FastPSC: Fast and Maliciously Secure Computation of Intersection and Union on Multi-Owner Sets

ABSTRACT

Recently, significant research efforts have been devoted to developing specific protocols for private set intersection (PSI) and private set union (PSU). The primary focus of previous research lies in the multi-party setting, where set owners collaboratively execute PSI/PSU protocols on their sets. Limited research has investigated the more scalable outsourcing setting, where set owners secretly share their sets among a set of computing servers, which then perform PSI/PSU protocols on the secret-shared sets. In this paper, we present FastPSC, a new system supporting maliciously secure PSI/PSU in the outsourcing setting. FastPSC is built from a custom synergy of lightweight secure computation techniques and differential privacy. Our key insight is to leverage differentially private leakage to achieve a significant efficiency boost in secure and accurate computation of intersection/union in the outsourcing setting. Experiments show that with differentially private leakage allowed, FastPSC can achieve a significant performance advantage over the-state-of-the-art prior works without differentially private leakage. Specifically, compared to the work by Mohassel *et al.* (CCS'20) with semi-honest security, FastPSC achieves a speedup of $1.5\times - 52.2\times$ and reduces server-side communication cost by 78%-98%. In comparison to the work by Asharov *et al.* (CCS'23) with malicious security, FastPSC achieves a speedup of $4.2\times - 7.4\times$ and reduces server-side communication cost by 99%.

1 INTRODUCTION

Performing set operations on sets owned by different parties is crucial for collaborative analytics across private databases [20, 38]. However, in real-world scenarios, database owners often rely on a trusted third party to collect their private sets in plaintext for analysis. Such unconditional trust is worrisome and poses considerable security risks, as the trusted party becomes an attractive attack target and may be compromised [26]. Recent years have witnessed substantial research dedicated to the design of private set computation protocols, particularly with respect to private set intersection (PSI) and private set union (PSU). PSI and PSU are fundamental tools for secure collaborative analytics across private databases, underpinning numerous applications such as healthcare data federation analysis [28], database joins [27], contact discovery [20], online advertising effectiveness assessment [22], cyber risk assessment and management [23], and more. Overall PSI/PSU protocols aim to reveal only the result of set operations while minimizing any additional information leakage about individual input sets.

Existing solutions. In general, existing PSI/PSU protocols can be categorized into two types based on their system architecture: the *multi-party setting* and the *outsourcing setting*. Figure 1 illustrates their differences. In the literature, the majority of research [8, 17, 18, 23, 24, 33–36] has focused on the multi-party setting. In this setting, set owners cooperatively execute PSI/PSU protocols on their sets and obtain the plaintext intersection/union. Limited research

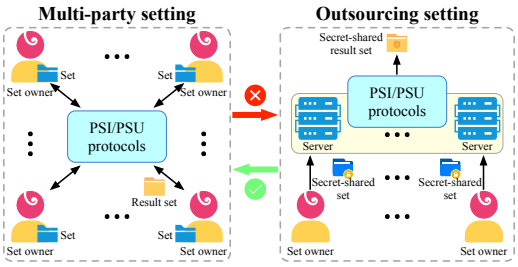

Figure 1: The differences in the system architecture of existing PSI/PSU protocols.

[3, 28, 31] has explored the outsourcing setting, where set owners secretly share their sets among a set of servers. These servers then execute PSI/PSU protocols on the secret-shared sets and return the secret-shared intersection/union to the set owners or a querier for recovering.

Note that the protocols in the multi-party setting cannot be readily adapted to the outsourcing setting, but the reverse adaptation is straightforward. This is because, in multi-party setting protocols, certain intermediate computations must be performed on plaintext input sets. In contrast, in the outsourcing setting, only the secret-shared sets are available to the computational parties (i.e., the servers). The protocols in the outsourcing setting can be adapted to the multi-party setting by having set owners secretly share their sets to other set owners, who then act as the servers to perform the PSI/PSU protocols on the secret-shared sets. Moreover, the protocols in the outsourcing setting presents a distinct advantage: allowing a large number of set owners to participate in PSI/PSU computation without the need for them to personally execute the intricate PSI/PSU protocols. Therefore, we design FastPSC in the more scalable outsourcing setting.

