec{t}_{b,i} \coloneqq \vec{m}_{b,i} + \Delta_b \cdot \vec{y}_i; \quad \vec{m}_{1-b,i}.$$

Notice that the first two vectors, $\vec{k}_{b,i}$ and $\vec{t}_{b,i}$, serve as MAC tags for the other server's DPF value $[\![\vec{y}_i]\!]_{1-b}$ and the unknown client input $\vec{y}_i$, respectively, both authenticated using the same MAC key Δ_b. Meanwhile, the last vector, $\vec{m}_{1-b,i}$, is randomly chosen during the VOLE preprocessing[1] in the $(1-b)^{th}$ instance and is known only by $\mathcal{S}_b$. It acts as a mask for the MAC tags $\vec{k}_{1-b,i}$ and $\vec{t}_{1-b,i}$, and must be revealed to the MAC verifier as part of the opening process.

Once both servers hold the MAC tags for each client's VDPF shares, they work together to perform a sequence of batched zero-knowledge proofs that guarantee the committed values satisfy the following properties without revealing them: Binary Vector, One-Hot Sum, and One-Hot Position Consistency. The first two ensure that the additive shares form a valid one-hot vector, i.e., a binary vector with exactly one entry equal to 1. The third ensures that this 1 appears at position α_i, the input specified by the non-discarded client $\mathcal{C}_i$. Together, these checks confirm that the servers are correctly committed to the function encoded by $\mathcal{C}_i$'s VDPF.

Binary Vector Property. For each client $i \in [\mathsf{N}]$, the sum of the additive shares $[\![\vec{y}_i]\!]_0 + [\![\vec{y}_i]\!]_1$ must belong to $\{0,1\}^{|\mathbf{X}|}$. To enforce this property, we introduce the functionality $\mathcal{F}_{\mathsf{aBV}}$ (see section 4). This functionality first verifies the correctness of the VOLE generated by both parties, which is used for the MAC as discussed earlier. It then checks the bitness of the MACed vector: $(x_{0,i} + x_{1,i}) \in \{0,1\},\ \forall i \in [\ell]$. That is, the sum of the shares at each index results in a bit (i.e., 0 or 1), ensuring that $\vec{x}_0$ and $\vec{x}_1$ are additive shares of a valid binary vector.

To realize $\mathcal{F}_{\mathsf{aBV}}$, we first introduce the authenticated double inner product argument ($\mathcal{F}_{\mathsf{adIPA}}$), which allows servers to check the consistency of VOLE-based MACs and validate quadratic relations without exposing the underlying inputs. $\mathcal{F}_{\mathsf{adIPA}}$ is implemented by employing randomized linear combinations of inputs together with VOLE correlations. Building on this, the aBV protocol incorporates additional verification steps to ensure that the inputs are restricted to binary values. At a high level, the protocol consists of four phases, which are presented in section 4. The servers begin by generating fresh VOLE correlations and jointly sampling random weighting vectors to mask later checks. Each server then computes commitments and challenge values that encode the quadratic relations required to verify the bit property. Following this, the servers exchange masked responses, which enable one party to validate consistency without learning the other's inputs. In the final phase, both servers invoke $\mathcal{F}_{\mathsf{adIPA}}$ to confirm VOLE-MAC correctness and verify that the randomized quadratic checks hold. The protocol succeeds if and only if the secret-shared inputs represent valid bits, catching malicious behavior with overwhelming probability.

One-Hot Sum Property. This property ensures that each client's vector $\vec{y}_i$ sums to exactly 1; that is, $\sum_{j \in \mathbf{X}} y_{i,j} = 1$, for every $i \in [\mathsf{N}] \setminus$ Ł. A natural way to verify this would be for the two servers to leverage the linearity of the commitment scheme: for each honest $\mathcal{C}_i$ and $\mathcal{S}_b$, the servers could use the MAC-tagged vector $\vec{t}_{b,i}$ to homomorphically compute and then open a commitment to $\sum_{j \in \mathbf{X}} y_{i,j}$. However, this approach requires opening $O(\mathsf{N} - |\mathcal{L}|)$ commitments and exchanging that many elements.

To reduce communication, our protocol instead verifies the condition using a compressed check via random linear combinations. Concretely, each server locally sums the components of $\vec{t}_{b,i}$ and $\vec{m}_{b,i}$ for all clients $i \notin \mathcal{L}$. These per-client sums are then packed into vectors $\vec{T}_b$

[1] Note that the index here is reversed. It corresponds to the session ID based on which server plays the receiver role with input Δ_b. For the instance where server $\mathcal{S}_b$ acts as the receiver with input Δ_b, it obtains $(\vec{k}_{b,0}, \ldots, \vec{k}_{b,\mathsf{N}-1})$ while the sender $\mathcal{S}_{1-b}$ obtains $(\vec{x}_b, \vec{m}_b) \coloneqq (\vec{x}_{b,0}, \vec{m}_{b,0}, \ldots, \vec{x}_{b,\mathsf{N}-1}, \vec{m}_{b,\mathsf{N}-1})$.

and $\vec{M}_{1-b}$, respectively. Mathematically, $\mathcal{S}_b$ computes $T_{b,i} \coloneqq \sum_{j \in \mathbf{X}} t_{b,i,j}$ and $M_{1-b,i} \coloneqq \sum_{j \in \mathbf{X}} m_{1-b,i,j}$ where $t_{b,i,j} = m_{1-b,i,j} \coloneqq 0$ for $i \in \text{Ł}$. We set $t_{b,i,j} = m_{b,i,j} = 0$, effectively excluding the contributions of malicious clients.

Next, the two servers cross-check the correctness of these sums using random linear combinations. Specifically, they exchange a hash of a random inner product over their masked values as $h_b \coloneqq \mathrm{H}(\vec{\eta_b} \cdot \vec{M}_{1-b})$, where $\vec{\eta}_b$ is a randomly sampled vector shared between the servers. Each server verifies that the same inner product can be derived from the MAC-tagged vectors adjusted by the expected server key Δ_b, by checking whether $h_b \stackrel{?}{=} \mathrm{H}(\vec{\eta_b} \cdot (\vec{T_b} - \vec{\delta_b}))$, where $\delta_{b,i} = 0$ if $i \in \text{Ł}$, and $\delta_{b,i} = \Delta_b$, otherwise. This check is sound because each MAC-tagged vector satisfies the relation $\vec{t}_{b,i} = \vec{m}_{b,i} + \Delta_b \cdot \vec{y_i}$. If the sum of the entries in $\vec{y_i}$ is 1, then the total contribution from $\Delta_b \cdot \vec{y_i}$ across all indices is exactly Δ_b, and so $\vec{T_b} - \vec{\delta_b} = \vec{M}_{1-b}$. In other words, if all honest clients submitted valid one-hot vectors (summing to 1), the computed inner products will agree, and the hashes will match. If any client's vector has an incorrect sum, the discrepancy will be detected with overwhelming probability through this randomized linear check.

One-Hot Position Consistency. The idea is to check that the position of the 1 in $\vec{y_i}$ matches the client's input α_i without learning $\vec{y_i}$. To do this, the servers rely on homomorphic encryption and MAC-authenticated shares from previous steps. This requires that every vector $\vec{y_i}$ has its non-zero component, which is equal to 1, at index α_i. Recall that α_i is $\mathcal{C}_i$'s input value. We ensure this property is true by verifying that $\sum_{j \in \mathbf{X}} j \cdot t_{b,i,j} = \mathsf{inppad}_{b,i} + \sum_{j \in \mathbf{X}} j \cdot m_{b,i,j}$ holds for each $\vec{y_i}$. The servers verify that this equation holds for each $\vec{y_i}$ by having each $\mathcal{S}_{1-b}$ send an ElGamal ciphertext of $\mathsf{inppad}_{b,i} + \sum_{j \in \mathbf{X}} j \cdot m_{b,i,j}$ and each $\mathcal{S}_b$ homomorphically compute a ciphertext for $(\mathsf{inppad}_{b,i} + \sum_{j \in \mathbf{X}} j \cdot m_{b,i,j}) - \sum_{j \in \mathbf{X}} j \cdot t_{b,i,j}$ and making sure it decrypts to 0. As in previous steps, if $\mathcal{C}_i$'s input fails this check, the servers discard it and add $\mathcal{C}_i$'s identifier i to Ł.

By proving the first two properties for each $\vec{y_i}$, we also indirectly proved that there is exactly one non-zero component in each $\vec{y_i}$ and that component is equal to 1. Additionally, by ensuring the third property is true for each $\vec{y_i}$, we proved that the component equal to 1 in each $\vec{y_i}$ is indexed by client $\mathcal{C}_i$'s input α_i. So, by ensuring these three properties hold in conjunction, we verified that the two servers committed to the secret shares for the function encoded by every honest client $\mathcal{C}_i$ as their VDPF key-pair.

Now that the two servers hold additive secret shares for every $\vec{y_i}$ and linear-homomorphic commitments to these same shares, they can straightforwardly aggregate the final histogram output by computing and opening commitments for $[\![\mathsf{cnt}_j]\!]_b = \sum_{i \in [\mathsf{N}] \setminus \text{Ł}} [\![y_{i,j}]\!]_b$, for $b \in \{0,1\}$ and $j \in \mathbf{X}$. This way, both servers can reconstruct cnt_j for every $j \in \mathbf{X}$, and output our final histogram value as $\mathsf{HIST} = (\mathsf{cnt}_1, \mathsf{cnt}_2, \ldots, \mathsf{cnt}_{\mathbf{X}})$.

Input: Server $\mathcal{S}_0$ invokes with input $(\vec{x}_1, \vec{m}_1, \vec{k}_0, \Delta_0)$. Server $\mathcal{S}_1$ invokes with input $(\vec{x}_0, \vec{m}_0, \vec{k}_1, \Delta_1)$ where $\vec{x}_0, \vec{x}_1, \vec{m}_0, \vec{m}_1, \vec{k}_0, \vec{k}_1 \in \mathbb{F}_p^\ell$ and $\Delta_0, \Delta_1 \in \mathbb{F}_p$.

Output: Output True if all three following conditions hold:

1. *VOLE 0 Verification:* $\vec{k}_0 = \vec{m}_0 + \vec{x}_0 \cdot \Delta_0$,
2. *VOLE 1 Verification:* $\vec{k}_1 = \vec{m}_1 + \vec{x}_1 \cdot \Delta_1$, and
3. *Bit Verification:* $\forall i \in [\ell], (x_{0,i} + x_{1,i}) \in \{0,1\}$.

If any of the above checks fail, then output False.

Figure 3: Authenticated Bit Verification functionality $\mathcal{F}_{\mathsf{aBV}}$ for verifying committed outputs of bits.

4 Authenticated Bit Verification (aBV)

Before presenting the Paras protocol, we introduce a new functionality called *authenticated bit verification*, denoted by $\mathcal{F}_{\mathsf{aBV}}$. This functionality involves two servers, $\mathcal{S}_0$ and $\mathcal{S}_1$, and

takes as input two VOLE correlations.[2] Specifically, $\mathcal{S}_0$ provides $\vec{k}_0$, $\vec{x}_1$, Δ_0, and $\vec{m}_1$, while $\mathcal{S}_1$ provides $\vec{k}_1$, $\vec{x}_0$, Δ_1, and $\vec{m}_0$. The goal of $\mathcal{F}_{\mathsf{aBV}}$ is to verify that the following three conditions hold. If all are satisfied, the functionality outputs True, otherwise, it returns False:

- **VOLE correctness for $\mathcal{S}_0$:** $\vec{k}_0 = \vec{m}_0 + \vec{x}_0 \cdot \Delta_0$ This verifies that $(\vec{k}_0, \vec{m}_0)$ form component-wise additive shares of a vector $\vec{x}_0$ authenticated under MAC key Δ_0.
- **VOLE correctness for $\mathcal{S}_1$:** $\vec{k}_1 = \vec{m}_1 + \vec{x}_1 \cdot \Delta_1$. Similarly, this confirms that $(\vec{k}_1, \vec{m}_1)$ form component-wise additive shares of a vector $\vec{x}_1$ authenticated under MAC key Δ_1.
- **Bitness of the MACed vector**: $(x_{0,i} + x_{1,i}) \in \{0, 1\}, \quad \forall i \in [\ell]$. That is, the sum of the shares at each index results in a bit (i.e., either 0 or 1), ensuring that $\vec{x}_0$ and $\vec{x}_1$ are additive shares of a valid binary vector.

The third check guarantees that the underlying vector being authenticated is indeed a bit vector, while the first two checks ensure that both servers' inputs are valid VOLE correlations. The formal specification of $\mathcal{F}_{\mathsf{aBV}}$ is presented in Fig. 3. Below, we describe our construction for the Π_{aBV} protocol, present its specification in Fig. 6 as an implementation of $\mathcal{F}_{\mathsf{aBV}}$, and provide the security proof in Section 6.2.

Input: $\mathcal{S}_0$ invokes with input $((\vec{x}_1, \vec{m}_1, \vec{k}_0, \Delta_0), (\vec{\delta}_0, \vec{\delta}_1, \mathsf{prod}_0))$ and
$\mathcal{S}_1$ invokes with input $((\vec{x}_0, \vec{m}_0, \vec{k}_1, \Delta_1), (\vec{\delta}_0, \vec{\delta}_1, \mathsf{prod}_1))$,
where $\vec{x}_0, \vec{x}_1, \vec{m}_0, \vec{m}_1, \vec{k}_0, \vec{k}_1, \vec{\delta}_0, \vec{\delta}_1 \in \mathbb{F}_p^\ell$, $\mathsf{prod}_0, \mathsf{prod}_1, \Delta_0, \Delta_1 \in \mathbb{F}_p$.
Output: Output True if all following conditions hold:
1. *Input Verification for Session 0:* $\vec{k}_1 = \vec{m}_1 + \vec{x}_1 \cdot \Delta_1$,
2. *Input Verification for Session 1:* $\vec{k}_0 = \vec{m}_0 + \vec{x}_0 \cdot \Delta_0$,
3. *Product Verification for Session 0:* $(\vec{\delta}_0 \odot \vec{x}_1) \cdot \vec{m}_0 = \mathsf{prod}_0$, and
4. *Product Verification for Session 1:* $(\vec{\delta}_1 \odot \vec{x}_0) \cdot \vec{m}_1 = \mathsf{prod}_1$.

If either checks fail, then output False.

Figure 4: Authenticated double inner product argument functionality $\mathcal{F}_{\mathsf{adIPA}}$ over VOLE outputs.

4.1 Authenticated Double Inner Product Argument (adIPA)

To construct Π_{aBV}, we begin by introducing a non-interactive zero-knowledge (NIZK) functionality that checks whether authenticated VOLE-based inputs satisfy inner product relations. Since our setting involves two such relations, we define the *Authenticated Double Inner Product Argument* functionality ($\mathcal{F}_{\mathsf{adIPA}}$). A formal description is provided in Fig. 4, followed by an efficient realizing protocol.

$\mathcal{F}_{\mathsf{adIPA}}$ Functionality. The functionality involves two servers, $\mathcal{S}_0$ and $\mathcal{S}_1$:

- $\mathcal{S}_0$ provides $((\vec{x}_1, \vec{m}_1, \vec{k}_0, \Delta_0), (\vec{\delta}_0, \vec{\delta}_1, \mathsf{prod}_0))$ as inputs, and
- $\mathcal{S}_1$ provides $((\vec{x}_0, \vec{m}_0, \vec{k}_1, \Delta_1), (\vec{\delta}_0, \vec{\delta}_1, \mathsf{prod}_1))$.

All vectors lie in $\mathbb{F}_p^\ell$, and all scalars are in $\mathbb{F}_p$. Upon receiving these inputs, $\mathcal{F}_{\mathsf{adIPA}}$ checks four conditions and outputs True if all are satisfied, or False otherwise:

- The first two checks verify the MACs and keys form valid VOLE correlations: $\vec{k}_1 = \vec{m}_1 + \vec{x}_1 \cdot \Delta_1$ and $\vec{k}_0 = \vec{m}_0 + \vec{x}_0 \cdot \Delta_0$.

[2] Note that the indices used in the VOLE inputs (e.g., $\vec{k}_0$, $\vec{x}_1$) do not correspond to the server identifiers. Instead, they refer to the VOLE instance or section ID—specifically, index 1 refers to the VOLE correlation in which $\mathcal{S}_1$ sampled Δ_1. For example, although $\vec{x}_1$ is input by $\mathcal{S}_0$, the subscript 1 indicates that this message was generated as part of the VOLE correlation initiated by $\mathcal{S}_1$ using Δ_1.

Inputs: $\mathcal{S}_0$ invokes with $((\vec{x}_1, \vec{m}_1, \vec{k}_0, \Delta_0), (\vec{\delta}_0, \vec{\delta}_1, \mathsf{prod}_0))$.
$\mathcal{S}_1$ invokes with $((\vec{x}_0, \vec{m}_0, \vec{k}_1, \Delta_1), (\vec{\delta}_0, \vec{\delta}_1, \mathsf{prod}_1))$,
where $\vec{x}_0, \vec{x}_1, \vec{m}_0, \vec{m}_1, \vec{k}_0, \vec{k}_1, \vec{\delta}_0, \vec{\delta}_1 \in \mathbb{F}_p^\ell$, $\Delta_0, \Delta_1, \mathsf{prod}_0, \mathsf{prod}_1 \in \mathbb{F}_p$.

Protocol Details:

1. *Key Sampling for Session 0:* ▷ Resp. Session 1
   $\mathcal{S}_0$ samples ElGamal key pair $(\mathsf{ipk}_0, \mathsf{isk}_0)$ and publishes ipk_0.
2. *Committing Inputs for Session 0:* ▷ Resp. Session 1
   $\mathcal{S}_0$ computes component wise products (for $i \in [\ell]$), encrypts them as $\mathsf{eprod}_{0,i} \coloneqq \mathsf{EG.Enc}(\mathsf{ipk}_0, \delta_{0,i} \cdot x_{1,i}, r_{0,i})$, stores the randomness as $\vec{r_0} \coloneqq \{r_{0,i}\}_{i\in[\ell]}$, and sends $\vec{\mathsf{eprod}}_0 \coloneqq \{\mathsf{eprod}_{0,i}\}_{i\in[\ell]}$ to $\mathcal{S}_1$.
3. *Pad Sampling for Session 0:* ▷ Resp. Session 1
   $\mathcal{S}_0$ and $\mathcal{S}_1$ call $\mathcal{F}_{\mathsf{VOLE}}^p$ with input $(\mathsf{extend}, \mathsf{sid}_1, 1)$ as sender and receiver to obtain (x'_1, m'_1) and k'_1, respectively, where $k'_1 = m'_1 + x'_1 \cdot \Delta_1$. $\mathcal{S}_0$ encrypts $\mathsf{emask}_0 \coloneqq \mathsf{EG.Enc}(\mathsf{ipk}_0, x'_1, r'_0)$ using the randomness r'_0 and sends it to $\mathcal{S}_1$.
4. *Input Verification for Session 0:* ▷ Resp. Session 1
   $\mathcal{S}_0$ proves that it encrypted $\vec{\delta}_0 \odot \vec{x}_1$ in $\vec{\mathsf{eprod}}_0$ by:
   a. Servers jointly sample $\vec{\eta_0} \leftarrow \mathbb{F}_p^\ell$ for input verification.
   b. $\mathcal{S}_0$ sends $u_0 \coloneqq \vec{\eta_0} \cdot (\vec{\delta}_0 \odot \vec{m}_1) + m'_1$ and $\mathsf{rnd}_0 \coloneqq \vec{\eta_0} \cdot \vec{r}_0 + r'_0$.
   c. $\mathcal{S}_1$ computes $e_1 \coloneqq \mathsf{emask}_0 \cdot \prod_{i\in[\ell]} (\mathsf{eprod}_{0,i})^{\eta_{0,i}}$ and $w_1 \coloneqq \vec{\eta_0} \cdot (\vec{\delta}_0 \odot \vec{k_1}) + k'_1$ and returns False if $e_1 \neq \mathsf{EG.Enc}(\mathsf{ipk}_0, (w_1 - u_0) \cdot \Delta_1^{-1}, \mathsf{rnd}_0)$, otherwise continues.
5. *Product Verification for Session 0:* ▷ Resp. Session 1
   a. $\mathcal{S}_1$ sends $\mathsf{eresp}_1 \coloneqq \left(\prod_{i\in[\ell]} (\mathsf{eprod}_{0,i})^{m_{0,i}}\right) \cdot \mathsf{EG.Enc}(\mathsf{ipk}_0, 0)$.
   b. If $\mathsf{EG.Dec}(\mathsf{isk}_0, \mathsf{eresp}_1^{-1} \cdot \mathsf{EG.Enc}(\mathsf{ipk}_0, \mathsf{prod}_0, 0)) \neq 0$, $\mathcal{S}_0$ outputs False, otherwise True.

Figure 5: Protocol Π_{adIPA} implementing $\mathcal{F}_{\mathsf{adIPA}}$ for authenticated double inner product argument over VOLE outputs.

- The remaining checks confirm that the claimed inner products match the MACs applied to masked VOLE inputs: $(\vec{\delta}_0 \odot \vec{x}_1) \cdot \vec{m}_0 = \mathsf{prod}_0$ and $(\vec{\delta}_1 \odot \vec{x}_0) \cdot \vec{m}_1 = \mathsf{prod}_1$.

Protocol. We now present the protocol Π_{adIPA} as shown in Fig. 5, which realizes the functionality $\mathcal{F}_{\mathsf{adIPA}}$. The protocol aims to verify two authenticated inner product relations and therefore runs two parallel sessions. For simplicity, Fig. 5 shows only the steps for session 0; session 1 follows the same structure with roles and subscripts swapped.

Steps 1 & 2. The protocol begins with each server $\mathcal{S}_b$ generating an ElGamal key pair $(\mathsf{ipk}_b, \mathsf{isk}_b)$ and publishing the public key ipk_b. For session 0, $\mathcal{S}_0$ computes the component-wise product $\vec{\delta}_0 \odot \vec{x}_1$, encrypts each element using ipk_0, and publishes the ciphertexts $\mathsf{eprod}_{0,i} \coloneqq \mathsf{EG.Enc}(\mathsf{ipk}_0, \delta_{0,i} \cdot x_{1,i}, r_{0,i})$ for $i \in [\ell]$, while storing the associated randomness in $\vec{r}_0$. These ciphertexts act as commitments to $\mathcal{S}_0$'s computation, allowing $\mathcal{S}_1$ to later verify correctness without learning $\vec{x}_1$.

An identical procedure is executed in session 1 by $\mathcal{S}_1$, with roles reversed. By the end of this step, each $\mathcal{S}_b$ holds their locally computed ciphertexts $\vec{\mathsf{eprod}}_b$, the randomness $\vec{r}_b$, and the ciphertexts $\vec{\mathsf{eprod}}_{1-b}$ from the other server.

Step 3. The servers invoke the $\mathcal{F}_{\mathsf{VOLE}}^p$ functionality with input $(\mathsf{extend}, \mathsf{sid}_1, 1)$, where sid_1 indicates that $\mathcal{S}_1$ uses its private key Δ_1. The functionality returns to $\mathcal{S}_0$ the pair (x'_1, m'_1) and to $\mathcal{S}_1$ the VOLE key k'_1. These values define a fresh random VOLE correlation used for zero-knowledge. $\mathcal{S}_0$ then encrypts the value x'_1 using fresh randomness and sends the ciphertext $\mathsf{emask}_0 \coloneqq \mathsf{EG.Enc}(\mathsf{ipk}_0, x'_1, r'_0)$ to $\mathcal{S}_1$. This masked encryption acts as a reference for $\mathcal{S}_1$ to later compare against the encrypted inner product commitment to verify consistency. The same procedure is repeated in the other session with server roles reversed.

Step 4. The goal is for $\mathcal{S}_0$ to convince $\mathcal{S}_1$ that each ciphertext in $\vec{\mathsf{eprod}}_0$ is indeed an encryption of the product $\vec{\delta}_0 \odot \vec{x}_1$, without revealing the underlying values. To enable this verification, the servers begin by jointly sampling a random vector $\vec{\eta}_0 \in \mathbb{F}_p^\ell$. This acts as a challenge, compressing the encrypted values and their corresponding MACs into a single value. Using this challenge vector, $\mathcal{S}_0$ computes:

- A compressed VOLE MAC response $u_0 \coloneqq \vec{\eta}_0 \cdot (\vec{\delta}_0 \odot \vec{m}_1) + m'_1$.
- A compressed randomness term $\mathsf{rnd}_0 \coloneqq \vec{\eta}_0 \cdot \vec{r}_0 + r'_0$.

$\mathcal{S}_0$ then sends the pair (u_0, rnd_0) to $\mathcal{S}_1$, who performs the verification as:

- A combined ciphertext $e_1 \coloneqq \mathsf{emask}_0 \cdot \prod_{i\in[\ell]} (\mathsf{eprod}_{0,i})^{\eta_{0,i}}$.
- A recomputed linear combination of the VOLE keys $w_1 \coloneqq \vec{\eta}_0 \cdot (\vec{\delta}_0 \odot \vec{k}_1) + k'_1$.

If the parties follow the protocol, the encrypted product $\prod_i \mathsf{eprod}_{0,i}^{\eta_{0,i}}$ should correspond to a ciphertext of $\vec{\eta}_0 \cdot (\vec{\delta}_0 \odot \vec{x}_1)$, and emask_0 adds in a one-time pad (the masked message x'_1) to hide this value. Thus, e_1 is a ciphertext of $x'_1 + \vec{\eta}_0 \cdot (\vec{\delta}_0 \odot \vec{x}_1)$. Additionally, when $\mathcal{S}_1$ computes $w_1 - u_0$, it recovers the same value scaled by Δ_1 as:

$$\begin{aligned} w_1 - u_0 &= \left(\vec{\eta}_0 \cdot (\vec{\delta}_0 \odot \vec{k}_1) + k'_1\right) - \left(\vec{\eta}_0 \cdot (\vec{\delta}_0 \odot \vec{m}_1) + m'_1\right) \\ &= \vec{\eta}_0 \cdot (\Delta_1 \cdot \vec{\delta}_0 \odot \vec{x}_1) + x'_1 \cdot \Delta_1. \\ &= \Delta_1 \cdot (\vec{\eta}_0 \cdot (\vec{\delta}_0 \odot \vec{x}_1) + x'_1). \end{aligned}$$

Therefore, $\mathcal{S}_1$ checks whether $e_1 \neq \mathsf{EG.Enc}(\mathsf{ipk}_0, (w_1 - u_0) \cdot \Delta_1^{-1}, \mathsf{rnd}_0)$. If this equality does not hold, $\mathcal{S}_1$ halts and outputs False. One can see that this check is sound because $\vec{\eta}_0$ is only sampled after $\mathcal{S}_0$ has published its ciphertexts; any deviation from honest behavior will be detected with overwhelming probability. This ensures that a malicious server cannot equivocate or cheat in the commitment step without being caught.

Step 5. In this step, the servers verify the correctness of the final inner products, as specified in conditions 3 and 4 of the functionality $\mathcal{F}_{\vec{}\mathsf{adIPA}}$.

First, $\mathcal{S}_1$ takes the encrypted vector eprod_0, where each ciphertext $\mathsf{eprod}_{0,i}$ should be an encryption of $\delta_{0,i} \cdot x_{1,i}$. It exponentiates each $\mathsf{eprod}_{0,i}$ by the corresponding VOLE MAC value $m_{0,i}$, and multiplies all results together:

$$\mathsf{eresp}_1 \coloneqq \Big(\prod_{i\in[\ell]} (\mathsf{eprod}_{0,i})^{m_{0,i}} \Big) \cdot \mathsf{EG.Enc}(\mathsf{ipk}_0, 0)$$

This step effectively computes an encryption of the scalar product $(\vec{\delta}_0 \odot \vec{x}_1) \cdot \vec{m}_0$. Intuitively, this ensures that $\mathcal{S}_1$ combines $\mathcal{S}_0$'s encrypted values with its own MACs to produce a joint encrypted computation result, while preserving privacy.

Then, $\mathcal{S}_1$ sends the resulting ciphertext eresp_1 to $\mathcal{S}_0$. Upon receiving this ciphertext, $\mathcal{S}_0$ verifies correctness by checking whether:

$$\mathsf{EG.Dec}(\mathsf{isk}_0, \mathsf{eresp}_1^{-1} \cdot \mathsf{EG.Enc}(\mathsf{ipk}_0, \mathsf{prod}_0; 0)) = 0$$

This step ensures that the product claimed by $\mathcal{S}_0$ matches the joint computation result, without revealing the underlying inputs. If this final equality fails, $\mathcal{S}_0$ outputs False and halts, ensuring soundness of the protocol.

4.2 The aBV Protocol

We now present our protocol Π_{aBV} that realizes the functionality $\mathcal{F}_{\mathsf{aBV}}$ as defined in Fig. 6. To demonstrate correctness and security, we walk through each step of the protocol and explain how they collectively enforce the four verification conditions specified in $\mathcal{F}_{\mathsf{aBV}}$, all while ensuring security against malicious adversaries. As before, we focus on session 0 for clarity, noting that session 1 proceeds analogously.

Steps 1 & 2. The protocol begins with the two servers invoking $\mathcal{F}^p_{\mathsf{VOLE}}$. $\mathcal{S}_1$ acts as the sender and $\mathcal{S}_0$ as the receiver, receiving the correlated outputs (x'_0, m'_0) and k'_0, respectively.

Input: Server $\mathcal{S}_0$ invokes with input $(\vec{x}_1, \vec{m}_1, \vec{k}_0, \Delta_0)$. Server $\mathcal{S}_1$ invokes with input $(\vec{x}_0, \vec{m}_0, \vec{k}_1, \Delta_1)$ where $\vec{x_0}, \vec{x_1}, \vec{m_0}, \vec{m_1}, \vec{k_0}, \vec{k_1} \in \mathbb{F}_p^\ell$ and $\Delta_0, \Delta_1 \in \mathbb{F}_p$.

Protocol Details:

1. *Pad Sampling for Session 0:* ▷ Resp. Session 1
   $\mathcal{S}_1$ and $\mathcal{S}_0$ call $\mathcal{F}_{\mathsf{VOLE}}^p$ with $(\mathsf{extend}, \mathsf{sid}_0, 1)$ as sender and receiver to obtain (x'_0, m'_0) and k'_0, respectively.
2. Servers jointly sample linear combiners $\vec{\delta_0}, \vec{\delta_1} \in \mathbb{F}_p^\ell$ for batch VOLE verification in sessions 0 and 1.
3. *Batch Verification for Session 0:* ▷ Resp. Session 1
   a. $\mathcal{S}_0$ locally computes $\vec{t_0} \coloneqq \vec{k}_0 + \vec{x}_1 \cdot \Delta_0$ and the challenge $\mathsf{chall}_0 \coloneqq \vec{\delta}_0 \cdot (\vec{t_0} \odot (\vec{t_0} - \Delta_0)) + k'_0$.
   b. $\mathcal{S}_1$ computes the 0^{th} degree and 1^{st} degree terms in Δ_0, respectively, as $\mathsf{degzero}_1 \coloneqq \vec{\delta}_0 \cdot (\vec{m}_0 \odot \vec{m}_0) + m'_0$, and $\mathsf{degone}_1 \coloneqq \vec{\delta}_0 \cdot (2\vec{m}_0 \odot \vec{x}_0 - \vec{m}_0) + x'_0$ and sends them to $\mathcal{S}_0$.
   c. $\mathcal{S}_0$ computes $\mathsf{prod}_0 \coloneqq (\mathsf{chall}_0 - (\mathsf{degzero}_1 + \mathsf{degone}_1 \cdot \Delta_0))/2\Delta_0$.
4. *Pairwise Product Check for both Sessions:*
   Both servers verify that $(\vec{\delta}_0 \odot \vec{x}_1) \cdot \vec{m}_0 = \mathsf{prod}_0$ and $(\vec{\delta}_1 \odot \vec{x}_0) \cdot \vec{m}_1 = \mathsf{prod}_1$ using $\mathcal{F}_{\mathsf{adIPA}}$. $\mathcal{S}_0$ returns $\mathcal{F}_{\mathsf{adIPA}}((\vec{x}_1, \vec{m}_1, \vec{k}_0, \Delta_0), (\vec{\delta}_0, \vec{\delta}_1, \mathsf{prod}_0))$ and $\mathcal{S}_1$ returns $\mathcal{F}_{\mathsf{adIPA}}((\vec{x}_0, \vec{m}_0, \vec{k}_1, \Delta_1), (\vec{\delta}_0, \vec{\delta}_1, \mathsf{prod}_1))$.

Figure 6: Our Π_{aBV} protocol implementing $\mathcal{F}_{\mathsf{aBV}}$ for verifying that committed VOLE outputs are to bits.

This correlation will later be used to mask the value (Step 3) that $\mathcal{S}_0$ challenges and $\mathcal{S}_1$ responds to, ensuring that the response is bound to a specific input without revealing it. Next, the servers jointly sample two vectors $\vec{\delta_0}, \vec{\delta_1} \in \mathbb{F}_p^\ell$ uniformly at random. These vectors will later serve as weights for randomized linear checks, which hide the structure of the inputs while enabling correctness verification.

Step 3. $\mathcal{S}_0$ begins by computing a "final" MAC key for verification as $\vec{t_0} \coloneqq \vec{k}_0 + \vec{x}_1 \cdot \Delta_0$. This key represents the sum of the committed key and the MAC of the opposing server's message, which must be consistent under the MAC check. $\mathcal{S}_0$ then uses this to compute a challenge $\mathsf{chall}_0 \coloneqq \vec{\delta_0} \cdot (\vec{t_0} \odot (\vec{t_0} - \Delta_0)) + k'_0$. This challenge encodes the squared terms necessary for checking whether the underlying input is a valid bit (i.e., in $\{0, 1\}$), while hiding the actual values via randomization. In response, $\mathcal{S}_1$ computes and sends two masked quantities:

$$\mathsf{degzero}_1 \coloneqq \vec{\delta_0} \cdot (\vec{m}_0 \odot \vec{m}_0) + m'_0 \quad \text{and} \quad \mathsf{degone}_1 \coloneqq \vec{\delta_0} \cdot (2\vec{m}_0 \odot \vec{x}_0 - \vec{m}_0) + x'_0.$$

These responses are used to indirectly verify that the squared value of the inputs matches the input itself (i.e., the bit-check). The terms are structured such that if x_0 is not in $\{0, 1\}$, this structure breaks, causing the check to fail.

Upon receiving these, $\mathcal{S}_0$ computes the expected result of the inner product check as $\mathsf{prod}_0 \coloneqq (\mathsf{chall}_0 - (\mathsf{degzero}_1 + \mathsf{degone}_1 \cdot \Delta_0))/(2\Delta_0)$.

Step 4. Finally, the two servers call the $\mathcal{F}_{\mathsf{adIPA}}$ functionality. Each server $\mathcal{S}_b$ submits $((\vec{x_{1-b}}, \vec{m_{1-b}}, \vec{k_b}, \Delta_b), (\vec{\delta_0}, \vec{\delta_1}, \mathsf{prod}_b))$ and receives the output from $\mathcal{F}_{\mathsf{adIPA}}$. $\mathcal{F}_{\mathsf{adIPA}}$ verifies two main conditions for each server $b \in \{0, 1\}$: $\vec{m}_b + \vec{x}_b \cdot \Delta_b = \vec{k}_b$ and $(\vec{\delta}_{1-b} \odot \vec{x}_b) \cdot \vec{m}_{1-b} = \mathsf{prod}_{1-b}$. The first condition checks that the MACs are correctly formed, satisfying the integrity requirements (corresponding to checks (1) and (2) in $\mathcal{F}_{\mathsf{aBV}}$). The second condition confirms that the input values are bits, either 0 or 1, by evaluating the masked quadratic expression (corresponding to check (3) in $\mathcal{F}_{\mathsf{aBV}}$).

To understand the core idea, let's unpack the second condition more carefully. Begin by expanding the definition of prod_0:

$$\begin{aligned} \mathsf{prod}_0 &= \frac{1}{2\Delta_0} \cdot \vec{\delta_0} \cdot \left(\vec{t_0} \odot (\vec{t_0} - \Delta_0) - \vec{m_0} \odot \vec{m_0} - \Delta_0 \cdot (2\vec{m_0} \odot \vec{x_0} - \vec{m_0})\right) \\ &= \frac{1}{2\Delta_0} \cdot \sum_{i \in [\ell]} \delta_{0,i} \cdot \left(t_{0,i} \cdot (t_{0,i} - \Delta_{0,i}) - m_{0,i}^2 - \Delta_0 \cdot (2m_{0,i} \cdot x_{0,i} - m_{0,i})\right) \end{aligned} \tag{1}$$

Additionally, we have: $\vec{t_b} = \vec{k_b} + \vec{x}_{1-b} \cdot \Delta_b = \vec{m}_b + (\vec{x}_b + \vec{x}_{1-b}) \cdot \Delta_b$. Now, we analyze two cases:

- If $x_{0,i}+x_{1,i}=0$, then $t_{0,i}=m_{0,i}$. Substituting this into the term $\big(t_{0,i}\cdot(t_{0,i}-\Delta_{0,i})-m_{0,i}^2-\Delta_0\cdot(2m_{0,i}\cdot x_{0,i}-m_{0,i})\big)$ in Eq. (1), simplifies the expression to $2\Delta_0\cdot m_{0,i}\cdot x_{1,i}$.
- If $x_{0,i}+x_{1,i}=1$, then $t_{0,i}=m_{0,i}+\Delta_0$, which leads to the same simplification.

So in both valid cases, we find that $(\vec{\delta_b}\odot\vec{x}_{1-b})\cdot\vec{m_b}=\mathsf{prod}_b$ for $b\in\{0,1\}$. Hence, the last condition in $\mathcal{F}_{\mathsf{adIPA}}$ will pass if and only if each pair of messages sums to a bit, enforcing the binary nature of inputs. From a security standpoint, if there exists any index i such that $x_{0,i}+x_{1,i}\notin\{0,1\}$ for any i, then the randomized inner product over $\vec{\delta_b}$ will differ from the expected result, and the verification will fail with overwhelming probability. This ensures that even a malicious party cannot trick the protocol into accepting invalid input.

5 Paras Histogram Protocol

Paras comprises three main phases: Server Initialization, Client Input Phase, and Histogram Computation, which align with the interaction pattern described in Section 3. We formally present the Paras protocol in Figs. 7 and 8.

5.1 Server Initialization

In this phase, $\mathcal{S}_0$ and $\mathcal{S}_1$ generate correlated randomness and initialize internal variables that will be used throughout the protocol. They also publish material that enables the detection of malicious behavior during the online phase.

Step 1. In this step, the two servers execute two instances of the $\mathcal{F}^p_{\mathsf{VOLE}}$ functionality, one where $\mathcal{S}_0$ plays the sender and $\mathcal{S}_1$ the receiver, and another where the roles are reversed. From the instance where $\mathcal{S}_b$ plays the receiver with input Δ_b, it obtains $(\vec{k}_{b,0},\ldots,\vec{k}_{b,\mathsf{N}-1})$ while the sender $\mathcal{S}_{1-b}$ obtains $(\vec{x}_b,\vec{m}_b):=(\vec{x}_{b,0},\vec{m}_{b,0},\ldots,\vec{x}_{b,\mathsf{N}-1},\vec{m}_{b,\mathsf{N}-1})$ where $\vec{k}_{b,i}=\vec{m}_{b,i}+\Delta_b\cdot\vec{x}_{b,i}$ for every $i\in[\mathsf{N}]$.