The state-of-the-art works in the outsourcing setting are [3, 31]. The work in [31] considers an honest-majority three-server setting. However, it only considers the semi-honest threat model, assuming that the servers honestly perform the protocols. Therefore, it cannot provide integrity guarantees for the intersection/union in the presence of malicious adversary. The work in [3] considers an honest-majority three (or more)-server setting and a stronger malicious threat model. In the threat model, the malicious server(s) could arbitrarily deviate from the protocols. The work in [3] provides security with abort upon the occurrence of deviation behaviors. However, a common limitation shared by both [3, 31] is their limited scalability as the number of set owners and the set size increase. For example, with 60 set owners, as the size of their sets increases from 2^{16} to 2^{24}, the running time of [31]'s PSI protocol increases from 12.8 seconds to 6661 seconds, and the running time of [3]'s PSI protocol increases from 37.6 seconds to 10046.4 seconds. Therefore, how to enable fast PSI/PSU with malicious security in the outsourcing setting remains to be fully explored.

Our results. We present FastPSC, a new system supporting fast and maliciously secure outsourced computation of intersection and union on multi-owner sets. FastPSC provides integrity guarantees for the intersection/union in the honest-majority four-server setting. Such four-server setting has been utilized in various recent studies [7, 9, 15]. FastPSC shows that allowing differentially private leakage makes it possible to achieve a significant performance improvement over the-state-of-the-art prior works [3, 31]. Specifically, in contrast to the work in [31] with semi-honest security, FastPSC not only achieves stronger malicious security but also achieves a speedup ranging from 1.5× to 52.2× and reduces server-side communication costs by 78%-98%; in contrast to the work in [3] with malicious security, FastPSC achieves a speedup ranging from 4.2× to 7.4× and reduces server-side communication costs by 99%.

1.1 Overview of Our Techniques

We observe that prior works [3, 31] keep the set elements of set owners and the set operation result fully confidential against the servers throughout the process of private set computation. Such extremely strong security guarantee, however, comes with high performance overhead which limits the scalability of these protocols for large-scale data analytics. Aiming for performant private set computation in the outsourcing setting, our key idea in designing FastPSC is to leverage *differentially private* leakage as a trade-off to achieve significant efficiency boost in secure and accurate computation of the intersection/union. Specifically, we depart from the prior works [3, 31] by allowing the servers to obtain differentially private leakage from the protocols for outsourced private set computation. To our best knowledge, such insight has not been explored before in the context of *outsourced* private set computation. Differential privacy (DP) [13] can ensure in our context that the adversary obtains similar leakage regardless of whether a particular set element of a set owner is involved in the computation or not.

In FastPSC, the underlying plaintext mechanism we follow to compute the intersection/union is histogram-based, as we observe that the intersection and union of sets can be deduced from the *histogram* of all the set elements across the set owners. Specifically, (1) if the count of an element equals the number of sets, it must be part of the intersection; otherwise, it is not[1]; (2) if the count of an element is greater than 0, it must be included in the union; otherwise, it is not. The high-level challenge in FastPSC is thus how to allow efficient, accurate, and maliciously-secure construction of a histogram for the set elements, while ensuring the servers only learn differentially private leakage. We develop custom techniques based on lightweight secret sharing and differential privacy to tackle this challenge, which we summarize below.

Building a secret-shared histogram with accuracy and pure DP guarantees: In order to accurately build a histogram for the set elements while preserving data privacy for set owners and facilitating efficient secure computation, we devise a custom mechanism to enable the servers to collect obfuscated set elements for histogram building with accuracy and pure DP guarantees. Our starting point is to add dummy elements into each set owner's set. Each dummy element is associated with a secret-shared flag of value 0, while each real element is associated with a secret-shared flag of value 1. The servers can observe the obfuscated sets of elements, and thus can easily aggregate the secret-shared flags associated with identical (real/dummy) elements. In this way, a histogram with secret-shared accurate counts for each involved element can be produced, which will be used later to generate the target set operation result. On another hand, as the servers can see the obfuscated sets of elements, there is a leakage about the input set elements to the servers and we aim to make such leakage differentially private. A critical challenge here is how to achieve *pure* DP for the leakage without actually removing any real elements so as to preserve result accuracy? Note that only adding dummy elements captures solely additive noise and fails to account for subtractive noise, which would only allow *approximate* DP. To address the challenge, we build on the recent notion of selective secure multi-party computation (selective MPC) [21] and develop a custom method which can simulate the removal of some real elements from each server's view without compromising the accuracy of the set operation results, with rigorous proofs on pure DP guarantees.