These two sets of VOLE correlations are used during the online phase by the servers towards the instantiation of MAC-based commitments. Since our protocol requires both servers to commit and verify commitments, we need two VOLE instances. The element Δ_0 provided to $\mathcal{S}_0$ serves as its MAC key when acting as a verifier, and similarly, Δ_1 serves as the MAC key for $\mathcal{S}_1$ in its role as verifier.

Step 2. To protect the value of Δ_b, each server $\mathcal{S}_{b\in\{0,1\}}$ generates its own ElGamal key pair $(\mathsf{pk}_b,\mathsf{sk}_b)$ and publishes a ciphertext $\mathsf{EG.Enc}(\mathsf{pk}_b,\Delta_b)$, encrypting Δ_b under pk_b, which is made available to all parties participating in the protocol. These encrypted MAC keys enable clients to authenticate their inputs during the online phase without requiring interaction with the servers. This mechanism is discussed in detail later in this section as part of the online phase description. To ensure the two servers publish ciphertexts that encrypt the correct MAC keys, they run an interactive sub-protocol verifying their correctness in the next step.

Step 3. At a high level, we leverage VOLE, the additive homomorphism of ElGamal encryption, and a lightweight NIZK to verify the consistency of Δ_b in the first two steps described above. More concretely, for $b\in\{0,1\}$, this sub-protocol has servers $\mathcal{S}_{1-b}$ and $\mathcal{S}_b$ call the $\mathcal{F}^p_{\mathsf{VOLE}}$ functionality as sender and receiver, respectively, requesting vectors of length 1. As a result, $\mathcal{S}_b$ receives $k'_b\in\mathbb{Z}_p$, and $\mathcal{S}_{1-b}$ receives $x'_b,m'_b\in\mathbb{Z}_p$. If both parties are honest, we have $k'_b=m'_b+x'_b\cdot\Delta_b$. Additionally, $\mathcal{S}_b$ commits to k'_b by sending the ciphertext $\mathsf{EG.Enc}(\mathsf{pk}_b,k'_b)$ to $\mathcal{S}_{1-b}$; this commitment is used in Step (3d). Next, $\mathcal{S}_{1-b}$ samples $m''_b\leftarrow\mathbb{F}$ and homomorphically computes $e''_b:=\mathsf{EG.Enc}(\mathsf{pk}_b,m''_b)\cdot\mathsf{enc}_b^{x'_b}$ then sends the result to $\mathcal{S}_b$. If computed correctly, this yields $e''_b:=\mathsf{EG.Enc}(\mathsf{pk}_b,m''_b+x'_b\cdot\Delta_b)$, meaning e''_b is a ciphertext of the MAC value $m''_b+x'_b\cdot\Delta_b$. Here, the random value m''_b also serves

as a mask for m'_b. When $\mathcal{S}_{1-b}$ sends $m'_b - m''_b$ to $\mathcal{S}_b$, it effectively proves that the same x'_b was used both in the VOLE session and in the e''_b.

After receiving $(e''_b, m'_b - m''_b)$, $\mathcal{S}_b$ computes $E'_b \coloneqq \mathsf{EG.Enc}(\mathsf{pk}_b, k'_b) \cdot (e''_b)^{-1} \cdot \mathsf{EG.Enc}(\mathsf{pk}_b, m'_b - m''_b)^{-1}$. If all parties are honest, the resulting ciphertext encrypts 0, enabling a consistency check on the actions of $\mathcal{S}_{1-b}$. If this check passes, both parties proceed to the final step: $\mathcal{S}_b$ generates a NIZK proof of correct decryption, $\pi'_b \coloneqq \mathsf{NIZK}_{\mathsf{EG.Dec}}.\mathsf{Prove}(E'_b, \mathsf{sk}_b)$ and sends it to $\mathcal{S}_{1-b}$. $\mathcal{S}_{1-b}$ then verifies the proof and aborts if the verification fails.

To enable this verification, $\mathcal{S}_{1-b}$ must independently compute the same ciphertext E'_b. We denote its local computation as $E''_b \coloneqq \widehat{e_b} \cdot (e''_b)^{-1} \cdot \mathsf{EG.Enc}(\mathsf{pk}_b, m'_b - m''_b, 0)^{-1}$ where $\widehat{e_b}$ refers to the commitment $\mathsf{EG.Enc}(\mathsf{pk}_b, k'_b)$ received earlier in Step (3a). To ensure that both parties have an identical ciphertext, we require that the encryption randomness used in $\mathsf{EG.Enc}(\mathsf{pk}_b, m'_b - m''_b)$ must be known to both. For simplicity, we fix this randomness to zero and explicitly denote it in our protocol as $\mathsf{EG.Enc}(\mathsf{pk}_b, m'_b - m''_b, 0)$.

Finally, the servers initialize a set $Ł \coloneqq \emptyset$, which is jointly maintained to track the identities of all clients detected as cheating during the protocol execution.

5.2 Client Input Phase

First, each $\mathcal{C}_i$ begins with generating a VDPF key pair $(\mathsf{key}_{0,i}, \mathsf{key}_{1,i})$ corresponding to the point function $f_{\alpha_i,1}(x)$ (recall 1 if $x = \alpha_i$, otherwise 0). The VDPF allows the servers to verify that the DPF was generated correctly – i.e., it encodes a function that is nonzero at most at one location – and to jointly compute the histogram. However, to prevent a malicious server from modifying the evaluated DPF output, each client must attach a MAC to their input using the server's MAC. Specifically, $\mathcal{C}_i$ samples $\mathsf{inppad}_{0,i}, \mathsf{inppad}_{1,i} \leftarrow \mathbb{F}$, and homomorphically computes two ElGamal ciphertexts $(\mathsf{inpenc}_{0,i}, \mathsf{inpenc}_{1,i})$ as follows:

$$\mathsf{inpenc}_{0,i} \coloneqq \mathsf{enc}_0^{\alpha_i} \cdot \mathsf{EG.Enc}(\mathsf{pk}_0, \mathsf{inppad}_{0,i})$$
$$\mathsf{inpenc}_{1,i} \coloneqq \mathsf{enc}_1^{\alpha_i} \cdot \mathsf{EG.Enc}(\mathsf{pk}_1, \mathsf{inppad}_{1,i})$$

Here, enc_0 and enc_1 are ciphertexts published by the servers, encrypting their respective MAC keys Δ_0 and Δ_1. As a result, the ciphertexts $\mathsf{inpenc}_{0,i}$ and $\mathsf{inpenc}_{1,i}$ computed by $\mathcal{C}_i$ respectively encrypt the values $\alpha_i \cdot \Delta_0 + \mathsf{inppad}_{0,i}$ and $\alpha_i \cdot \Delta_1 + \mathsf{inppad}_{1,i}$. These values can be interpreted as MAC tags on the same message α_i, each under a different key held by a different server. Lastly, each $\mathcal{C}_i$ completes the input phase by sending $(\mathsf{key}_{b,i}, \mathsf{inpenc}_{b,i}, \mathsf{inppad}_{1-b,i})$ to $\mathcal{S}_{b\in\{0,1\}}$.

5.3 Histogram Computation

To ensure the correctness and robustness of the histogram computation, the servers perform a series of checks on the secret-shared vectors committed by clients and exchanged between themselves. These checks prevent malicious DPF encodings, verify the servers' commitments to their shares during histogram computation, and guard against malicious server behavior.

Step 1: Preventing Malicious DPF Encodings. In this phase, the two servers verify that all VDPF keys from the Client Input phase are as specified by the protocol. Specifically, they check that: (i) each VDPF key-pair encodes a vector with *at most* one non-zero element, and (ii) if such an element exists, its value is exactly one. This phase assumes semi-honest servers and malicious clients; any malicious behavior by the servers is addressed in later phases.

Property (i) follows directly from the design of the VDPF scheme. Each $\mathcal{S}_b$ computes $(\llbracket \vec{y_i} \rrbracket_b, \pi_{b,i}) \coloneqq \mathsf{VDPF.BatchEval}(b, \mathsf{key}_{b,i}, \mathbf{X})$, exchanges their proofs $\pi_{b,i}$, and then executes $\mathsf{VDPF.Verify}(\pi_{b,i}, \pi_{1-b,i})$, for every $\mathcal{C}_i$. By the definition of the VDPF scheme, this verification succeeds if and only if the vector $\vec{y_i}$ contains at most one non-zero element. Because

the entire domain $\mathbf{X}$ is evaluated, the verification guarantees that the VDPF key-pair encodes a vector with at most one non-zero component. However, VDPF alone does not ensure the second property; i.e., that the non-zero element's value must be exactly one.

To verify that each $\mathcal{C}_i$ encoded the value 1 as the non-zero element in their VDPF keys, the servers exchange the sums $\sum_{j\in\mathbf{X}}[\![y_{i,j}]\!]_b$ and confirm whether this sum equals 1. Given that the first check (ensuring at most one non-zero component) has passed, this sum equals 1 if and only if the non-zero value in $\vec{y}_i$ is indeed 1. This verification does not leak additional information because the sum hides the position of the non-zero element.

By the end of this protocol phase, any $\mathcal{C}_i$ who failed to correctly encode their VDPF key pair is identified. For each such client, their input is discarded, and their identifier i is added to the rejection set $\mathcal{L}$. As a result, all inputs from clients in $\mathcal{L}$ are excluded from the final histogram output.

Step 2: Server Commitment. In this phase, each $\mathcal{S}_b$ commits to the additive, component-wise secret shares $[\![\vec{y}_i]\!]_b$ resulting from its full-domain evaluation of the VDPF keys $\mathsf{key}_{b,i}$ received from clients. Specifically, the goal is for each $\mathcal{S}_b$ to hold the following for every client $i \in [\mathsf{N}]$ and $b \in \{0,1\}$: $\vec{k}_{b,i} = \vec{m}_{b,i} + \Delta_b \cdot [\![\vec{y}_i]\!]_{1-b}$; and $\vec{t}_{b,i} = \vec{m}_{b,i} + \Delta_b \cdot \vec{y}_i$. It is easy to see that $\vec{k}_{b,i}$ and $\vec{t}_{b,i}$ serve as MAC tags for each other's VDPF evaluation $[\![\vec{y}_i]\!]_{1-b}$ and the value $\vec{y}_i$, respectively, both authenticated under the same key Δ_b. The term $\vec{m}_{b,i}$ acts as a random mask, hiding the underlying message from the commitment verifier.

Before detailing the steps, we note that this phase consumes the pre-processed VOLE correlations $(\vec{k}_{b,0},\ldots,\vec{k}_{b,\mathsf{N}-1})$ and $(\vec{x}_{b,0},\vec{m}_{b,0},\ldots,\vec{x}_{b,\mathsf{N}-1},\vec{m}_{b,\mathsf{N}-1})$ generated during the Server Initialization phase to instantiate a MAC-based commitment scheme, for $b \in \{0,1\}$. Recall that each $\vec{k}_{b,i}$ satisfies $\vec{k}_{b,i} = \vec{m}_{b,i} + \Delta_b \cdot \vec{x}_{b,i}$.

To achieve the commitment described earlier, the two servers begin this phase by exchanging vectors $\vec{w}_{0,i}$ and $\vec{w_{1,i}}$ for every $i \in [\mathsf{N}]$. For $b \in \{0,1\}$, $\mathcal{S}_{1-b}$ sends $\vec{w}_{b,i} \coloneqq [\![\vec{y}_i]\!]_{1-b} - \vec{x}_{b,i}$ to $\mathcal{S}_b$. Upon receiving $\vec{w}_{b,i}$, each $\mathcal{S}_b$ updates its corresponding $\vec{k}_{b,i}$ as $\vec{k}_{b,i} \coloneqq \vec{k}_{b,i} + \Delta_b \cdot \vec{w}_{b,i}$. Here, we have:

$$\begin{aligned}\vec{k}_{b,i} &\coloneqq \vec{k}_{b,i} + \Delta_b \cdot \vec{w}_{b,i} = (\vec{m}_{b,i} + \vec{x}_{b,i} \cdot \Delta_b) + \Delta_b \cdot \vec{w}_{b,i} \\ &= (\vec{m}_{b,i} + \vec{x}_{b,i} \cdot \Delta_b) + \Delta_b \cdot ([\![\vec{y}_i]\!]_{1-b} - \vec{x}_{b,i}) = \vec{m}_{b,i} + \Delta_b \cdot [\![\vec{y}_i]\!]_{1-b}.\end{aligned}$$

This final expression confirms that $\vec{k}_{b,i}$ now represents a MAC tag on the message $[\![\vec{y}_i]\!]_{1-b}$ under key Δ_b, masked by the random vector $\vec{m}_{b,i}$. Each $\mathcal{S}_b$ then completes this phase by computing $\vec{t}_{b,i} \coloneqq \vec{k}_{b,i} + \Delta_b \cdot [\![\vec{y}_i]\!]_b$. We have $\vec{t}_{b,i} \coloneqq \vec{k}_{b,i} + \Delta_b \cdot [\![\vec{y}_i]\!]_b = (\vec{m}_{b,i} + \Delta_b \cdot [\![\vec{y}_i]\!]_{1-b}) + \Delta_b \cdot [\![\vec{y}_i]\!]_b = \vec{m}_{b,i} + \Delta_b \cdot \vec{y}_i$.

In this step, the only way a malicious $\mathcal{S}_{1-b}$ can deviate from the protocol is by sending an incorrect value $\vec{w}_{b,i}$ to $\mathcal{S}_b$. This effectively means that $\mathcal{S}_{1-b}$ is committing to an invalid value; i.e., one that violates the expected relation $\vec{k}_{b,i} = \vec{m}_{b,i} + \Delta_b \cdot [\![\vec{y}_i]\!]_{1-b}$. To address this, the next phase leverages the linear homomorphic property of the commitment scheme, enabling both servers to verify that the committed values are consistent with the protocol specifications.

Step 3: Preventing a Malicious Server. To ensure that the shares $[\![\vec{y}_i]\!]_0$ and $[\![\vec{y}_i]\!]_1$ committed by the servers in the previous phase are correct (i.e., that they follow the protocol's specifications), the servers perform a joint verification. Specifically, for each $\mathcal{C}_i$ not in the set $\mathcal{L}$ of flagged (malicious) clients, the servers check that the following three properties hold: the binary vector property, the one-hot sum property, and the one-hot position consistency property.

The binary vector property guarantees that each additive share pair $[\![\vec{y}_i]\!]_0, [\![\vec{y}_i]\!]_1$ corresponds to shares of a binary vector, while the one-hot sum property ensures these shares sum to exactly one. Together, these two properties confirm that $[\![\vec{y}_i]\!]_0, [\![\vec{y}_i]\!]_1$ represent

shares of a vector with exactly one component equal to 1 and all others 0. The final property further ensures that this unique 1 is located at position α_i, the input chosen by the non-discarded $\mathcal{C}_i$. Collectively, these three properties ensure that both servers are correctly committed to shares of the function encoded by $\mathcal{C}_i$ as its VDPF.

Step 3a: Binary Vector Property. This phase begins by verifying the binary vector property for all VDPF keys from clients not in the discard set $\mathcal{L}$. The property requires that, for each client $i \in [\mathsf{N}]$, the sum of the additive shares $[\![\vec{y_i}]\!]_0 + [\![\vec{y_i}]\!]_1$ must lie in $\{0,1\}^{|\mathbf{X}|}$. In other words, the two servers must have committed to additive shares of a binary vector.

The verification of this property is carried out by having both servers invoke the functionality $\mathcal{F}_{\mathsf{aBV}}$ (see Fig. 3). Each $\mathcal{S}_b$ inputs the tuple $\big(\{\vec{k}_{b,i}\}_{i\in[\mathsf{N}]}, \{[\![\vec{y_i}]\!]_{1-b}\}_{i\in[\mathsf{N}]}, \Delta_b, \{\vec{m}_{1-b,i}\}_{i\in[\mathsf{N}]}\big)$ for $b \in \{0,1\}$. The functionality $\mathcal{F}_{\mathsf{aBV}}$ uses the MAC-based commitments from the previous phase to verify that the sum of the shares $[\![\vec{y_i}]\!]_0 + [\![\vec{y_i}]\!]_1$ is in $\{0,1\}^{|\mathbf{X}|}$ for each i. It returns True if the condition is satisfied and False, otherwise, in which case the servers abort.

Step 3b: One-Hot Sum Property. This property ensures that for each $\mathcal{C}_i$, whose shares $[\![\vec{y_i}]\!]_0$ and $[\![\vec{y_i}]\!]_1$ were committed to by the servers in the previous phase, the following holds: $\sum_{j\in\mathbf{X}}([\![\vec{y}_{i,j}]\!]_0 + [\![\vec{y}_{i,j}]\!]_1) = 1$. In other words, the sum of all components of the secret-shared vector $\vec{y}_i$ must equal 1 for every $i \in [\mathsf{N}]$. While Step (1c) verifies this condition, it assumes the servers are semi-honest and provide the correct $\sum_{j\in\mathbf{X}}[\![y_{i,j}]\!]_b$. This step removes that assumption and verifies the condition without trusting the servers. To achieve this, we use the MAC tag $\vec{t}_{b,i}$ of $\vec{y_i}$ masked using the random vector $\vec{m}_{b,i}$, computed in Step (2a). Note that the correctness of $\vec{t}_{b,i}$ is implicitly verified in Step (3a) through $\mathcal{F}_{\mathsf{aBV}}$.

Each $\mathcal{S}_b$ begins begins by computing the component-wise sum of the vectors $\vec{t}_{b,i}$ and $\vec{m}_{b,i}$ over the indices of honest clients, as follows:

$$\vec{T}_b \coloneqq \big(\sum_{j\in\mathbf{X}} t_{b,0,j}, \ldots, \sum_{j\in\mathbf{X}} t_{b,\mathsf{N}-1,j}\big) \quad \text{and} \quad \vec{M}_{1-b} \coloneqq \big(\sum_{j\in\mathbf{X}} m_{1-b,0,j}, \ldots, \sum_{j\in\mathbf{X}} m_{1-b,\mathsf{N}-1,j}\big)$$

where for every $i \in Ł$, we set $t_{b,i,j} = m_{b,i,j} = 0$, effectively excluding the contributions of malicious clients.

Next, both servers sample random vectors $\vec{\eta}_0, \vec{\eta}_1 \leftarrow \mathbb{F}^{\mathsf{N}}$, and then exchange the values $h_0 \coloneqq \mathrm{H}(\vec{\eta_0} \cdot \vec{M_0})$ and $h_1 \coloneqq \mathrm{H}(\vec{\eta_1} \cdot \vec{M_1})$. Each $\mathcal{S}_b$ then verifies the received hash and aborts if $h_b \neq \mathrm{H}(\vec{\eta_b} \cdot (\vec{T_b} - \vec{\delta_b}))$, where $\delta_{b,i} = \Delta_b$ for each honest $\mathcal{C}_i$ and $\delta_{b,i} = 0$ if $i \in Ł$, i.e., clients identified as malicious. To understand how this interaction verifies the second property, consider a $\mathcal{C}_i$ whose input has not been discarded. Suppose that indeed $\sum_{j\in\mathbf{X}}([\![\vec{y_{i,j}}]\!]_0 + [\![\vec{y_{i,j}}]\!]_1) = 1$. Based on the equation governing how $\mathcal{S}_b$ computes $\vec{t}_{b,i}$, we have $T_{b,i} - M_{b,i} = (\sum_{j\in\mathbf{X}} t_{b,i,j}) - (\sum_{j\in\mathbf{X}} m_{b,i,j}) = (\sum_{j\in\mathbf{X}} m_{b,i,j} + \Delta_b \cdot y_{i,j}) - (\sum_{j\in\mathbf{X}} m_{b,i,j}) = \Delta_b \cdot \sum_{j\in\mathbf{X}} y_{i,j} = \Delta_b$. It follows that $\eta_{b,i} \cdot (T_{b,i} - \Delta_b) = \eta_{b,i} \cdot M_{b,i}$. Since the servers set $t_{b,i,j}$ and $m_{b,i,j}$ to zero for every discarded $\mathcal{C}_i$, we have $\vec{\eta_b} \cdot (\vec{T_b} - \vec{\delta_b}) = \vec{\eta_b} \cdot \vec{M_b}, \forall b \in \{0,1\}$. Therefore, this check succeeds if both parties follow the protocol honestly.