Securing the private histogram computation: To support secure secret-shared histogram generation, a natural choice is the additive secret sharing (ASS) [12] technique. Meanwhile, ASS can be used together with information-theoretic message authentication codes (MACs) [10] to achieve integrity against malicious adversary, through operating on secret-shared authenticated data. However, the attempt to combine ASS with MACs for use in our context is challenging. Simply having the set owners authenticate their private inputs with a MAC key unknown to the servers would require all set owners to share the same MAC key, posing an overly strong assumption that the malicious server must not collude with any set owner. To counter this, one might think of shifting the MAC creation from the set owners' side to the servers' side, by having the querier distribute a secret-shared MAC key to the servers for jointly authenticating the secret-shared inputs themselves. However, this is still not a working solution: a malicious server may tamper with the secret shares of the inputs as soon as it receives from the set owners. To address this problem, we instead choose to layer 2-out-of-3 replicated secret sharing (RSS) [1] with information-theoretic MACs, and develop secure protocols with computational integrity guarantees against the malicious adversary.

Controlling result release to the querier: After securely building the secret-shared histogram, the servers then need to enable the querier to learn the intersection or union. A plausible method is to directly return the secret-shared histogram to the querier. However, this will make the querier learn additional information beyond the intersection or union. For example, the querier can learn: (1) both the intersection and the union even if it only queries one of them; (2) the number of set owners possessing an element not in the intersection; (3) the number of set owners possessing an element in the union. This obviously violates the security requirements for PSI/PSU, i.e., the querier should only learn the intersection or union, without any additional information. To address this challenge, we design secure mechanisms to enable the servers to randomize the secret-shared count of each element in the histogram based on the set operation type. This guarantees that, from the querier's view, the elements in the intersection or union exhibit counts that are entirely distinct from those of elements not in the intersection or

[1] Similar to prior works [28, 31], this paper does not consider multisets.

union. In other words, based on the returned histogram, the querier only learns the intersection or union and nothing beyond that.

1.2 Justifications for DP Leakage

We emphasize that our DP guarantees should be understood as a meaningful improvement over the insecure status quo for intersection and union computation on multi-owner sets. They are not intended to replace standard PSI or PSU protocols, which offer stronger privacy guarantees but often suffer from limited scalability in our target settings. In particular, the leakage incurred by FastPSC corresponds to the differentially private element histogram computed over the *federated* sets contributed by all participating data owners. This form of leakage is considered acceptable in application scenarios where the primary privacy objective is to protect the input set of an individual data owner, rather than the overall distribution of elements across all sets. For example: (1) In healthcare data federation analysis, the patient cases held by each individual hospital are highly sensitive, whereas the aggregate case distribution across all hospitals is generally considered less sensitive and is often publicly reported in epidemiological studies and health system reports; (2) In contact discovery, each user's contact list is sensitive, whereas the overall contact distribution across all users is typically not a major privacy concern; (3) In online advertising effectiveness assessment, the list of engaged users for a particular company is sensitive, while the combined engagement data across multiple companies is often acceptable to expose in an aggregated and differentially private manner. Therefore, FastPSC serves as a well-performing and practical alternative to standard PSI and PSU protocols for applications that place less stringent privacy requirements on the global distribution of set elements.

2 RELATED WORK

Recently, a large amount of work has been done on specific PSI and PSU protocols. The majority of research (e.g., [8, 17, 18, 23, 24, 33, 34]) has focused on the multi-party setting. In this setting, set owners execute PSI or PSU protocols on their sets by themselves and some of them obtain the plaintext intersection or union. The recent works are mainly based on oblivious transfer [23, 34], oblivious key-value store [17, 33], and oblivious pseudorandom function [8, 18, 23, 33]. Most of the works [17, 18, 33] focus on the typical PSI or PSU problems. Other works delve into specialized PSI or PSU problems, including labeled PSI [8], PSI for small sets (500 elements or fewer), and PSU for unbalanced sets [23]. In addition, some works [18, 24, 35, 36] achieve more efficient PSI/PSU in the multi-party setting by allowing differentially private leakage.