If the property does not hold, since η_0 and η_1 are sampled uniformly at random, the check will fail with overwhelming probability. To see why, suppose one of the servers committed in the previous phase to shares such that for a subset $I \subseteq [\mathsf{N}]$ we have $\sum_{j\in\mathbf{X}} y_{i,j} = e_i \neq 1$ for every $i \in I$. In this case, $\vec{\eta_b} \cdot (\vec{T_b} - \vec{\delta_b}) - \vec{\eta_b} \cdot \vec{M_b} = \Delta_b \cdot \sum_{i\in I} \eta_{b,i} \cdot (e_i - 1)$. Because the servers committed to the VDPF output shares before jointly sampling $\vec{\eta_0}$ and $\vec{\eta_1}$, which are sampled uniformly at random, this value will be nonzero with overwhelming probability.

Additionally, a malicious $\mathcal{S}_{1-b}$ might try to send an incorrect h_b value to $\mathcal{S}_b$, for by using an incorrect $\vec{\eta_b}$. However, to cheat, $\mathcal{S}_{1-b}$ would need to guess Δ_b correctly, since it must produce $h_b = \mathrm{H}(\vec{\eta_b} \cdot (\vec{T_b} - \vec{\delta_b}))$ to pass this validation step. Since the probability of guessing Δ_b is negligible, this attack is infeasible.

Step 3c: One-Hot Position Consistency Property. This property ensures $\vec{y}_{i,\alpha_i} = 1$, i.e., the unique 1 in the one-hot vector $\vec{y_i}$ is located at position α_i, the input of an honest $\mathcal{C}_i$. Our

goal is to verify that the following evaluates to zero:

$$(\alpha_i - \sum_{j \in \mathbf{X}} j \cdot \vec{y}_{i,j}) \cdot \Delta_b \text{ for } b \in \{0,1\}. \tag{2}$$

Given that the two prior checks have already confirmed that $\vec{y}_i$ is a valid one-hot vector with its non-zero entry equal to one, we know the above expression evaluates to zero if and only if the index of the 1 in $\vec{y}_i$ is exactly α_i.

To achieve this, the servers first exchange the ciphertexts $\mathsf{padenc}_{b,i} \coloneqq \mathsf{EG.Enc}(\mathsf{pk}_b, \mathsf{inppad}_{b,i} + \sum_{j \in \mathbf{X}} j \cdot m_{b,i,j})$ and homomorphically compute:

$$\mathsf{inpenc}_{b,i} \cdot \mathsf{padenc}_{b,i}^{-1} \cdot \mathsf{EG.Enc}(\mathsf{pk}_b, \sum_{j \in \mathbf{X}} j \cdot t_{b,i,j}) \text{ for } b \in \{0,1\}. \tag{3}$$

Recall that $\mathsf{inpenc}_{b,i}$ is the encryption of $\alpha_i \cdot \Delta_b + \mathsf{inppad}_{b,i}$, provided by the client during the client input phase. Given this, the homomorphic computation yields an encryption of $\alpha_i \cdot \Delta_0 + \sum_{j \in \mathbf{X}} j \cdot m_{b,i,j} - \sum_{j \in \mathbf{X}} j \cdot t_{b,i,j}$. Using the relation from Step 2 of the histogram computation phase, namely $\vec{t}_{b,i} = \vec{m}_{b,i} + \Delta_b \cdot \vec{y}_i$, the above expression simplifies to Eq. (2). Therefore, the servers can verify the one-hot position consistency property by checking whether the decrypted value of the resulting ciphertext in Eq. (3) equals zero.

Note that if one of $\mathsf{inpenc}_{0,i}, \mathsf{inpenc}_{1,i}$ was incorrect, the ciphertexts resulting from the homomorphic computation will be non-zero except with negligible probability, as a result of the security property of the MAC-based commitment.

5.4 Output Phase

Lastly, the servers compute and output the final histogram. For each $b \in 0, 1$, $\mathcal{S}_b$ begins by computing and sending $[\![\mathsf{cnt}_j]\!]_b \coloneqq sum_{i \in [\mathsf{N}] \setminus Ł} [\![y_{i,j}]\!]_b$ for every $j \in \mathbf{X}$. After exchanging these values, both servers compute the aggregated count for each position as $\mathsf{cnt}_j \coloneqq [\![\mathsf{cnt}_j]\!]_0 + [\![\mathsf{cnt}_j]\!]_1$ for every $j \in \mathbf{X}$. We can see $\mathsf{cnt}_j = \sum_{i \in [\mathsf{N}] \setminus Ł} y_{i,j}$ where each $\vec{y}_i$ is a one-hot vector with $y_{i,\alpha_i} = 1$. This means that cnt_x is equal to the number of undiscarded clients who submitted x as their input. Thus, the servers output the final histogram as $\mathsf{HIST} \coloneqq (\mathsf{cnt}_1, \mathsf{cnt}_2, \ldots, \mathsf{cnt}_{\mathbf{X}})$.

However, if a malicious server sends incorrect shares to the other party, it could corrupt the final histogram. To protect against this attack vector, we require both servers $\mathcal{S}_b$ to commit to the values $([\![\mathsf{cnt}_j]\!]_b, \mathsf{aux}_{b,j})$ using cryptographic hashes and open these commitments before proceeding, where $\mathsf{aux}_{b,j} \coloneqq \sum_{i \in [\mathsf{N}] \setminus Ł} m_{1-b,i,j}$. $\mathcal{S}_b$ will abort if the following consistency check fails $(\mathsf{aux}_{1-b,j} + \Delta_b \cdot \mathsf{cnt}_j) \neq \sum_{i \in [\mathsf{N}] \setminus Ł} t_{b,i,j}$. This check is sound because, assuming both servers follow the protocol correctly, we have:

$$\begin{aligned}(\mathsf{aux}_{1-b,j} + \Delta_b \cdot \mathsf{cnt}_j) &= \sum_{i \in [\mathsf{N}] \setminus Ł} m_{b,i,j} + \Delta_b \cdot \mathsf{cnt}_j = \sum_{i \in [\mathsf{N}] \setminus Ł} m_{b,i,j} + \Delta_b \cdot \sum_{i \in [\mathsf{N}] \setminus Ł} y_{i,j} \\ &= \sum_{i \in [\mathsf{N}] \setminus Ł} (m_{b,i,j} + \Delta_b \cdot y_{i,j}) = \sum_{i \in [\mathsf{N}] \setminus Ł} t_{b,i,j}.\end{aligned}$$

This mechanism ensures integrity because if a malicious $\mathcal{S}_b$ attempts to send an incorrect value $[\![\mathsf{cnt}_j]\!]_b$, it must also know a corresponding $\mathsf{aux}_{b,j}$ that satisfies the above equality. Achieving this would require guessing the honest server's secret key Δ_{1-b}, which is computationally infeasible. As a result, any such forgery attempt will cause the protocol to abort with overwhelming probability.

We establish the security of our Paras scheme by proving Theorem 1, with the full proof provided in Section 6.

Theorem 1. *Assuming H is a programmable and observable random oracle, VDPF is a verifiable DPF, and EG.Enc is an additively homomorphic IND-CPA encryption scheme with $\mathsf{NIZK}_{\mathsf{EG.Dec}}$ as the underlying NIZK argument for proof of correct decryption to 0. Assuming a PPT adversary $\mathcal{A}$ maliciously corrupts a list $\widehat{\text{Ł}} \subseteq [N]$ of clients and server $\mathcal{S}_1$ (or $\mathcal{S}_0$), then Π_{Paras} securely implements $\mathcal{F}_{\mathsf{HIST}}$ against $\mathcal{A}$ in the random oracle and $(\mathcal{F}^p_{\mathsf{VOLE}}, \mathcal{F}_{\mathsf{aBV}})$ model.*

Parameters:
- *Inputs:* N Clients $\mathcal{C}_i$, each inputs $\alpha_i \in \mathbf{X}$, and histogram input domain $\mathbf{X} := [|\mathbf{X}|]$. Two servers $\mathcal{S}_0, \mathcal{S}_1$ with no input.
- *Primitives*: $\mathcal{F}^p_{\mathsf{VOLE}}$ (Fig. 1), AHE $(\mathsf{EG.Enc}, \mathsf{EG.Dec})$, VDPF (Sec. 2.1), $(\mathsf{NIZK}_{\mathsf{EG.Dec}}.\mathsf{Prove}, \mathsf{NIZK}_{\mathsf{EG.Dec}}.\mathsf{Verify})$ for proof of correct decryption of AHE ciphertext to 0, and random oracle $\mathrm{H} : \{0,1\}^* \to \{0,1\}^\kappa$.

[Offline] Server Initialization:
1. *VOLE Preprocessing:* For each $b \in \{0,1\}$, $\mathcal{S}_b$ and $\mathcal{S}_{1-b}$ call $\mathcal{F}^p_{\mathsf{VOLE}}$ with input $(\mathsf{init}, \mathsf{sid}_b)$ as receiver and sender respectively. $\mathcal{S}_b$ obtains Δ_b as output. $\mathcal{S}_b$ and $\mathcal{S}_{1-b}$ call $\mathcal{F}^p_{\mathsf{VOLE}}$ with input $(\mathsf{extend}, \mathsf{sid}_b, |\mathbf{X}| \times \mathsf{N})$ as receiver and sender respectively to obtain $(\vec{k}_{b,0}, \ldots, \vec{k}_{b,\mathsf{N}-1})$ and $(\vec{x}_b, \vec{m}_b) := (\vec{x}_{b,0}, \vec{m}_{b,0}, \ldots, \vec{x}_{b,\mathsf{N}-1}, \vec{m}_{b,\mathsf{N}-1})$ respectively where $\vec{m}_{b,i}, \vec{k}_{b,i} \in \mathbb{F}^{|\mathbf{X}|}$ and $\vec{x}_{b,i} \in \mathbb{F}^{|\mathbf{X}|}$ for $i \in [\mathsf{N}]$.
2. *Posting Public Keys and Server Encryptions:* For $b \in \{0,1\}$, server $\mathcal{S}_b$ samples $(\mathsf{pk}_b, \mathsf{sk}_b)$, and posts pk_b and $\mathsf{enc}_b = \mathsf{EG.Enc}(\mathsf{pk}_b, \Delta_b)$.
3. *Checking Δ_b is encrypted in $\mathsf{EG.Enc}(\mathsf{pk}_b, \Delta_b)$ and used in VOLE Session sid_b:* For each $b \in \{0,1\}$, $\mathcal{S}_b$ and $\mathcal{S}_{1-b}$ perform the following:
   a. $\mathcal{S}_b$ and $\mathcal{S}_{1-b}$ call $\mathcal{F}^p_{\mathsf{VOLE}}$ with input $(\mathsf{extend}, \mathsf{sid}_b, 1)$ as receiver and sender respectively to obtain k'_b and (x'_b, m'_b) and respectively.
   b. $\mathcal{S}_b$ commits to k'_b by sending $\widehat{e_b} := \mathsf{EG.Enc}(\mathsf{pk}_b, k'_b)$ to $\mathcal{S}_{1-b}$.
   c. $\mathcal{S}_{1-b}$ samples m''_b, computes $e''_b := \mathsf{EG.Enc}(\mathsf{pk}_b, m''_b) \cdot \mathsf{enc}_b^{x'_b}$. The $\mathcal{S}_{1-b}$ also computes $m'_b - m''_b$ as the proof that the same x'_b is committed in the VOLE session and encrypted in e''_b. $\mathcal{S}_{1-b}$ sends $(e''_b, m'_b - m''_b)$ to $\mathcal{S}_b$.
   d. $\mathcal{S}_b$ computes $E'_b := \widehat{e_b} \cdot (e''_b)^{-1} \cdot \mathsf{EG.Enc}(\mathsf{pk}_b, m'_b - m''_b, 0)^{-1}$.
      - If $\mathsf{EG.Dec}(\mathsf{sk}_b, E'_b) \neq 0$, $\mathcal{S}_b$ aborts as $\mathcal{S}_{1-b}$ did not encrypt x'_b in e''_b.
      - Otherwise, $\mathcal{S}_b$ computes a NIZK proof of correct decryption as $\pi'_b := \mathsf{NIZK}_{\mathsf{EG.Dec}}.\mathsf{Prove}(E'_b, \mathsf{sk}_b)$ and sends it to $\mathcal{S}_{1-b}$.
        - $\mathcal{S}_{1-b}$ computes $E''_b := \widehat{e_b} \cdot (e''_b)^{-1} \cdot \mathsf{EG.Enc}(\mathsf{pk}_b, m'_b - m''_b, 0)^{-1}$. It aborts if $\mathsf{NIZK}_{\mathsf{EG.Dec}}.\mathsf{Verify}(E''_b) \neq 1$, else continue.
4. *Variables Initialization:* For $b \in \{0,1\}$, $\mathcal{S}_b$ initializes a list $\text{Ł} := \emptyset$ for storing the identities of malicious clients caught cheating.

[Online] Client Input Phase: For $i \in [\mathsf{N}]$, ▷ Repeated for N clients.
1. Client $\mathcal{C}_i$ with input $\alpha_i \in \mathbf{X}$ prepares VDPF keys $(\mathsf{key}_{0,i}, \mathsf{key}_{i,1}) := \mathsf{VDPF.Gen}(1^\kappa, \alpha_i, 1)$.
2. For each $b \in \{0,1\}$, the $\mathcal{C}_i$ samples $\mathsf{inppad}_{b,i} \leftarrow \mathbb{F}$, and compute $\mathsf{inpenc}_{b,i} := \mathsf{enc}_b^{\alpha_i} \cdot \mathsf{EG.Enc}(\mathsf{pk}_b, \mathsf{inppad}_{b,i})$.
3. The client sends $(\mathsf{key}_{0,i}, \mathsf{inpenc}_{0,i}, \mathsf{inppad}_{1,i})$ to $\mathcal{S}_0$ and $(\mathsf{key}_{1,i}, \mathsf{inpenc}_{1,i}, \mathsf{inppad}_{0,i})$ to $\mathcal{S}_1$.

Figure 7: Part 1 of our histogram protocol Π_{Paras}, continues in Fig. 8.

6 Security Proofs

6.1 Security Proof of Π_{Paras}

Theorem 2. *Assuming H is a programmable and observable random oracle, VDPF is a verifiable DPF, and EG.Enc is an additively homomorphic IND-CPA encryption scheme with $\mathsf{NIZK}_{\mathsf{EG.Dec}}$ as the underlying NIZK argument for proof of correct decryption to 0. Assuming a PPT adversary $\mathcal{A}$ maliciously corrupts a list $\widehat{\text{Ł}} \subseteq [N]$ of clients and $\mathcal{S}_1$ (or $\mathcal{S}_0$), then Π_{Paras} securely implements $\mathcal{F}_{\mathsf{HIST}}$ against $\mathcal{A}$ in the random oracle and $(\mathcal{F}^p_{\mathsf{VOLE}}, \mathcal{F}_{\mathsf{aBV}})$ model.*

We assume that there is a PPT adversary $\mathcal{A}$ that maliciously corrupts a list $\widehat{\text{Ł}}$ of clients and $\mathcal{S}_1$. We provide a PPT simulator algorithm Sim in Figs. 9, 10 for this corruption scenario, where Sim simulates the honest clients and $\mathcal{S}_0$ in the ideal world execution with access to $\mathcal{F}_{\mathsf{HIST}}$. At the end of the execution, the ideal world adversary view is produced. We prove that Π_{Paras} securely implements $\mathcal{F}_{\mathsf{HIST}}$ (Fig. 2) by showing that the real and

[Online] Histogram Computation:

1. **Preventing Malicious DPFs:** For $i \in [\mathsf{N}]$, $b \in \{0,1\}$, server $\mathcal{S}_b$ computes:
   a. *VDPF Evaluation:* $\mathcal{S}_b$ locally evaluates the VDPF on the client-provided keys as $(\llbracket \vec{y_i} \rrbracket_b, \pi_{b,i}) \coloneqq \mathsf{VDPF.BatchEval}(b, \mathsf{key}_{b,i}, \mathbf{X})$.
   b. *Proof Verification:* Servers exchange $\pi_{0,i}, \pi_{1,i}$ and run $\mathsf{VDPF.Verify}(\pi_{0,i}, \pi_{1,i})$. If the verification fails, they discard $\mathcal{C}_i$'s input and update $Ł \coloneqq Ł \cup i$.
   c. *1-hot Verification:* $\mathcal{S}_b$ sends $\sum_{j \in \mathbf{X}} \llbracket y_{i,j} \rrbracket_b$ to $\mathcal{S}_{1-b}$. They compute $\beta_i \coloneqq \sum_{j \in \mathbf{X}} \llbracket y_{i,j} \rrbracket_0 + \sum_{j \in \mathbf{X}} \llbracket y_{i,j} \rrbracket_1$. If $\beta_i \neq 1$, they discard $\mathcal{C}_i$'s input and update Ł.
2. **Server's Commitment to the Share of Histogram Computation:** For $\mathcal{C}_i$ where $i \in [\mathsf{N}] \setminus Ł$, the servers commit to their $\llbracket \vec{y_i} \rrbracket_b$ shares using preprocessed VOLE outputs as follows. Here, we denote $\vec{y_i} = \llbracket \vec{y_i} \rrbracket_0 + \llbracket \vec{y_i} \rrbracket_1 \coloneqq (y_{i,1}, y_{i,2} \ldots y_{i,|\mathbf{X}|})$. For $b \in \{0,1\}$, $\mathcal{S}_{1-b}$ sends $\vec{w}_{b,i} \coloneqq \llbracket \vec{y_i} \rrbracket_{1-b} - \vec{x}_{b,i}$. $\mathcal{S}_b$ updates $\vec{k}_{b,i} \coloneqq \vec{k}_{b,i} + \Delta_b \cdot \vec{w}_{b,i}$, and computes $\vec{t}_{b,i} \coloneqq \vec{k}_{b,i} + \Delta_b \cdot \llbracket \vec{y_i} \rrbracket_b$. The following equations are satisfied at the end of the commitment phase: $\vec{k}_{0,i} \coloneqq \vec{m}_{0,i} + \Delta_0 \cdot \llbracket \vec{y_i} \rrbracket_1, \vec{t}_{0,i} \coloneqq \vec{m}_{0,i} + \Delta_0 \cdot \vec{y}_i, \vec{k}_{1,i} \coloneqq \vec{m}_{1,i} + \Delta_1 \cdot \llbracket \vec{y_0} \rrbracket_1, \vec{t}_{1,i} \coloneqq \vec{m}_{1,i} + \Delta_1 \cdot \vec{y}_i$.
3. **Preventing a Malicious Server:**
   a. ***Binary Vector Property:*** *Verifying each committed $y_{i,j} \in \{0,1\}$ for every undiscarded client $\mathcal{C}_i$ and $j \in \mathbf{X}$:* For each $b \in \{0,1\}$, $\mathcal{S}_b$ invokes $\mathcal{F}_{\mathsf{aBV}}$ (Fig. 3) with inputs $\left(\{\vec{k}_{b,i}\}_{i \in [\mathsf{N}]}, \{\llbracket \vec{y_i} \rrbracket_b\}_{i \in [\mathsf{N}]}, \Delta_b, \{\vec{m}_{1-b,i}\}_{i \in [\mathsf{N}]}\right)$. If $\mathcal{F}_{\mathsf{aBV}}$ outputs 0, the servers abort.
   b. ***One-Hot Sum Property:*** *Verifying committed $\sum_{j \in \mathbf{X}} \vec{y_{i,j}} = 1$ for every undiscarded $\mathcal{C}_i$:*
      i. For $b \in \{0,1\}$, $\mathcal{S}_b$ computes $\vec{T_b} \coloneqq \left(\sum_{j \in \mathbf{X}} t_{b,0,j}, \ldots, \sum_{j \in \mathbf{X}} t_{b,\mathsf{N}-1,j}\right)$ and $\vec{M}_{1-b} \coloneqq \left(\sum_{j \in \mathbf{X}} m_{1-b,0,j}, \ldots, \sum_{j \in \mathbf{X}} m_{1-b,\mathsf{N}-1,j}\right)$ where $t_{b,i,j} = m_{1-b,i,j} \coloneqq 0$ for $i \in Ł$.
      ii. Both servers sample random $\vec{\eta_0}, \vec{\eta_1} \leftarrow \mathbb{F}^{\mathsf{N}}$. For $b \in \{0,1\}$, $\mathcal{S}_{1-b}$ computes hash $h_b \coloneqq \mathrm{H}(\vec{\eta_b} \cdot \vec{M_b})$ and sends it to $\mathcal{S}_b$.
      iii. For $b \in \{0,1\}$, $\mathcal{S}_b$ aborts if $h_b \neq \mathrm{H}(\vec{\eta_b} \cdot (\vec{T_b} - \vec{\delta_b}))$, where $\delta_{b,i} = 0$ if $i \in Ł$, and $\delta_{b,i} = \Delta_b$ otherwise.
   c. ***One-Hot Position Consistency Property:*** *Verifying that the servers have committed to the client's input by ensuring that the one-hot vector y_i has a 1 at index α_i for each undiscarded client $\mathcal{C}_i$.* For each $\mathcal{C}_{i \in [\mathsf{N}] \setminus Ł}$: $\mathcal{S}_{1-b}$ sends $\mathsf{padenc}_{b,i} \coloneqq \mathsf{EG.Enc}(\mathsf{pk}_b, \mathsf{inppad}_{b,i} + \sum_{j \in \mathbf{X}} j \cdot m_{b,i,j})$ to $\mathcal{S}_b$. $\mathcal{S}_b$ checks that $\mathsf{inpenc}_{b,i} \cdot \mathsf{padenc}_{b,i}^{-1} \cdot \mathsf{EG.Enc}(\mathsf{pk}_b, \sum_{j \in \mathbf{X}} j \cdot t_{b,i,j})$ decrypts to 0 using sk_b. If not, then discard this client's input and update $Ł \coloneqq Ł \cup i$.
4. **Output:** For $b \in \{0,1\}$, $j \in \mathbf{X}$ the servers aggregate the honest clients' outputs.
   a. $\mathcal{S}_b$ computes $\llbracket \mathsf{cnt}_j \rrbracket_b \coloneqq \sum_{i \in [\mathsf{N}] \setminus Ł} \llbracket y_{i,j} \rrbracket_b$ and $\mathsf{aux}_{b,j} \coloneqq \sum_{i \in [\mathsf{N}] \setminus Ł} m_{1-b,i,j}$.
   b. $\mathcal{S}_b$ commits to $(\llbracket \mathsf{cnt}_j \rrbracket_b, \mathsf{aux}_{b,j})$ and then opens it. They compute $\mathsf{cnt}_j = \llbracket \mathsf{cnt}_j \rrbracket_0 + \llbracket \mathsf{cnt}_j \rrbracket_1$.
   c. Server $\mathcal{S}_b$ aborts if $(\mathsf{aux}_{1-b,j} + \Delta_b \cdot \mathsf{cnt}_j) \neq \sum_{i \in [\mathsf{N}] \setminus Ł} t_{b,i,j}$.

If none of the servers has aborted, then both output $\mathsf{HIST} \coloneqq (\mathsf{cnt}_1, \mathsf{cnt}_2, \ldots, \mathsf{cnt}_{\mathbf{X}})$.

Figure 8: Part 2 of our histogram protocol Π_{Paras}. Continuing from Fig. 7.

ideal world adversary views are indistinguishable. The corruption of $\mathcal{S}_0$ can be argued similarly, where Sim would simulate $\mathcal{S}_1$ (instead of $\mathcal{S}_0$) running the same protocol steps.

To simulate an honest client, the simulator $\mathsf{Sim}_{\mathsf{Paras}}$ runs the honest client algorithm on input X_1. In the simulation of the server algorithm, the simulator simulates $\mathcal{F}^p_{\mathsf{VOLE}}$ to obtain the inputs of the corrupt clients and the corrupt server's view. Then $\mathsf{Sim}_{\mathsf{Paras}}$ invokes $\mathcal{F}_{\mathsf{HIST}}$ with the extracted inputs of the corrupt clients to obtain the output, and then simulates the output in the simulated protocol. The adversarial server cannot double vote or change the votes of the honest parties because it does not know Δ_0 due to the hiding property of $\mathcal{F}^p_{\mathsf{VOLE}}$. We argue indistinguishability between the real and ideal world execution as follows:

- HYB_0: This is the real-world execution of the protocol.
- HYB_1: This is the same as HYB_0, except the $\mathcal{F}^p_{\mathsf{VOLE}}$ invocations are simulated and $\mathsf{Sim}_{\mathsf{Paras}}$ obtains the inputs of $\mathcal{S}_1^*$ in both VOLE sessions. Indistinguishability follows in the $\mathcal{F}^p_{\mathsf{VOLE}}$ model.
- HYB_2 : This is same as HYB_1, except $\mathsf{Sim}_{\mathsf{Paras}}$ computes the NIZK π'_0 by invoking the NIZK simulator. Indistinguishability follows from the ZK property of the NIZK protocol.

[Offline] Server Initialization:

1. *VOLE Preprocessing for Session 0:* $\mathsf{Sim}_{\mathsf{Paras}}$ emulates $\mathcal{F}^p_{\mathsf{VOLE}}$ for session sid_0 with a random Δ_0, where $\mathsf{Sim}_{\mathsf{Paras}}$ and $\mathcal{S}_1^*$ is the receiver and sender respectively. When $\mathcal{S}_1^*$ invokes $\mathcal{F}^p_{\mathsf{VOLE}}$ with input $(\vec{x}_0, \vec{m}_0) \coloneqq (\vec{x}_{0,0}, \vec{m}_{0,0}, \ldots, \vec{x}_{0,\mathsf{N}-1}, \vec{m}_{0,\mathsf{N}-1})$ as a corrupt sender, $\mathsf{Sim}_{\mathsf{Paras}}$ receives it from $\mathcal{F}^p_{\mathsf{VOLE}}$ and stores it.
2. *VOLE Preprocessing for Session 1:* When $\mathcal{S}_1^*$ calls $\mathcal{F}^p_{\mathsf{VOLE}}$ as a corrupt receiver with input Δ_1 for initialization of VOLE session sid_1, $\mathsf{Sim}_{\mathsf{Paras}}$ obtains Δ_1 and stores it. When $\mathcal{S}_1^*$ invokes $\mathcal{F}^p_{\mathsf{VOLE}}$ with input $(\vec{k}_{1,0}, \ldots, \vec{k}_{1,\mathsf{N}-1})$ as a corrupt receiver, $\mathsf{Sim}_{\mathsf{Paras}}$ receives it from $\mathcal{F}^p_{\mathsf{VOLE}}$ and stores it.
3. *Posting Public Keys and Server Encryptions:* Server $\mathsf{Sim}_{\mathsf{Paras}}$ samples $(\mathsf{pk}_0, \mathsf{sk}_0)$ and posts $\mathsf{enc}_0 = \mathsf{EG.Enc}(\mathsf{pk}_0, 0)$. Server $\mathcal{S}_1^*$ posts pk_1 and enc_1.
4. *Checking Δ_b is encrypted in $\mathsf{EG.Enc}(\mathsf{pk}_b, \Delta_b)$ and used in VOLE Session sid_b for $b \in \{0,1\}$:* $\mathsf{Sim}_{\mathsf{Paras}}$ and $\mathcal{S}_1^*$ perform the following:
   a. $\mathsf{Sim}_{\mathsf{Paras}}$ simulates the two $\mathcal{F}^p_{\mathsf{VOLE}}$ calls to obtain (x'_0, m'_0, k'_0) and (x'_1, m'_1, k'_1) in sid_0 and sid_1 respectively.
   b. $\mathsf{Sim}_{\mathsf{Paras}}$ commits to 0 by sending $\widehat{e_0} \coloneqq \mathsf{EG.Enc}(\mathsf{pk}_0, 0)$ to $\mathcal{S}_1$ and $\widehat{e_1}$ from $\mathcal{S}_1^*$.
   c. $\mathcal{S}_1^*$ sends $(e''_0, m'_0 - m''_0)$ to $\mathsf{Sim}_{\mathsf{Paras}}$ and $\mathsf{Sim}_{\mathsf{Paras}}$ sends $(e''_1, m'_1 - m''_1)$ by running the honest server algorithm.
   d. $\mathsf{Sim}_{\mathsf{Paras}}$ computes m''_0 from $m'_0 - m''_0$ and m'_0. $\mathsf{Sim}_{\mathsf{Paras}}$ checks that $E'_0 \coloneqq (e''_0)^{-1} \cdot \mathsf{EG.Enc}(\mathsf{pk}_0, m''_0 + x'_0 \cdot \Delta_0, 0)^{-1}\cdot$ decrypts to 0. If not, then abort; else invoke the NIZK simulator to simulate the proof π'_0 and send it to $\mathcal{S}_1^*$.
   e. Upon obtaining π'_1 from $\mathcal{S}_1^*$, if the NIZK proof verification fails then $\mathsf{Sim}_{\mathsf{Paras}}$ aborts, else continue.
5. *Variables Initialization:* For $b \in \{0,1\}$, $\mathcal{S}_b$ initializes a list $Ł \coloneqq \emptyset$ for storing the identities of malicious clients caught cheating.

[Online] Client Input Phase: For $i \in [\mathsf{N}]$, ▷ Repeated for N clients.

1. $\mathsf{Sim}_{\mathsf{Paras}}$ simulates the honest clients with input $X_1 \in \mathbf{X}$.
2. The corrupt clients send $(\mathsf{key}_{0,i}, \mathsf{inpenc}_{0,i}, \mathsf{inppad}_{1,i})$ to $\mathsf{Sim}_{\mathsf{Paras}}$ and $(\mathsf{key}_{1,i}, \mathsf{inpenc}_{1,i}, \mathsf{inppad}_{0,i})$ to $\mathcal{S}_1^*$.

Figure 9: Simulator algorithm $\mathsf{Sim}_{\mathsf{Paras}}$ (that simulates $\mathcal{S}_0$) for Π_{Paras} against an adversary $\mathcal{A}$ (that corrupts $\mathcal{S}_1$ and list $\widehat{Ł} \subseteq [\mathsf{N}]$ of clients).

[Online] Histogram Computation:

1. **Preventing Malicious DPF Encodings:** $\mathsf{Sim}_{\mathsf{Paras}}$ runs this step following the honest server algorithm to obtain $(\llbracket \vec{y_i} \rrbracket_0, \pi_{0,i})$ and construct the discard list Ł.
2. **Server's Commitment to the share of Histogram Computation:** For $\mathcal{C}_i$ where $i \in [\mathsf{N}] \setminus Ł$, when $\mathcal{S}_1^*$ sends $\vec{w}_{0,i}$, $\mathsf{Sim}_{\mathsf{Paras}}$ computes $\llbracket \vec{y_i} \rrbracket_1 \coloneqq \vec{w}_{0,i} + \vec{x_{0,i}}$. $\mathsf{Sim}_{\mathsf{Paras}}$ computes $\vec{w}_{1,i}$ honestly and sends it to $\mathcal{S}_1^*$. $\mathsf{Sim}_{\mathsf{Paras}}$ computes $\vec{y}_i = \llbracket \vec{y_i} \rrbracket_0 + \llbracket \vec{y_i} \rrbracket_1 \coloneqq (y_{i,1}, y_{i,2} \ldots y_{i,|\mathbf{X}|})$ and extracts the input of corrupt client $\mathcal{C}_i$ as α_j s.t. $y_{i,j} == 1$ and $y_{i,\forall j' \neq j} == 0$. If there are two indices j and j' s.t. $y_{i,j} \neq 0$ and $y_{i,j'} \neq 0$, then skip the extraction step.
3. **Preventing a Malicious Server:**
   a. ***Binary Vector Property*** – *Verifying each committed $y_{i,j} \in \{0,1\}$ for every undiscarded client $\mathcal{C}_i$ and $j \in \mathbf{X}$:* For each $b \in \{0,1\}$, the servers invoke $\mathcal{F}_{\mathsf{aBV}}$ (Fig. 3) to perform the check. $\mathsf{Sim}_{\mathsf{Paras}}$ invokes the simulator of $\mathcal{F}_{\mathsf{aBV}}$ to perform the simulation. If $\mathcal{F}_{\mathsf{aBV}}$ outputs 0, then the servers abort; else they continue.
   b. ***One-Hot Sum Property*** – *Verifying each committed $(\sum_{j \in \mathbf{X}} \vec{y_{i,j}}) = 1$ for every undiscarded client $\mathcal{C}_i$:*
      i. If there are two indices j and j' s.t. $y_{i,j} \neq 0$ and $y_{i,j'} \neq 0$, then $\mathsf{Sim}_{\mathsf{Paras}}$ aborts.
      ii. If there is a $y_{i,j}$ s.t. $y_{i,j} \neq \{0,1\}$ then $\mathsf{Sim}_{\mathsf{Paras}}$ aborts.
      iii. Otherwise, $\mathsf{Sim}_{\mathsf{Paras}}$ executes the honest server algorithm.
   c. ***One-Hot Position Consistency Property*** – *Verifying that the servers committed to the client provided inputs for every undiscarded client $\mathcal{C}_i$:*
      i. For each client $\mathcal{C}_i$ where $i \in [\mathsf{N}] \setminus Ł$: $\mathsf{Sim}_{\mathsf{Paras}}$ checks that $\vec{k_{0,i}} \coloneqq \vec{m_{0,i,j}} + \llbracket \vec{y_i} \rrbracket_1$ and discards the client if the check fails. Then, $\mathsf{Sim}_{\mathsf{Paras}}$ runs the honest protocol of $\mathcal{S}_0$ to find out additional corrupt clients.
4. **Output Phase:** For $b \in \{0,1\}$, $j \in \mathbf{X}$ the servers aggregate the clients' outputs.
   a. $\mathsf{Sim}_{\mathsf{Paras}}$ invokes $\mathcal{F}_{\mathsf{HIST}}$ with inputs of corrupt clients (represented by list $\widehat{Ł}$) and the discard list Ł. If $\mathsf{Sim}_{\mathsf{Paras}}$ has aborted in the above steps, then it instructs $\mathcal{F}_{\mathsf{HIST}}$ to abort as well.
   b. $\mathsf{Sim}_{\mathsf{Paras}}$ obtains the output histogram HIST. $\mathsf{Sim}_{\mathsf{Paras}}$ computes $\llbracket \mathsf{cnt}_j \rrbracket_1$ and $\mathsf{aux}_{1,j}$ for $j \in \mathbf{X}$, and sets $\llbracket \mathsf{cnt}_j \rrbracket_0 \coloneqq \mathsf{HIST} - \llbracket \mathsf{cnt}_j \rrbracket_1$ and computes $\mathsf{aux}_{0,j} \coloneqq \sum_{i \in [\mathsf{N}] \setminus Ł} t_{1,i,j} - \Delta_1 \cdot \mathsf{cnt}_j)$.
   c. $\mathsf{Sim}_{\mathsf{Paras}}$ commits to a random value using the hash, and then programs the hash to open to $(\llbracket \mathsf{cnt}_j \rrbracket_0, \mathsf{aux}_{1,j})$.
   d. $\mathsf{Sim}_{\mathsf{Paras}}$ runs the honest server algorithm.

   If none of the servers aborts, then both servers output the final aggregated histogram vector $\mathsf{HIST} \coloneqq (\mathsf{cnt}_1, \mathsf{cnt}_2, \ldots, \mathsf{cnt}_{\mathbf{X}})$.

Figure 10: Simulator algorithm $\mathsf{Sim}_{\mathsf{Paras}}$ (that simulates $\mathcal{S}_0$) for Π_{Paras} against an adversary $\mathcal{A}$ (that corrupts $\mathcal{S}_1$ and list $\widehat{Ł} \subseteq [\mathsf{N}]$ of clients).

- HYB_3 : This is same as HYB_2, except $\mathsf{Sim}_{\mathsf{Paras}}$ commits to 0 in $\widehat{e_0}$. Indistinguishability follows from the IND-CPA property of the encryption scheme.

- HYB_4 : This is the same as HYB_3, except $\mathsf{Sim}_{\mathsf{Paras}}$ simulates $\mathcal{F}_{\mathsf{aBV}}$ by invoking the simulator and aborts the protocol if any of the commit $y_{i,j}$ value is not a bit. Indistinguishability follows in the $\mathcal{F}_{\mathsf{aBV}}$-model.

- HYB_5 : This is the same as HYB_4, except $\mathsf{Sim}_{\mathsf{Paras}}$ aborts in One-Hot Sum Property check if there are two indices j and j' s.t. $y_{i,j} \neq 0$ and $y_{i,j'} \neq 0$, or there is a single $y_{i,j} \neq \{0,1\}$. The adversarial $\mathcal{S}_1^*$ passes the one-hot sum property check while committing to two non-zero values $y_{i,j}$ and $y_{i,j'}$ or if the sum is not equal to 1 - if it either breaks the collision resistance of the hash function, or if it guesses Δ_0 or if the random $\vec{\eta_0}$ is s.t. the check passes. The first event occurs with negligible probability in the random oracle model (assuming H is a random oracle), the second event occurs with negligible probability in $\mathcal{F}^p_{\mathsf{VOLE}}$ model and the third event occurs with statistically negligible probability since $\vec{\eta_0}$ is randomly chosen from a statistically large field.

- HYB_6 : This is the same as HYB_5, except $\mathsf{Sim}_{\mathsf{Paras}}$ discards the client if $\vec{k_{0,i}} \neq \vec{m_{0,i,j}} + [\![\vec{y_i}]\!]_1$ in One-hot consistency property check. Indistinguishability follows from the $\mathcal{F}^p_{\mathsf{VOLE}}$ model since $\mathcal{S}_1^*$ has to guess $k_{0,i,j}$ to pass the check if $y_i \neq 1$ at client's input index α_i.

- HYB_7 : This is the same as HYB_6, except $\mathsf{Sim}_{\mathsf{Paras}}$ discards the client's input if the VDPF proof fails to verify. Indistinguishability follows from the verifiability of the VDPF.

- HYB_7 : This is the same as HYB_6, except $\mathsf{Sim}_{\mathsf{Paras}}$ simulates the honest client algorithm with input X_1, extracts the corrupt clients' inputs, invokes $\mathcal{F}_{\mathsf{HIST}}$ with the inputs to obtain the output, and then simulates the output as per the simulation algorithm. This is the ideal world execution of the protocol. Indistinguishability follows from the preimage resistance of the hash function (modeled as a random oracle).