Limited attention [3, 19, 28, 31] has been devoted to the more scalable outsourcing setting. The work [31] considers only the semi-honest threat model and fails to provide integrity guarantees for the intersection and union in the presence of malicious adversary. The work [19] claims to provide an integrity guarantee for the intersection under a malicious threat model; however, this guarantee relies on the blockchain, with the assumption that its nodes are semi-honest and will honestly execute the consensus protocols. While the work [28] provides integrity guarantees for the intersection and union under the malicious threat model, it has two serious limitations. Firstly, it requires all set owners to hold the same permutation to provide integrity guarantees while assuming that the malicious server will not collude with any set owners. The assumption is notably strong, particularly when the number of set owners is substantial. If the malicious server colludes with any owner, the integrity guarantees would be entirely compromised. Secondly, the party receiving the intersection can infer information beyond that, such as identifying elements that share the same number of owners. A common limitation of [3, 28, 31] is their limited scalability as the number of set owners and the set size increase. Additionally, [19, 28, 31] share a limitation in their loss of accuracy due to the use of hashing. Therefore, achieving fast and accurate PSI/PSU with malicious security in the outsourcing setting remains to be fully explored.

3 BUILDING BLOCKS

3.1 Differential Privacy

DP [13] can be formally defined as follows.

Definition 1. *A randomized mechanism $\mathcal{M}$ is said to be (ε, δ)-DP, if for any two neighboring datasets D_1 and D_2 that differ on a single element, we have: $\forall D' \in Range(\mathcal{M})$,*

$$Pr[\mathcal{M}(D_1) = D'] \leq e^{\varepsilon} \cdot Pr[\mathcal{M}(D_2) = D'] + \delta,$$

where $Range(\mathcal{M})$ denotes the set of all possible outputs of $\mathcal{M}$, ε is the privacy budget, and δ is privacy parameter.

Intuitively, when presented with the output D' of randomized mechanism $\mathcal{M}$, an adversary is unable to confidently infer (with the level of confidence controlled by ε) whether the input dataset is D_1 or its neighbor D_2, thereby providing plausible deniability for individuals associated with the sensitive dataset. δ represents the probability of DP failure. If $\delta \neq 0$, we say that $\mathcal{M}$ provides approximate DP; if $\delta = 0$, we say that $\mathcal{M}$ provides pure DP.

3.2 Secret Sharing

In this paper, we use two forms of secret sharing: 2-out-of-3 RSS [1] and n-out-of-n ($n = 3$ or 4) ASS [12].

Replicated secret sharing. Given a private value $x \in \mathbb{F}_p$, where $\mathbb{F}_p$ denotes the finite field modulo a prime number p, 2-out-of-3 RSS [1] divides it into three secret shares $\langle x \rangle_1$, $\langle x \rangle_2$, and $\langle x \rangle_3$. Here, $\langle x \rangle_1$ and $\langle x \rangle_2$ are two values randomly drawn from $\mathbb{F}_p$, and $\langle x \rangle_3 = x - \langle x \rangle_1 - \langle x \rangle_2 \mod p$. Subsequently, each pair $(\langle x \rangle_1, \langle x \rangle_2)$, $(\langle x \rangle_2, \langle x \rangle_3)$, and $(\langle x \rangle_3, \langle x \rangle_1)$ is distributed to three non-colluding parties P_1, P_2, and P_3, respectively. For clarity, when utilizing an index to represent the i-th secret share, $i - 1$ and $i + 1$ denote the preceding and succeeding secret share with wrap-around. By this way, the secret shares held by $\mathsf{P}_i, i \in \{1, 2, 3\}$ can be denoted as $(\langle x \rangle_i, \langle x \rangle_{i+1})$. We denote such RSS of x as $[\![x]\!]$.

The basic secure operations supported by the RSS are as follows. (1) Linear operations: Given two public constants w, v and two secret-shared values $[\![x]\!], [\![y]\!]$, to compute $[\![u]\!] = [\![w \cdot x + v \cdot y]\!]$, each $\mathsf{P}_i, i \in \{1, 2, 3\}$ only needs to locally calculate $\langle u \rangle_i = w \cdot \langle x \rangle_i + v \cdot \langle y \rangle_i$ and $\langle u \rangle_{i+1} = w \cdot \langle x \rangle_{i+1} + v \cdot \langle y \rangle_{i+1}$. (2) Multiplication operations: Given two secret-shared values $[\![x]\!], [\![y]\!]$, to compute $[\![z]\!] = [\![x \cdot y]\!]$, each $\mathsf{P}_i, i \in \{1, 2, 3\}$ first locally computes $\langle z \rangle_i = \langle x \rangle_i \cdot \langle y \rangle_i + \langle x \rangle_i \cdot \langle y \rangle_{i+1} + \langle x \rangle_{i+1} \cdot \langle y \rangle_i$. Then P_i re-shares $\langle z \rangle_i$ to obtain a RSS of the product z. We denote such secure multiplication as $[\![z]\!] = [\![x]\!] \cdot [\![y]\!]$.