Input: $\mathsf{Sim}_{\mathsf{aBV}}$ obtains $(\vec{x}_0, \vec{m}_0, \vec{k}_1, \Delta_1)$ as the candidate inputs of $\mathcal{S}_1^*$. $\mathsf{Sim}_{\mathsf{aBV}}$ calls $\mathsf{Sim}_{\mathsf{VOLE}}$ with inputs $(\mathsf{Get\text{-}Key}, \mathsf{sid}_0)$ and $(\mathsf{Get\text{-}Keysid}_1)$ to receive Δ_0 and Δ_1.

Protocol Details:

1. *Pad Sampling for Session 0:*
   $\mathcal{S}_1^*$ and $\mathsf{Sim}_{\mathsf{aBV}}$ call $\mathcal{F}^p_{\mathsf{VOLE}}$ with $(\mathsf{extend}, \mathsf{sid}_0, 1)$ as sender and receiver to obtain (x'_0, m'_0) and k'_0, respectively.
   Pad Sampling for Session 1:
   $\mathsf{Sim}_{\mathsf{aBV}}$ and $\mathcal{S}_1^*$ call $\mathcal{F}^p_{\mathsf{VOLE}}$ with $(\mathsf{extend}, \mathsf{sid}_1, 1)$ as sender and receiver to obtain (x'_1, m'_1) and k'_1, respectively.
2. Servers jointly sample linear combiners $\vec{\delta_0}, \vec{\delta_1} \in \mathbb{F}_p^\ell$ for batch VOLE verification in sessions 0 and 1.
3. *Batch Verification for Session 0:*
   a. $\mathcal{S}_1^*$ sends $\mathsf{degzero}_1$ and degone_1 to $\mathsf{Sim}_{\mathsf{aBV}}$.
4. *Batch Verification for Session 1:*
   a. $\mathsf{Sim}_{\mathsf{aBV}}$ locally computes $\vec{t_1} := \vec{k}_1 + \vec{x}_0 \cdot \Delta_1$ and the challenge $\mathsf{chall}_1 := \vec{\delta}_1 \cdot (\vec{t}_1 \odot (\vec{t}_1 - \Delta_1)) + k'_1$ as the input of $\mathcal{S}_1^*$ to $\mathsf{Sim}_{\mathsf{adIPA}}$.
   b. $\mathsf{Sim}_{\mathsf{aBV}}$ samples $\mathsf{degzero}_0, \mathsf{degone}_0 \leftarrow \mathbb{F}$ and sends them to $\mathcal{S}_1^*$.
   c. $\mathsf{Sim}_{\mathsf{aBV}}$ computes $\mathsf{prod}_1 := \frac{(\mathsf{chall}_1 - (\mathsf{degzero}_0 + \mathsf{degone}_0 \cdot \Delta_1))}{2\Delta_1}$.
5. *Pairwise Product Check for both Sessions:*
   $\mathsf{Sim}_{\mathsf{aBV}}$ invokes $\mathsf{Sim}_{\mathsf{adIPA}}$ with inputs of $\mathcal{S}_1^*$ as $((\vec{x}_0, \vec{m}_0, \vec{k}_1, \Delta_1), (\vec{\delta}_0, \vec{\delta}_1, \mathsf{prod}_1))$ to obtain output $\mathsf{out}_{\mathsf{adIPA}}$.
   $\mathsf{Sim}_{\mathsf{aBV}}$ invokes $\mathcal{F}_{\mathsf{aBV}}(\vec{x}_0, \vec{m}_0, \vec{k}_1, \Delta_1)$ to obtain output $\mathsf{out}_{\mathsf{aBV}}$. $\mathsf{Sim}_{\mathsf{aBV}}$ sets $\mathsf{out} := \mathsf{False}$ if $(\mathsf{out}_{\mathsf{adIPA}} = \mathsf{False}) \vee (\mathsf{out}_{\mathsf{aBV}} = \mathsf{False})$. $\mathsf{Sim}_{\mathsf{aBV}}$ outputs out.

Figure 11: Simulator $\mathsf{Sim}_{\mathsf{aBV}}$ (that simulates $\mathcal{S}_0$) for Π_{aBV} against an adversary $\mathcal{A}$ (that corrupts $\mathcal{S}_1$).

6.2 Security Proof of Π_{aBV}

Theorem 3. *Assuming a PPT adversary $\mathcal{A}$ maliciously corrupting either $\mathcal{S}_0$ or $\mathcal{S}_1$, then Π_{aBV} (Fig. 6) securely implements $\mathcal{F}_{aBV}$ (Fig. 3) against $\mathcal{A}$ in $(\mathcal{F}^p_{VOLE}, \mathcal{F}_{adIPA})$ model.*

Proof. We assume that there is a PPT adversary $\mathcal{A}$ which maliciously corrupts $\mathcal{S}_1$ and we denote the corrupt server as $\mathcal{S}_1^*$. We provide a PPT simulator algorithm $\mathsf{Sim}_{\mathsf{aBV}}$ in Fig. 11 for this corruption scenario where $\mathsf{Sim}_{\mathsf{aBV}}$ simulates the honest $\mathcal{S}_0$ in the ideal world execution with access to $\mathcal{F}_{\mathsf{aBV}}$. At the end of the execution, the ideal world adversary view is produced. We prove that Π_{aBV} (Fig. 6) securely implements $\mathcal{F}_{\mathsf{aBV}}$ (Fig. 3) by showing that the real and ideal world adversary views are indistinguishable. The corruption of $\mathcal{S}_0$ can be argued similarly, where $\mathsf{Sim}_{\mathsf{aBV}}$ would simulate $\mathcal{S}_1$ (instead of $\mathcal{S}_0$) running the same protocol steps.

We argue indistinguishability between the real and ideal world execution:

- HYB_0: This is the real-world execution of the protocol Π_{aBV}.
- HYB_1: This is the same as HYB_0, except $\mathsf{Sim}_{\mathsf{aBV}}$ computes prod_1, samples random $\mathsf{degzero}_0, \mathsf{degone}_0 \leftarrow \mathbb{F}$ and sends them to $\mathcal{S}_1^*$ in *Batch Verification for Session 1* and invokes $\mathsf{Sim}_{\mathsf{adIPA}}$ in *Pairwise Product Check for both Sessions* with the inputs of $\mathcal{S}_1^*$ as $((\vec{x}_0, \vec{m}_0, \vec{k}_1, \Delta_1), (\vec{\delta}_0, \vec{\delta}_1, \mathsf{prod}_1))$ to obtain output $\mathsf{out}_{\mathsf{adIPA}}$ and outputs False if $\mathsf{out}_{\mathsf{adIPA}} = \mathsf{False}$.

  Adversary $\mathcal{A}$ distinguishes between the two hybrids if 1) $\mathcal{S}_1^*$ detects that $\mathsf{Sim}_{\mathsf{aBV}}$ simulated $\mathsf{degzero}_0$ and degone_0 in HYB_1, or 2) $\mathcal{S}_1^*$ uses a different input in its call to $\mathcal{F}_{\mathsf{adIPA}}$ other than $((\vec{x}_0, \vec{m}_0, \vec{k}_1, \Delta_1), (\vec{\delta}_0, \vec{\delta}_1, \mathsf{prod}_1))$. The first case is only distinguishable in the call to $\mathcal{F}_{\mathsf{adIPA}}$ where in HYB_0 the honest $\mathcal{S}_0$ invokes $\mathcal{F}_{\mathsf{adIPA}}$ with its inputs, and in HYB_1 this call is simulated by calling $\mathsf{Sim}_{\mathsf{adIPA}}$ which returns True. However, if $\mathcal{A}$ distinguishes between the two hybrids then it breaks the simulation security of the $\mathcal{F}_{\mathsf{adIPA}}$ model. We argue that the second case for distinguishing the hybrids also occurs with negligible probability. Recall, that the $\mathcal{F}_{\mathsf{adIPA}}$ functionality checks that $\vec{k}_0 = \vec{m}_0 + \vec{x}_0 \cdot \Delta_0$ and $\vec{k}_1 = \vec{m}_1 + \vec{x}_1 \cdot \Delta_1$ where $\mathcal{S}_0$ provides $(\vec{x}_1, \vec{m}_1, \vec{k}_0, \Delta_0)$ to $\mathcal{F}_{\mathsf{adIPA}}$ in HYB_0 and $\mathsf{Sim}_{\mathsf{aBV}}$ simulates this call in HYB_1 by calling $\mathsf{Sim}_{\mathsf{adIPA}}$. If the adversary is able to distinguish between the two then adversary is able to find a different set of $(\widehat{\vec{x}_0}, \widehat{\vec{m}_0}, \widehat{\vec{k}_1}, \widehat{\Delta_1})$ s.t. the checks pass in the real world, but in the ideal world it fails. This requires either breaking the security of $\mathcal{F}_{\mathsf{adIPA}}$ or guessing the value of Δ_0 and hence breaking the security of $\mathcal{F}_{\mathsf{VOLE}}^p$.
- HYB_2: This is the same as HYB_1, except $\mathsf{Sim}_{\mathsf{aBV}}$ invokes $\mathcal{F}_{\mathsf{aBV}}(\vec{x}_0, \vec{m}_0, \vec{k}_1, \Delta_1)$ in *Pairwise Product Check for both Sessions* to obtain output $\mathsf{out}_{\mathsf{aBV}}$ and outputs False if $\mathsf{out}_{\mathsf{aBV}} = \mathsf{False}$. This is the ideal world execution of the protocol.

  Adversary $\mathcal{A}$ is able to distinguish between the two hybrids if $\mathcal{F}_{\mathsf{aBV}}$ returns False in HYB_2, whereas the check passed in HYB_1. $\mathcal{A}$ has to compute $\mathsf{degzero}_1$ and degone_1 s.t. the following holds:

$$
\begin{aligned}
(\vec{\delta}_0 \odot \vec{x}_1) \cdot \vec{m}_0 = \mathsf{prod}_0 &= \frac{\mathsf{chall}_0 - (\mathsf{degzero}_1 + \mathsf{degone}_1 \cdot \Delta_0)}{2\Delta_0} \\
&= \frac{\mathsf{chall}_0 - (\mathsf{degzero}_1 + \mathsf{degone}_1 \cdot \Delta_0)}{2\Delta_0} \\
&= \frac{\vec{\delta}_0 \cdot (\vec{t}_0 \odot (\vec{t}_0 - \Delta_0)) + k'_0 - (\mathsf{degzero}_1 + \mathsf{degone}_1 \cdot \Delta_0)}{2\Delta_0} \\
\implies \quad 2\Delta_0 \cdot (\vec{\delta}_0 \odot \vec{x}_1) \cdot \vec{m}_0 & \\
&= \vec{\delta}_0 \cdot (\vec{t}_0 \odot (\vec{t}_0 - \Delta_0)) + k'_0 - (\mathsf{degzero}_1 + \mathsf{degone}_1 \cdot \Delta_0) \\
&= \vec{\delta}_0 \cdot ((\vec{m}_0 + (\vec{x}_0 + \vec{x}_1)\Delta_0) \odot (\vec{m}_0 + (\vec{x}_0 + \vec{x}_1 - 1)\Delta_0)) \\
&\quad + k'_0 - (\mathsf{degzero}_1 + \mathsf{degone}_1 \cdot \Delta_0)
\end{aligned}
$$

$$\begin{aligned}\implies \quad & 2\Delta_0 \cdot (\vec{\delta_0} \odot \vec{x_1}) \cdot \vec{m_0} \\ & - \vec{\delta_0} \cdot ((\vec{m_0} + (\vec{x_0} + \vec{x_1})\Delta_0) \odot (\vec{m_0} + (\vec{x_0} + \vec{x_1} - 1)\Delta_0)) - k'_0 \\ & = -(\mathsf{degzero}_1 + \mathsf{degone}_1 \cdot \Delta_0)\end{aligned}$$

If $(x_0 + x_1) \neq \{0, 1\}$ then there will be a Δ_0^2 term. This term vanishes for one specific value of $\vec{\delta}$ (which occurs with $\frac{1}{|\mathbb{F}|}$ probability) or if $\mathcal{A}$ correctly guesses Δ_0 which occurs with negligible probability in the $\mathcal{F}_{\mathsf{VOLE}}^{p}$ model.

□

Inputs: $\mathsf{Sim}_{\mathsf{adIPA}}$ obtains $((\vec{x}_0, \vec{m}_0, \vec{k}_1, \Delta_1), (\vec{\delta}_0, \vec{\delta}_1, \mathsf{prod}_1))$ as the candidate inputs of $\mathcal{S}_1^*$. $\mathsf{Sim}_{\mathsf{adIPA}}$ calls $\mathsf{Sim}_{\mathsf{VOLE}}$ with inputs $(\mathsf{Get\text{-}Key}, \mathsf{sid}_0)$ and $(\mathsf{Get\text{-}Keysid}_1)$ to receive Δ_0 and Δ_1.

Protocol Details:

1. *Key Sampling for Session 0:*
   $\mathsf{Sim}_{\mathsf{adIPA}}$ samples ElGamal key pair $(\mathsf{ipk}_0, \mathsf{isk}_0)$ and publishes ipk_0.
2. *Key Sampling for Session 1:*
   $\mathcal{S}_1^*$ sends ipk_1 to $\mathsf{Sim}_{\mathsf{adIPA}}$.
3. *Committing Inputs for Session 0:*
   $\mathsf{Sim}_{\mathsf{adIPA}}$ samples $\{\xi_{0,i}\}_{i\in[\ell]} \leftarrow \mathbb{F}_p^\ell$ and encrypts it as $\mathsf{eprod}_{0,i} := \mathsf{EG.Enc}(\mathsf{ipk}_0, \xi_{0,i}, r_{0,i})$, stores the randomness as $\vec{r_0} := \{r_{0,i}\}_{i\in[\ell]}$, $\vec{\xi_0} := \{\xi_{0,i}\}_{i\in[\ell]}$ and sends $\vec{\mathsf{eprod}}_0 := \{\mathsf{eprod}_{0,i}\}_{i\in[\ell]}$ to $\mathcal{S}_1^*$.
4. *Committing Inputs for Session 1:*
   $\mathsf{Sim}_{\mathsf{adIPA}}$ receives $\vec{\mathsf{eprod}}_1 := \{\mathsf{eprod}_{1,i}\}_{i\in[\ell]}$ from $\mathcal{S}_1^*$.
5. *Pad Sampling for Session 0:*
   $\mathsf{Sim}_{\mathsf{adIPA}}$ and $\mathcal{S}_1^*$ call $\mathcal{F}_{\mathsf{VOLE}}^{p}$ with input $(\mathsf{extend}, \mathsf{sid}_1, 1)$ as sender and receiver to obtain (x'_1, m'_1) and k'_1, respectively, where $k'_1 = m'_1 + x'_1 \cdot \Delta_1$. $\mathsf{Sim}_{\mathsf{adIPA}}$ encrypts and sends $\mathsf{emask}_0 := \mathsf{EG.Enc}(\mathsf{ipk}_0, x'_1, r'_0)$ to $\mathcal{S}_1$.
6. *Pad Sampling for Session 1:*
   $\mathcal{S}_1^*$ and $\mathsf{Sim}_{\mathsf{adIPA}}$ call $\mathcal{F}_{\mathsf{VOLE}}^{p}$ with input $(\mathsf{extend}, \mathsf{sid}_0, 1)$ as sender and receiver to obtain (x'_0, m'_0) and k'_0, respectively, where $k'_0 = m'_0 + x'_0 \cdot \Delta_0$. $\mathcal{S}_1^*$ sends $\mathsf{emask}_1 := \mathsf{EG.Enc}(\mathsf{ipk}_1, x'_0, r'_1)$ to $\mathsf{Sim}_{\mathsf{adIPA}}$.
7. *Input Verification for Session 0:*
   $\mathsf{Sim}_{\mathsf{adIPA}}$ simulates the proof that it encrypted $\vec{\delta}_0 \odot \vec{x}_1$ in $\vec{\mathsf{eprod}}_0$:
   a. Servers jointly sample $\vec{\eta_0} \leftarrow \mathbb{F}_p^\ell$ for input verification.
   b. $\mathsf{Sim}_{\mathsf{adIPA}}$ sends $u_0 := \vec{\eta_0} \cdot (\vec{\delta_0} \odot \vec{k_1}) + m'_1$ and $\mathsf{rnd}_0 := \vec{\eta_0} \cdot \vec{r_0} + r'_0$.
   c. $\mathcal{S}_1^*$ performs its own adversarial algorithm.
8. *Input Verification for Session 1:*
   $\mathsf{Sim}_{\mathsf{adIPA}}$ runs the honest $\mathcal{S}_0$ algorithm.
9. *Product Verification for Session 0:*
   a. $\mathcal{S}_1^*$ sends eresp_1 to $\mathsf{Sim}_{\mathsf{adIPA}}$.
   b. If $\mathsf{EG.Dec}(\mathsf{isk}_0, \mathsf{eresp}_1) \neq \vec{\xi_0} \cdot \vec{m_0}$ then $\mathsf{Sim}_{\mathsf{adIPA}}$ outputs False.
   c. $\mathsf{Sim}_{\mathsf{adIPA}}$ invokes $\mathcal{F}_{\mathsf{adIPA}}$ with inputs $((\vec{x}_0, \vec{m}_0, \vec{k}_1, \Delta_1), (\vec{\delta}_0, \vec{\delta}_1, \mathsf{prod}_1))$ to obtain output $\mathsf{out} \in \{\mathsf{True/False}\}$. If $\mathsf{out} = \mathsf{False}$, then $\mathsf{Sim}_{\mathsf{adIPA}}$ outputs False.
10. *Product Verification for Session 1:*
    a. $\mathsf{Sim}_{\mathsf{adIPA}}$ sends $\mathsf{eresp}_1 := \mathsf{EG.Enc}(\mathsf{ipk}_1, \mathsf{prod}_1)$.
    b. $\mathcal{S}_1^*$ performs it own adversarial algorithm.
11. If $\mathsf{Sim}_{\mathsf{adIPA}}$ has not output False then it outputs True.

Figure 12: Simulator $\mathsf{Sim}_{\mathsf{adIPA}}$ (that simulates $\mathcal{S}_0$) for Π_{adIPA} against an adversary $\mathcal{A}$ (that corrupts $\mathcal{S}_1$).

6.3 Security Proof of Π_{adIPA}

Theorem 4. *Assume EG.Enc is an additively homomorphic IND-CPA encryption scheme. Assuming a PPT adversary $\mathcal{A}$ maliciously corrupting either $\mathcal{S}_0$ or $\mathcal{S}_1$, then Π_{adIPA} (Fig. 5) UC-securely implements $\mathcal{F}_{adIPA}$ (Fig. 4) against $\mathcal{A}$ in the $\mathcal{F}_{VOLE}^{p}$ model.*

Proof. We assume that there is a PPT adversary $\mathcal{A}$ which maliciously corrupts $\mathcal{S}_1$ and we denote the corrupt server as $\mathcal{S}_1^*$. We provide a PPT simulator algorithm $\mathsf{Sim}_{\mathsf{adIPA}}$ in Fig. 12 for this corruption scenario where $\mathsf{Sim}_{\mathsf{adIPA}}$ simulates the honest $\mathcal{S}_0$ in the ideal world execution with access to $\mathcal{F}_{\mathsf{adIPA}}$. At the end of the execution, the ideal world adversary view is produced. We prove that Π_{adIPA} (Fig. 5) securely implements $\mathcal{F}_{\mathsf{adIPA}}$ (Fig. 4) by showing that the real and ideal world adversary views are indistinguishable.