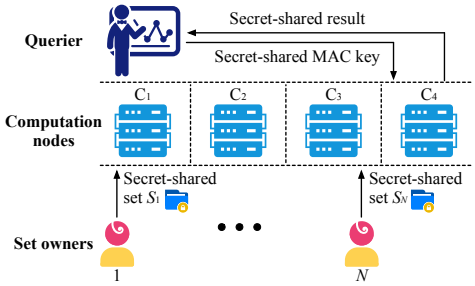

Figure 2: The system architecture of FastPSC.

Additive secret sharing. n-out-of-n ASS [12] divides a private value $x \in \mathbb{F}_p$ into n secret shares $\langle x\rangle_1, \langle x\rangle_2, \cdots, \langle x\rangle_n \in \mathbb{F}_p$, where $x = \langle x\rangle_1 + \langle x\rangle_2 + \cdots + \langle x\rangle_n \mod p$. Each share $\langle x\rangle_i$ is distributed to a party P_i. We denote such ASS of x as $[\![x]\!]^A$.

3.3 Information-Theoretic MACs

In the context of MPC, information-theoretic MACs [10] are extensively employed to achieve malicious security. Given a private value $x \in \mathbb{F}_p$, computational parties possess both the ciphertext of x and the ciphertext of its MAC $\hat{x} = \alpha \cdot x \mod p$, where $\alpha \in \mathbb{F}_p$ is a randomly generated MAC key unknown to the parties. During the secure computation phase, the secure operations are performed over both the (encrypted or secret-shared) values and their (encrypted or secret-shared) MACs. The protocol aborts if the output z and its MAC $\hat{z}$ do not satisfy $\hat{z} = \alpha \cdot z \mod p$. The malicious parties may specify errors to be added to a private value x and its MAC $\hat{x}$, resulting in the values x' and $\hat{x}'$. They win if $x \neq x'$ and $\hat{x}' = \alpha \cdot x'$. It is evident that they must correctly guess α to successfully pass the check, leading to a winning probability of $\frac{1}{|\mathbb{F}_p|}$.

4 PROBLEM STATEMENT

4.1 Functionality and Notation

We consider there are N set owners in the system, each denoted by an index $i \in [N]$, and possessing a set $\mathcal{S}_i$. Here, $[N]$ denotes the set $\{1, 2, \cdots, N\}$. We use $e_{ij}, j \in [|\mathcal{S}_i|]$ to represent each element in $\mathcal{S}_i$, where $|\mathcal{S}_i|$ is the size of $\mathcal{S}_i$, $e_{ij} \in \mathcal{D}$ and $\mathcal{D}$ is the domain of the elements. Consistent with prior studies [28, 31], we assume that each element $e \in \mathcal{D}$ appears at most once in each set $\mathcal{S}_i$. In this paper, all computations are conducted in the finite field $\mathbb{F}_p$. Given a collection of sets $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_N$, we focus on two commonly used and practical set operations: *intersection* $\mathcal{S}_1 \cap \mathcal{S}_2 \cap \cdots \cap \mathcal{S}_N$ and *union* $\mathcal{S}_1 \cup \mathcal{S}_2 \cup \cdots \cup \mathcal{S}_N$.

4.2 System Architecture

Figure 2 illustrates the system architecture of FastPSC. In the system, there are three types of entities: set owners, computation nodes (i.e., servers), and the querier. The querier wants to obtain the intersection or union of the sets of N set owners. For example, in the field of healthcare, the health data of patients are usually distributed in different medical institutions [28]. By performing the union operation, patient health records from different medical institutions (i.e., set owners) can be merged to provide doctors or researchers (i.e., the querier) with more comprehensive patient information to support diagnosis, treatment, and disease prevention. FastPSC considers an outsourcing service paradigm in which the computation nodes collect sets from the set owners, execute set operations on these sets, and subsequently return the result to the querier. However, due to privacy concerns regarding the proprietary sets and the set operation result, it is essential to embed security in such outsourced services to safeguard both the sets and the result. The workflow in FastPSC is as follows.

(1) Each set owner secretly shares its set among the nodes.

(2) The querier specifies the set operation (intersection or union) and secretly shares its MAC key among the nodes.

(3) The computation nodes perform the set operation on the secret-shared sets. Finally, they return the (secret-shared) result to the querier for recovery and verification.

As will be detailed later, FastPSC builds on the paradigm of selective MPC [21] and RSS for lightweight secure set computation with DP leakage. As such, FastPSC leverages a distributed trust framework, where four computation nodes from distinct trust domains, denoted by $\mathsf{C}_{1234} = \{\mathsf{C}_1, \mathsf{C}_2, \mathsf{C}_3, \mathsf{C}_4\}$, are employed to collaboratively provide the computation service.In practice, C_{1234} could be hosted by different commercial cloud providers or managed by distinct and potentially competing organizations. Such distributed trust framework has recently gained significant attentions in both academia [3, 7, 9, 11, 15, 28, 31, 37] and industry [16, 30, 32].

4.3 Threat Model and Security Guarantees

Threat model. For the computation nodes C_{1234}, similar to prior security designs under the four-server setting [7, 9, 15], FastPSC considers a non-colluding and honest majority threat model. Specifically, FastPSC considers that each of C_{1234} may *individually* try to infer private information while performing the set operations and provides security with abort if at most one of C_{1234} is malicious. The malicious computation node can arbitrarily *deviate* from our protocol, engaging in actions like executing computations divergent from expectations or substituting its inputs with those from other computation nodes. FastPSC allows the malicious computation node to collude with at most $N-1$ set owners. In addition, since the querier wants to obtain correct results and acts as a verifier for the integrity of computation at the computation nodes, FastPSC assumes that it does not collude with any of the computation nodes.

Security guarantees. Under the aforementioned threat model, FastPSC guarantees that the adversary only learns the noisy count of owners for each element with DP guarantee, and nothing more. We consider two multisets of all owners' elements that differ by a single element as neighboring inputs.

If one of C_{1234} is malicious and deviates from our protocol, FastPSC guarantees that the querier can detect the discrepancy and abort. In addition, FastPSC guarantees that the querier only learns the result set and nothing beyond that. Consistent with prior works [3, 28, 31], FastPSC treats the number of owners N and the element domain $\mathcal{D}$ as publicly accessible information for all parties. Consistent with prior works [3, 28], FastPSC does not consider scenarios where the set owners provide manipulated sets as input.

5 A STRAW-MAN APPROACH

We introduce a straw-man approach as a warmup. We will pinpoint the issues faced by this approach, which motivate the design of our proposed solution to be presented later. It works as follows:

Step I: Set padding and secret sharing. Each set owner first pads "some" dummy elements (marked with flag 0) sampled from $\mathcal{D}$ into their sets. The real elements in the sets are marked with flag 1. Each set owner then applies RSS on the flags and forwards the pairs (element, ⟦flag⟧) to the computation nodes C_{123} (solely C_{123} are employed in the straw-man solution). Here, we use RSS instead of ASS to safeguard against tampering by the malicious computation node as soon as it receives the secret shares.

Step II: Secure result set construction. C_{123} receive the data from all set owners, locally aggregate secret-shared flags attached to identical elements to obtain the secret-shared count for each element, and then return the pairs (element, ⟦count⟧) to the querier.

Step III: Result set verification and phrasing. Upon receiving the result, the querier verifies its integrity by comparing the secret shares from different computation nodes. If the adversary deviates from the protocol, the secret sharing of the results will not be in the "replicated" form. If the result passes the verification, the querier recover the counts to obtain the pairs (element, count). The elements in the *intersection* are those with a count equal to N, while the elements in the *union* are those with a count not equal to 0.