The simulator algorithm $\mathsf{Sim}_{\mathsf{adIPA}}$ is called with the candidate inputs of a corrupt $\mathcal{S}_1^*$. The corrupt $\mathcal{S}_1^*$ can decide to use other inputs since it is maliciously secure. But those inputs are committed via VOLE, and our simulator $\mathsf{Sim}_{\mathsf{adIPA}}$ checks (except with negligible statistical probability) that the same inputs are used in the ideal world protocol, and the simulator invokes the $\mathcal{F}_{\mathsf{adIPA}}$ with the same inputs. This way, the simulator $\mathsf{Sim}_{\mathsf{adIPA}}$ forces the malicious adversarial server $\mathcal{S}_1^*$ to follow the protocol steps using the committed inputs that are initially provided to it from the calling party (which is the environment in the UC setting and $\mathsf{Sim}_{\mathsf{aBV}}$ in our setting), as the honest server has the corresponding VOLE decommitments to check that the same inputs are used by $\mathcal{S}_1^*$. So, we enforce semi-honest behavior from the adversarial $\mathcal{S}_1^*$ in following the protocol steps and also running the protocol with the inputs provided to it.

The corruption of $\mathcal{S}_0$ can be argued similarly, where $\mathsf{Sim}_{\mathsf{adIPA}}$ would simulate $\mathcal{S}_1$ (instead of $\mathcal{S}_0$) running the same protocol steps. We argue indistinguishability between the real and ideal world execution as follows:

- HYB_0: This is the real-world execution of the protocol Π_{adIPA}.
- HYB_1: This is the same as HYB_0, except $\mathsf{Sim}_{\mathsf{adIPA}}$ sets $\mathsf{eresp}_1 \coloneqq \mathsf{EG.Enc}(\mathsf{ipk}_1, \mathsf{prod}_1)$ in *Product Verification for Session 1.*

  Adversary $\mathcal{A}$ distinguishes between the two hybrids if $\mathcal{S}_1^*$ did not encrypt $\vec{\delta}_1 \odot \vec{x}_0$ in $\vec{\mathsf{eprod}}_1$ and yet passed the checks in *Input verification for Session 1.* In that case, the distribution of eresp_1 is an encryption of prod_1 in HYB_1, whereas in HYB_0 it is not. $\mathcal{S}_1^*$ possessing the corresponding secret key isk_1 can distinguish between the two hybrids. However, we show that this occurs with negligible statistical probability (specifically $1/|\mathbb{F}|$) in the $\mathcal{F}_{\mathsf{VOLE}}^p$ model. Assume that $\mathcal{S}_1^*$ encrypted $\vec{\delta}_1 \odot \vec{x}_0 + \vec{\mathsf{err}}$ in $\vec{\mathsf{eprod}}_1$ where $\vec{\mathsf{err}}$ is adversarially chosen error. To pass the check $e_0 = \mathsf{EG.Enc}(\mathsf{ipk}_1, (w_0 - u_1) \cdot \Delta_0^{-1}, \mathsf{rnd}_1)$, where $e_0 \coloneqq \mathsf{emask}_1 \cdot \prod_{i \in [\ell]} (\mathsf{eprod}_{1,i})^{\eta_{1,i}}$ and $w_0 \coloneqq \vec{\eta}_1 \cdot (\vec{\delta}_1 \odot \vec{k}_0) + k'_0$, the following should hold:

$$\mathsf{emask}_1 \cdot \prod_{i \in [\ell]} (\mathsf{eprod}_{1,i})^{\eta_{1,i}} = \mathsf{EG.Enc}(\mathsf{ipk}_1, (w_0 - u_1^*) \cdot \Delta_0^{-1}, \mathsf{rnd}_1).$$

  Since ElGamal Encryption is perfectly correct, the adversary needs to pass the check over the message and randomness space separately. We focus on the message space. Here, the adversary ensures that the following holds:

$$x'_0 + (\vec{\delta}_1 \odot \vec{x}_0) \cdot \vec{\eta_{1,i}} + \vec{\mathsf{err}} \cdot \vec{\eta_{1,i}} = (w_0 - u_1^*) \cdot \Delta_0^{-1}.$$

  To ensure that the above equation holds $\mathcal{A}$ needs to provide u_1^* that satisfies:

$$u_1^* = w_0 - (x'_0 + (\vec{\delta}_1 \odot \vec{x}_0) \cdot \vec{\eta_{1,i}}) \cdot \Delta_0 + \vec{\mathsf{err}} \cdot \vec{\eta_{1,i}} \cdot \Delta_0.$$

  To produce such a u_1^* the adversary either needs to figure out Δ_0 or $\vec{\mathsf{err}} \cdot \vec{\eta_{1,i}} = 0$. The former does not occur in the $\mathcal{F}_{\mathsf{VOLE}}^p$ model and the latter occurs with $\frac{1}{|\mathbb{F}|}$ probability since $\vec{\eta_{1,i}}$ is uniformly and independently sampled by both servers once $\vec{\mathsf{err}}$ is fixed in $\vec{\mathsf{eprod}}_1$.
- HYB_2: This is the same as HYB_1, except in *Product Verification for Session 0*, $\mathsf{Sim}_{\mathsf{adIPA}}$ invokes $\mathcal{F}_{\mathsf{adIPA}}$ with inputs $((\vec{x}_0, \vec{m}_0, \vec{k}_1, \Delta_1), (\vec{\delta}_0, \vec{\delta}_1, \mathsf{prod}_1))$ to obtain output $\mathsf{out} \in \{\mathsf{True}/\mathsf{False}\}$ and if $\mathsf{out} = \mathsf{False}$, $\mathsf{Sim}_{\mathsf{adIPA}}$ outputs False.

  The two hybrids are distinguishable only when $\mathcal{F}_{\mathsf{adIPA}}$ returns $\mathsf{out} = \mathsf{False}$ in HYB_3 causing $\mathsf{Sim}_{\mathsf{adIPA}}$ to output False. And in both hybrids, $\mathsf{EG.Dec}(\mathsf{isk}_0, \mathsf{eresp}_1^{-1} \cdot \mathsf{EG.Enc}(\mathsf{ipk}_0, \mathsf{prod}_0, 0)) = 0$ leading to $\mathsf{Sim}_{\mathsf{adIPA}}$ outputting True in HYB_2. $\mathcal{F}_{\mathsf{adIPA}}$ outputs False whereas HYB_1 does not when the $\mathcal{A}$ encrypted a different input $\vec{\delta}_1 \odot \vec{x}_0$

in the protocol and yet passed the check in *Input Verification 1*. However, as shown in the previous indistinguishability argument (between HYB_0 and HYB_1), this occurs with negligible probability and hence HYB_1 and HYB_2 are identically distributed.

- HYB_3: This is the same as HYB_1, except $\mathsf{Sim}_{\mathsf{adIPA}}$ sets $\mathsf{eprod}_{0,i} := \mathsf{EG.Enc}(\mathsf{ipk}_0, \xi_{0,i}, r_{0,i})$ in *Committing Inputs for Session 0* and $u_0 := \vec{\eta_0} \cdot (\vec{\delta_0} \odot \vec{k_1}) + m'_1$ in *Input Verification for Session 0*. $\mathsf{Sim}_{\mathsf{adIPA}}$ outputs False if $\mathsf{EG.Dec}(\mathsf{isk}_0, \mathsf{eresp}_1) \neq \vec{\xi_0} \cdot \vec{x_0}$ then $\mathsf{Sim}_{\mathsf{adIPA}}$ outputs False. This is the ideal world execution of the protocol.

  The two hybrids are indistinguishable from the IND-CPA security of ElGamal Encryption in the $\mathcal{F}^p_{\mathsf{VOLE}}$ model. If $\mathcal{A}$ distinguishes the two hybrids with advantage ϵ_{12} then the CPA adversary breaks IND-CPA security with ϵ_{CPA} s.t. $\epsilon_{12} \leq \ell \cdot \epsilon_{\mathsf{CPA}}$ since there are ℓ encryptions. Another way for the adversary to distinguish is if it computes $\mathsf{eresp}_1 \neq \left(\prod_{i\in[\ell]} (\mathsf{eprod}_{0,i})^{m_{0,i}}\right) \cdot \mathsf{EG.Enc}(\mathsf{ipk}_0, 0)$ and yet passes the check $\mathsf{EG.Dec}(\mathsf{isk}_0, \mathsf{eresp}_1) == \vec{\xi_0} \cdot \vec{m_0}$, resulting in the ideal world adversary $\mathsf{Sim}_{\mathsf{adIPA}}$ to output True, whereas the real world execution results in $\mathcal{S}_0$ outputting False as $\mathcal{S}_1^*$ did not use the candidate inputs $m_{0,i}$. However, this happens with negligible probability as the adversary has to guess the secret values $\vec{\xi_0}$ (which are encrypted in $\vec{\mathsf{eprod}_0}$) s.t. $\vec{\xi_0} \cdot \vec{m_0} == \vec{\xi_0} \cdot \vec{\widehat{m_0}}$, where adversary uses $\vec{\widehat{m_0}}$ instead of $\vec{m_0}$ in computing eresp_1.

□

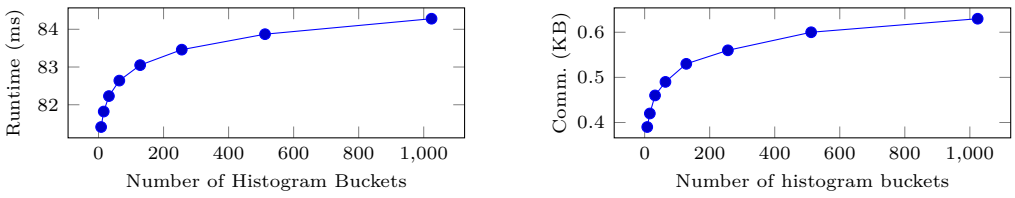
7 Performance Evaluation

We evaluate the runtime, communication, and storage costs of Paras. We first describe our estimation methodology for each of these three performance metrics and then elaborate on the results. All our experiments are fully automated and reproducible (we will open-source our code after the blind review).

Methodology. First, we identified all primitive operations involved across the protocols presented in this paper; notably, EC scalar-point multiplication, EC point-point addition, AES evaluation, and Vector OLE. The two EC operations are utilized within the EC ElGamal cryptosystem for encryption, decryption, and linear homomorphic computations. The AES cipher serves as the length-doubling PRG required by the VDPF scheme implemented in our protocol, and the VOLE functionality is employed to instantiate our MAC-based commitments.

Estimating the communication cost of Paras requires identifying every message exchange between parties and summing the sizes of all messages. For communication costs, we distinguish between client-to-server and server-to-server communication, rather than combining them into a single estimate. This distinction is meaningful because these two types of communication are likely to occur over networks with very different characteristics.

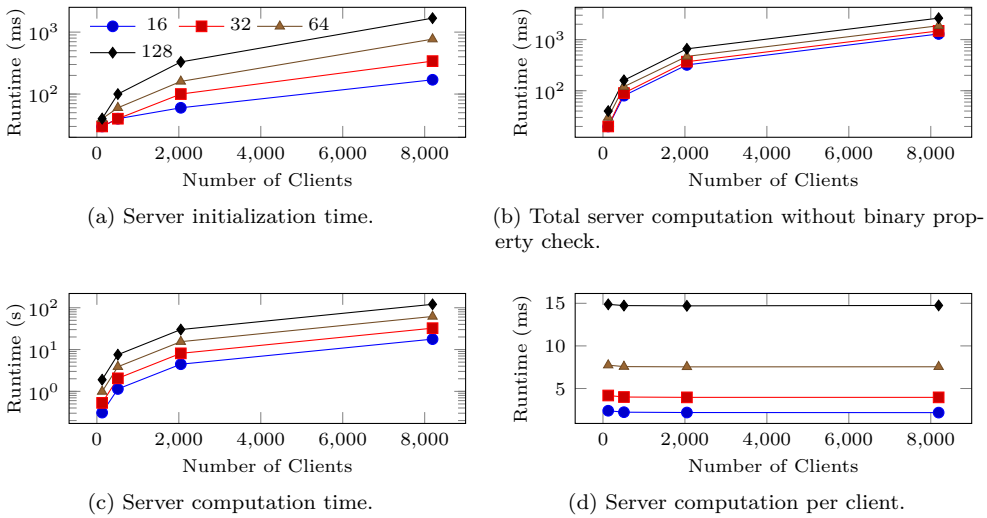
Computation Microbenchmarks. After identifying these primitive operations, we run microbenchmarks to measure their individual runtimes. Each microbenchmark executes



(a) Single client runtime.

(b) Single client communication cost.

Figure 13: Client runtime and communication for an increasing number of buckets.



(a) Server initialization time.

(b) Total server computation without binary property check.

(c) Server computation time.

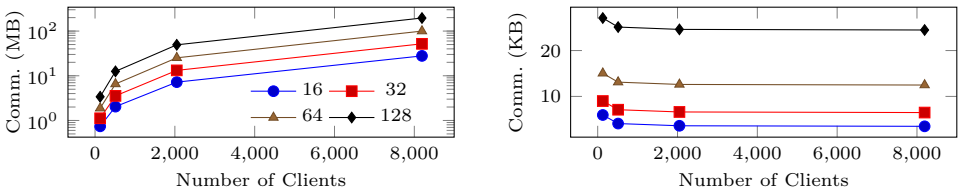
(d) Server computation per client.

Figure 14: Server runtime for different numbers of buckets (16, 32, 64, 128) and different numbers of clients.

the operation 10^3 times and reports the mean. The EC operation benchmarks were implemented using the `curve25519-dalek` Rust library [Cry25], the AES benchmark was implemented with OpenSSL [Inc25], and the VOLE benchmark was implemented using the libOTe [PR]. All benchmarks were executed on a server equipped with an AMD EPYC 74F3 processor (3.2 GHz base clock, up to 4.0 GHz boost) and 256 GB of RAM. In this environment, we observed mean running times of 8.68ms for scalar-point multiplication, 0.83μs for point-point addition, 0.41μs for AES cipher evaluation, and 183.87ms for VOLE when sampling vectors of size 2^{20}. With these measurements, we revisited our protocol, counted the total occurrences of the primitive identified operations, and summed them up to get an overall runtime of Paras. To ensure that this estimation is sound and reproducible, we automated the entire process using a Python script. Given the mean running times of the primitive operations, along with the number of clients and histogram buckets, the script automatically computes the total runtime. Please note that our runtime benchmarks assume every operation happens sequentially, that is, all in a single CPU core. We leave benchmarks that take into account the parts of the protocol that can be parallelized, such as the VDPF evaluations and verifications, as future work.

Communication Microbenchmarks. Similar to our runtime benchmarks, we automated the calculation of communication costs, yielding the same benefits of auditability and reproducibility. The script takes as input the number of clients and histogram buckets, and outputs estimates of client-to-server and server-to-server communication for a single protocol execution with the specified parameters. It assumes that EC points exchanged between parties are represented using 256 bits, and that both the hash function outputs and PRG seeds used in the protocol are 128 bits long.

Storage Microbenchmarks. To compute storage costs, we accounted for all data from the server initialization phase that must be retained until the end, namely, the VOLE correlations, as well as all messages sent by clients at the end of their input phase. We calculated the size of these VOLE correlations and added them to the total amount of data sent by clients to the servers. The client data contribution is computed using the same process as in the client-to-server communication cost estimation. Following the approach used for the other metrics, we automated the storage cost calculations, using parameters essentially identical to those required by the script for communication cost estimation.



(a) Total $\mathcal{S}$-to-$\mathcal{S}$ communication.

(b) Total $\mathcal{S}$-to-$\mathcal{S}$ comm. per client.

Figure 15: Total server-to-server ($\mathcal{S}$-to-$\mathcal{S}$) communication costs for an increasing number of clients and histogram buckets (16, 32, 64, 128).

Concrete Costs. We demonstrate runtime, communication, and storage costs for a single client, as well as the Paras servers. We separate the graphs as, in many cases, clients and servers run on machines with substantially different processing capabilities, and client-to-server and server-to-server communication often occur over networks with different characteristics.

Fig. 13 (a) shows the time it takes for a single client to execute its input phase with respect to the number of histogram buckets, and Fig. 13 (b) shows the aggregate size of the two messages produced as the result of a single client's input phase. Both graphs scale logarithmically, a behavior directly induced by the VDPF scheme underlying our protocol.

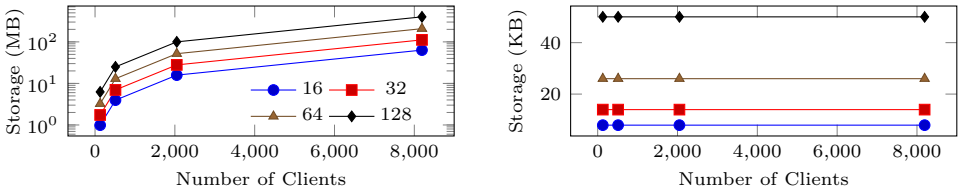
In more detail, observing the protocol's input phase, all computation is done with a constant amount of operations, aside from the VDPF key generation algorithm, which by itself has runtime $\mathcal{O}(\log n)$, where n is the number of histogram buckets. As a result, we have a client runtime that scales logarithmically with the number of histogram buckets, which we also verified in Fig. 13. Similarly, we see that the messages sent by the clients at the end of the input phase are also of constant size, except for the VDPF keys, which have $\mathcal{O}(\log n)$ size, where n again is the number of histogram buckets. This translates into a client input message size that scales logarithmically with the number of buckets.

Regarding the servers' performance, Fig. 14 demonstrates the running time estimates for the server computation, while Fig. 15 shows the amount of data exchanged between the two servers. As expected, both graphs scale according to the protocol's asymptotic cost of $\mathcal{O}(\mathsf{N} \cdot |\mathbf{X}|)$. First, observe that if we fix the histogram bucket count, server computation time grows linearly with the number of clients, which can be seen in the three graphs (b), (c), and (d) of Fig. 14. This becomes especially clear when observing graph (d), which plots the total server computation time per client, and shows all curves as horizontal lines. Secondly, we also call attention to the graphs (a), (b), and (c) of Fig. 14, which together show that the binary property check dominates the server computation time, and that the total protocol runtime is relatively small save for this check. We would also like to

With respect to the communication between the two servers during the protocol, the two graphs in Fig. 15 show, similarly to the runtime, that the amortized communication cost over multiple executions between the same two servers scales linearly to the number of clients when the bucket count is fixed. Additionally, we observe from Fig. 15 (b) that the communication cost per client is relatively low, with the cost per client being between 10 and 15 KB when the number of histogram buckets is fixed to 64.

Finally, we focus on the storage cost of Paras in Fig. 16. Similar to the two previous metrics, the storage scales linearly with respect to the number of clients for a fixed bucket count. Additionally, Fig. 16 (b) shows the communication for different histogram bucket counts. For example, for a histogram with 64 buckets, the storage per client is minimal (i.e., between 20 and 30 KB).

Comparison with Prior Work. Table 2 compares Paras with PLASMA and the sorting-based protocol [AHI+22], the closest prior works in terms of functionality and security guarantees. Unlike earlier two-server protocols such as Poplar, Doplar, and



(a) Total server storage.

(b) Total server storage per client.

Figure 16: Total server storage cost for an increasing number of clients and histogram buckets (16, 32, 64, 128).

Mastic, which do not guarantee correctness against a malicious server, PLASMA and [AHI+22] are the only previous protocols that provide robustness against both malicious clients and a malicious server for private histogram-style analytics, albeit in the three-server honest-majority setting. Therefore, they represent the most meaningful baseline for comparison.

We emphasize two structural differences that make the numbers in Table 2 illustrative rather than a direct benchmark comparison. First, PLASMA [MST24] and [AHI+22] operate in the three-server honest-majority setting and report measurements from a complete implementation, whereas Paras targets the two-server dishonest-majority setting and reports analytical estimates derived from the measured costs of its underlying primitives

Table 2: Comparison between Paras, PLASMA, and the sorting-based protocol [AHI+22]. PLASMA reports measurements from a complete implementation using 10^6 clients over 256-bit heavy hitters in a WAN setting, whereas [AHI+22] has LAN numbers for up to 400k clients. Paras reports performance estimates for histogram computation with 8192 clients over a domain of size $|\mathbf{X}| = 128$. Consequently, the reported numbers correspond to different workloads and should not be interpreted as direct benchmark comparisons.

Metric	**PLASMA [MST24]**	**Sorting [AHI+22]**	**Paras (This Work)**
Number of Servers	3	3	2
Security Model*	Honest Majority	Honest Majority	Dishonest Majority
Correctness & Privacy against Malicious Server & Clients	✓	✓	✓
Client Upload	8 VIDPF keys	3 secret shares + 3 MACs	2 VDPF keys + 2 ElGamal CTs
Consistency Mechanism	VIDPF + Merkle trees	Authenticated secret sharing + shuffling	VOLE + aBV + adIPA
Client Runtime	$20 - 24\mu$s	$\approx 1\mu$s	81 ms†
Client Communication	≈55 KB	≈0.14 KB	0.42 KB†
Server Runtime (amortized)	$\leq$ 0.3 ms ($\leq$ 5 min. for 10^6 clients)	$\leq$ 0.03 ms ($\leq$ 15 sec. 400k clients)	14 ms†
Server Com. (amortized)	$\leq$ 1 KB ($\leq$ 1 GB for 10^6 clients)	$>$ 45 KB ($>$ 45 GB for 10^6 clients)	24 KB†

* PLASMA and the sorting-based protocol [AHI+22] tolerate one malicious server among three non-colluding servers; Paras tolerates a malicious server in a two-party setting with no honest-majority fallback.

† Paras reports analytical performance estimates for histogram computation based on the measured costs of the underlying cryptographic primitives, excluding network latency. PLASMA reports measurements from a complete implementation for heavy-hitters, including WAN network latency. [AHI+22] reports numbers from LAN but is not open-source.

rather than an end-to-end implementation. We reproduce PLASMA's and [AHI+22]'s numbers directly from [MST24], in which the authors do an extensive comparison between the two. Second, PLASMA's and [AHI+22]'s numbers correspond to *heavy-hitters* over an exponentially large domain encoded as $n = 256$-bit strings, evaluated over a 256-level prefix tree with threshold-based pruning, whereas Paras targets *histogram* computation over a domain of size $|\mathbf{X}| = 128$, evaluated over a $\log_2 |\mathbf{X}| = 7$-level tree with no pruning search. Both factors account for most of the gap between the two columns.

The rest of the overhead, after accounting for the different settings above, comes from each protocol's underlying cryptographic assumptions. PLASMA relies entirely on PRG evaluations and hashing, whose cost is negligible (0.41μs per AES call); this suffices because their honest-majority assumption lets a malicious server be caught by cross-checking three sessions. Similarly, [AHI+22] relies on authenticated secret sharing with shuffling and unshuffling primitives, which outperforms PLASMA but incurs significantly more server-to-server communication overhead. Lastly, Paras relies on EC ElGamal encryption for verifiable homomorphic aggregation in the two-server dishonest-majority setting, where a single EC scalar-point multiplication costs 8.68ms in our benchmarks, roughly 360× PLASMA's entire reported client runtime. In short, Table 2's overhead is the price of replacing an honest-majority, three-party, and symmetric-key-only protocol with a two-party, public-key-based protocol that tolerates a malicious dishonest majority.

Acknowledgments

The second and fourth authors were supported by NSF award #2451972, and ARPA-H SP4701-23-C-0074.

References

[ACD+24] Erik Anderson, Melissa Chase, F. Betül Durak, Kim Laine, and Chenkai Weng. Precio: Private aggregate measurement via oblivious shuffling. In Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie, editors, *ACM CCS 2024: 31st Conference on Computer and Communications Security*, pages 1819–1833. ACM Press, October 2024. doi:10.1145/3658644.3670280.

[AHI+22] Gilad Asharov, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Ariel Nof, Benny Pinkas, Katsumi Takahashi, and Junichi Tomida. Efficient secure three-party sorting with applications to data analysis and heavy hitters. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022: 29th Conference on Computer and Communications Security*, pages 125–138. ACM Press, November 2022. doi:10.1145/3548606.3560691.

[Akb23] Ertuğrul Akbaş. Enhancing incident response with live logs: The significance and challenges of maintaining sufficient log retention for mitigating cyber attacks. In *International Conference on Cyber Security (ICCYS-23)*, volume 10, 2023.

[BBC+21] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Lightweight techniques for private heavy hitters. In *2021 IEEE Symposium on Security and Privacy*, pages 762–776. IEEE Computer Society Press, May 2021. doi:10.1109/SP40001.2021.00048.

[BBCC+25] Borja Balle, James Bell-Clark, Albert Cheu, Adrià Gascón, Jonathan Katz, Mariana Raykova, Phillipp Schoppmann, and Thomas Steinke. Hash-prune-

invert: Improved differentially private heavy-hitter detection in the two-server model. In Marina Blanton, William Enck, and Cristina Nita-Rotaru, editors, *2025 IEEE Symposium on Security and Privacy*, pages 2903–2918. IEEE Computer Society Press, May 2025. doi:10.1109/SP61157.2025.00186.

[BBG+20] James Henry Bell, Kallista A. Bonawitz, Adrià Gascón, Tancrède Lepoint, and Mariana Raykova. Secure single-server aggregation with (poly)logarithmic overhead. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020: 27th Conference on Computer and Communications Security*, pages 1253–1269. ACM Press, November 2020. doi:10.1145/3372297.3417885.

[BGG+22] James Bell, Adrià Gascón, Badih Ghazi, Ravi Kumar, Pasin Manurangsi, Mariana Raykova, and Phillipp Schoppmann. Distributed, private, sparse histograms in the two-server model. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022: 29th Conference on Computer and Communications Security*, pages 307–321. ACM Press, November 2022. doi:10.1145/3548606.3559383.

[BGI15] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part II*, volume 9057 of *Lecture Notes in Computer Science*, pages 337–367. Springer, Berlin, Heidelberg, April 2015. doi:10.1007/978-3-662-46803-6_12.

[BGI16] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, pages 1292–1303. ACM Press, October 2016. doi:10.1145/2976749.2978429.

[BGL+23] James Bell, Adrià Gascón, Tancrède Lepoint, Baiyu Li, Sarah Meiklejohn, Mariana Raykova, and Cathie Yun. ACORN: Input validation for secure aggregation. In Joseph A. Calandrino and Carmela Troncoso, editors, *USENIX Security 2023: 32nd USENIX Security Symposium*, pages 4805–4822. USENIX Association, August 2023. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/bell.

[BGR+24] Lennart Braun, Adrià Gascón, Mariana Raykova, Phillipp Schoppmann, and Karn Seth. Malicious security for sparse private histograms. Cryptology ePrint Archive, Report 2024/469, 2024. URL: https://eprint.iacr.org/2024/469.

[BIK+17] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for privacy-preserving machine learning. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 1175–1191. ACM Press, October / November 2017. doi:10.1145/3133956.3133982.

[BK21] Jonas Böhler and Florian Kerschbaum. Secure multi-party computation of differentially private heavy hitters. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021: 28th Conference on Computer and Communications Security*, pages 2361–2377. ACM Press, November 2021. doi:10.1145/3460120.3484557.

[BKM+20] Prasad Buddhavarapu, Andrew Knox, Payman Mohassel, Shubho Sengupta, Erik Taubeneck, and Vlad Vlaskin. Private matching for compute. Cryptology ePrint Archive, Report 2020/599, 2020. URL: https://eprint.iacr.org/2020/599.

[BS15] Raef Bassily and Adam D. Smith. Local, private, efficient protocols for succinct histograms. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *47th Annual ACM Symposium on Theory of Computing*, pages 127–135. ACM Press, June 2015. doi:10.1145/2746539.2746632.

[CB22] Graham Cormode and Akash Bharadwaj. Sample-and-threshold differential privacy: Histograms and applications. In *International Conference on Artificial Intelligence and Statistics*, pages 1420–1431. PMLR, 2022.

[CD97] Ronald Cramer and Ivan Damgård. Linear zero-knowledge - a note on efficient zero-knowledge proofs and arguments. In *29th Annual ACM Symposium on Theory of Computing*, pages 436–445. ACM Press, May 1997. doi:10.1145/258533.258635.

[CG24] Clément L Canonne and Abigail Gentle. Locally private histograms in all privacy regimes. *arXiv preprint arXiv:2408.04888*, 2024.

[CGB17] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, page 259–282, USA, 2017. USENIX Association.

[CGJvdM22] Amrita Roy Chowdhury, Chuan Guo, Somesh Jha, and Laurens van der Maaten. EIFFeL: Ensuring integrity for federated learning. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022: 29th Conference on Computer and Communications Security*, pages 2535–2549. ACM Press, November 2022. doi:10.1145/3548606.3560611.

[CGS97] Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. A secure and optimally efficient multi-authority election scheme. In Walter Fumy, editor, *Advances in Cryptology – EUROCRYPT'97*, volume 1233 of *Lecture Notes in Computer Science*, pages 103–118. Springer, Berlin, Heidelberg, May 1997. doi:10.1007/3-540-69053-0_9.

[CM85] William S Cleveland and Robert McGill. Graphical perception and graphical methods for analyzing scientific data. *Science*, 229(4716):828–833, 1985.

[CP93] David Chaum and Torben P. Pedersen. Wallet databases with observers. In Ernest F. Brickell, editor, *Advances in Cryptology – CRYPTO'92*, volume 740 of *Lecture Notes in Computer Science*, pages 89–105. Springer, Berlin, Heidelberg, August 1993. doi:10.1007/3-540-48071-4_7.

[Cry25] Dalek Cryptography. Dalek elliptic curve cryptography. https://github.com/dalek-cryptography/curve25519-dalek, 2025. Accessed: 2025-09-16.

[CSU+19] Albert Cheu, Adam D. Smith, Jonathan Ullman, David Zeber, and Maxim Zhilyaev. Distributed differential privacy via shuffling. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019, Part I*, volume 11476 of *Lecture Notes in Computer Science*, pages 375–403. Springer, Cham, May 2019. doi:10.1007/978-3-030-17653-2_13.

[CZ22] Albert Cheu and Maxim Zhilyaev. Differentially private histograms in the shuffle model from fake users. In *2022 IEEE Symposium on Security and Privacy*, pages 440–457. IEEE Computer Society Press, May 2022. doi:10.1109/SP46214.2022.9833614.

[dCP22] Leo de Castro and Antigoni Polychroniadou. Lightweight, maliciously secure verifiable function secret sharing. In Orr Dunkelman and Stefan Dziembowski, editors, *Advances in Cryptology – EUROCRYPT 2022, Part I*, volume 13275 of *Lecture Notes in Computer Science*, pages 150–179. Springer, Cham, May / June 2022. doi:10.1007/978-3-031-06944-4_6.

[DPRS23] Hannah Davis, Christopher Patton, Mike Rosulek, and Phillipp Schoppmann. Verifiable distributed aggregation functions. *Proceedings on Privacy Enhancing Technologies*, 2023(4):578–592, October 2023. doi:10.56553/popets-2023-0126.

[ElG84] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In G. R. Blakley and David Chaum, editors, *Advances in Cryptology – CRYPTO'84*, volume 196 of *Lecture Notes in Computer Science*, pages 10–18. Springer, Berlin, Heidelberg, August 1984. doi:10.1007/3-540-39568-7_2.

[FD81] David Freedman and Persi Diaconis. On the histogram as a density estimator: L 2 theory. *Zeitschrift für Wahrscheinlichkeitstheorie und verwandte Gebiete*, 57(4):453–476, 1981.

[FPE16] Giulia C. Fanti, Vasyl Pihur, and Úlfar Erlingsson. Building a RAPPOR with the unknown: Privacy-preserving learning of associations and data dictionaries. *Proceedings on Privacy Enhancing Technologies*, 2016(3):41–61, July 2016. doi:10.1515/popets-2016-0015.

[GI14] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*, volume 8441 of *Lecture Notes in Computer Science*, pages 640–658. Springer, Berlin, Heidelberg, May 2014. doi:10.1007/978-3-642-55220-5_35.

[GKK+22] Badih Ghazi, Ben Kreuter, Ravi Kumar, Pasin Manurangsi, Jiayu Peng, Evgeny Skvortsov, Yao Wang, and Craig Wright. Multiparty reach and frequency histogram: Private, secure, and practical. *Proceedings on Privacy Enhancing Technologies*, 2022(1):373–395, January 2022. doi:10.2478/popets-2022-0019.

[GKKM22] Badih Ghazi, Pritish Kamath, Ravi Kumar, and Pasin Manurangsi. Anonymized histograms in intermediate privacy models. *Advances in Neural Information Processing Systems*, 35:8456–8468, 2022.

[Inc25] OpenSSL Software Foundation Inc. Openssl. https://github.com/openssl/openssl, 2025. Accessed: 2025-09-16.

[JKK+24] Pranav Jangir, Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, Bhavish Raj Gopal, and Somya Sangal. Vogue: Faster computation of private heavy hitters. *IEEE Transactions on Dependable and Secure Computing*, 21(4):3096–3108, 2024. doi:10.1109/TDSC.2023.3322511.

[KB12] Vijay Katkar and SG Bhirud. Novel DoS/DDoS attack detection and signature generation. *International Journal of Computer Applications*, 47(10):18–24, 2012.

[KTM+21] Ryan Karl, Jonathan Takeshita, Alamin Mohammed, Aaron Striegel, and Taeho Jung. Cryptogram: Fast private calculations of histograms over multiple users' inputs. In *2021 17th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pages 25–34, 2021. doi:10.1109/DCOSS52077.2021.00017.

[LBV+23] Hidde Lycklama, Lukas Burkhalter, Alexander Viand, Nicolas Küchler, and Anwar Hithnawi. RoFL: Robustness of secure federated learning. In *2023 IEEE Symposium on Security and Privacy*, pages 453–476. IEEE Computer Society Press, May 2023. doi:10.1109/SP46215.2023.10179400.

[LPGC17] Fabrizio Lamberti, Gianluca Paravati, Valentina Gatteschi, and Alberto Cannavò. Supporting web analytics by aggregating user interaction data from heterogeneous devices using viewport-dom-based heat maps. *IEEE Transactions on Industrial Informatics*, 13(4):1989–1999, 2017. doi:10.1109/TII.2017.2658663.

[LPR+21] Tancrède Lepoint, Sarvar Patel, Mariana Raykova, Karn Seth, and Ni Trieu. Private join and compute from PIR with default. In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2021, Part II*, volume 13091 of *Lecture Notes in Computer Science*, pages 605–634. Springer, Cham, December 2021. doi:10.1007/978-3-030-92075-3_21.

[MMT+24] Dimitris Mouris, Daniel Masny, Ni Trieu, Shubho Sengupta, Prasad Buddhavarapu, and Benjamin M. Case. Delegated private matching for compute. *Proceedings on Privacy Enhancing Technologies*, 2024(2):49–72, April 2024. doi:10.56553/popets-2024-0040.

[MPD+25] Dimitris Mouris, Christopher Patton, Hannah Davis, Pratik Sarkar, and Nektarios Georgios Tsoutsos. Mastic: Private weighted heavy-hitters and attribute-based metrics. *Proceedings on Privacy Enhancing Technologies*, 2025(1):290–319, January 2025. doi:10.56553/popets-2025-0017.

[MST24] Dimitris Mouris, Pratik Sarkar, and Nektarios Georgios Tsoutsos. PLASMA: Private, lightweight aggregated statistics against malicious adversaries. *Proceedings on Privacy Enhancing Technologies*, 2024(3):4–24, July 2024. doi:10.56553/popets-2024-0064.

[Mun05] Barbara Hazard Munro. *Statistical methods for health care research*, volume 1. lippincott williams & wilkins, 2005.

[NLT25] Truong Son Nguyen, Tancrède Lepoint, and Ni Trieu. Mario: Multi-round multiple-aggregator secure aggregation with robustness against malicious actors. In *2024 IEEE European Symposium on Security and Privacy*, pages 1140–1164. IEEE Computer Society Press, June / July 2025. doi:10.1109/EuroSP63326.2025.00069.

[Ots79] Nobuyuki Otsu. A threshold selection method from gray-level histograms. *IEEE Transactions on Systems, Man, and Cybernetics*, 9(1):62–66, 1979. doi:10.1109/TSMC.1979.4310076.

[PR] Lance Roy Peter Rindal. libOTe: an efficient, portable, and easy to use Oblivious Transfer Library. https://github.com/osu-crypto/libOTe. Accessed: 2025-09-16.

[PYB+06] Bernice R. Packer, Meredith Yeager, Laura Burdett, Robert Welch, Michael Beerman, Liqun Qi, Hugues Sicotte, Brian Staats, Mekhala Acharya, Andrew Crenshaw, Andrew Eckert, Vinita Puri, Daniela S. Gerhard, and Stephen J. Chanock. Snp500cancer: a public resource for sequence validation, assay development, and frequency analysis for genetic variation in candidate genes. *Nucleic Acids Research*, 34(suppl_1):D617–D621, 01 2006. doi:10.1093/nar/gkj151.

[RSWP23] Mayank Rathee, Conghao Shen, Sameer Wagh, and Raluca Ada Popa. ELSA: Secure aggregation for federated learning with malicious actors. In *2023 IEEE Symposium on Security and Privacy*, pages 1961–1979. IEEE Computer Society Press, May 2023. doi:10.1109/SP46215.2023.10179468.

[SMX+25] Manuel B. Santos, Dimitris Mouris, Xiang Xie, Miguel de Vega, and Andrei Lapets. TLShare: Private authenticated MPC and FHE inputs over TLS. Cryptology ePrint Archive, Report 2025/1434, 2025. URL: https://eprint.iacr.org/2025/1434.

[SNKG23] Vinay Singh, Brijesh Nanavati, Arpan Kumar Kar, and Agam Gupta. How to maximize clicks for display advertisement in digital marketing? a reinforcement learning approach. *Information Systems Frontiers*, 25(4):1621–1638, 2023. doi:10.1007/s10796-022-10314-0.

[SSG+25] Ali Shahin Shamsabadi, Peter Snyder, Ralph Giles, Aurélien Bellet, and Hamed Haddadi. Nebula: Efficient, private and accurate histogram estimation. In Chun-Ying Huang, Jyh-Cheng Chen, Shiuh-Pyng Shieh, David Lie, and Véronique Cortier, editors, *ACM CCS 2025: 32nd Conference on Computer and Communications Security*, pages 498–512. ACM Press, October 2025. doi:10.1145/3719027.3744789.

[WHT+15] Shaowei Wang, Liusheng Huang, Miaomiao Tian, Wei Yang, Hongli Xu, and Hansong Guo. Personalized privacy-preserving data aggregation for histogram estimation. In *2015 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE, 2015. doi:10.1109/GLOCOM.2015.7417364.

[XZX+13] Jia Xu, Zhenjie Zhang, Xiaokui Xiao, Yin Yang, Ge Yu, and Marianne Winslett. Differentially private histogram publication. *The VLDB journal*, 22(6):797–822, 2013. doi:10.1007/s00778-013-0309-y.

[YSWW21] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. QuickSilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021: 28th Conference on Computer and Communications Security*, pages 2986–3001. ACM Press, November 2021. doi:10.1145/3460120.3484556.

[YWL+20] Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. Ferret: Fast extension for correlated OT with small communication. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020: 27th Conference on Computer and Communications Security*, pages 1607–1626. ACM Press, November 2020. doi:10.1145/3372297.3417276.

[ZKL+10] Tao Zhou, Zoltán Kuscsik, Jian-Guo Liu, Matúš Medo, Joseph Rushton Wakeling, and Yi-Cheng Zhang. Solving the apparent diversity-accuracy dilemma of recommender systems. *Proceedings of the National Academy of Sciences*, 107(10):4511–4515, 2010. doi:10.1073/pnas.1000488107.