While the straw-man solution seems to meet our objective, it encounters three issues, each corresponding to a step:

Issue I: The computation nodes can infer elements that are not contained in each set. Note that a set owner pads dummy elements to its set without removing any real elements from the set. Therefore, in the view of C_{123}, if a particular element does not appear in its set, they can learn that the element is not contained in its set. Therefore, the straw-man solution falls to meet our security guarantee for each set. The *full padding* strategy, i.e., for each element $e \in \mathcal{D}$ not in its set, the set owner pads a dummy pair $(e, 0)$ into its set, seems to address this issue. However, this requires substantial communication cost from the set owners to the computation nodes, as well as high computation cost in the subsequent secure set computation phase.

Issue II: The computation nodes can infer elements that are not contained in the result set. Similar to Issue I, from the perspective of C123, if an element does not appear in any of the input sets, it can be confidently inferred that the element is not part of the result set. Furthermore, if the occurrence count of an element is less than N, C123 can deterministically conclude that the element will not be included in the intersection. Therefore, the straw-man approach fails to satisfy our security guarantees for the result set.

Issue III: The querier can learn additional information beyond the result set it queries. Since C_{123} return the count of each element to the querier, it obtains the histogram of elements instead of the result set. The querier can learn additional information beyond the result set from the histogram. For example, it can learn: (1) both the intersection and the union even if it only queries one of them; (2) the number of set owners possessing an element not in the intersection; (3) the number of set owners possessing an element in the union. Therefore, the straw-man solution falls to meet our security guarantee for the result set against the querier.

6 OUR PROPOSED APPROACH

6.1 An Overview of Our Solution

(**Section 6.2**) To address **Issue I**, we have each set owner secretly share the real and dummy elements, rather than only sharing their flags, i.e., (⟦element⟧, ⟦flag⟧). To address **Issue II**, we draw inspiration from selective MPC [21] and have the set owners secretly share each pair (⟦element⟧, ⟦flag⟧) among three randomly selected computation nodes from C_{1234}, rather than fixed C_{123}. This ensures that in the subsequent element revelation phase, each computation node only learns a random subset of the set elements. In other words, FastPSC simulates deleting some real elements from the view of each computation node, capturing subtractive noise (similar to how adding dummy elements captures additive noise), thereby achieving pure DP without compromising the accuracy.

(**Section 6.3**) We then consider how to reveal the obfuscated set elements. A naive method is to have every three of C_{1234} directly recover the secret-shared elements they hold. However, this method at most achieves local DP [25]. Achieving comparable accuracy to central DP, local DP demands a substantial increase in randomization [4, 5], which translates to the requirement of adding a large number of dummy elements. To address this problem, we layer the oblivious shuffle technique [6] with DP. Specifically, we have every three of C_{1234} obliviously shuffle the secret-shared pairs (⟦element⟧, ⟦flag⟧) they hold before recovering the elements to them. The oblivious shuffle ensures that, upon revealing an element, the computation nodes remain unaware of its owner. Nevertheless, a shuffle protocol with malicious security is still required. Although such shuffle protocols are available in [2, 14], they cannot be seamlessly integrated into our solution. Specifically, the oblivious shuffle protocol in [14] is constructed based on 2-out-of-2 ASS instead of 2-out-of-3 RSS, whereas the oblivious shuffle protocol in [2] is tailored for binary values. Hence, leveraging information-theoretic MACs [10], we design a new maliciously secure shuffle protocol. C_{1234} can then reveal the shuffled elements. In Section 7.1, we will prove that the information each computation node can learn during the revelation is bounded by our pure DP guarantee.

(**Section 6.4**) To solve **Issue III**, we design secure randomization mechanisms. They enable the computation nodes to randomize the secret-shared count of each element in the histogram based on the set operation type. They ensure that, from the querier's view, the elements in the result set exhibit counts entirely distinct from those not in the result set. In other words, from the randomized histogram, the querier only learns the result set and nothing beyond that.

6.2 Secure Collection of Multi-Owner Sets

Set owner i performs the following steps on its set $\mathcal{S}_i$:

(1) **Preparation:** Each element in $\mathcal{S}_i$ is assigned a flag with a value of 1: $\mathcal{S}'_i = \{(e_{ij}, f_{ij}) | e_{ij} \in \mathcal{S}_i\}$, where $f_{ij} = 1$.

(2) **Padding:** Set owner i randomly samples an integer r_i from the *geometric distribution* with parameter q:

$$\mathbb{G}(r_i = x) = (1 - q)^{x-1} \cdot q,$$

where $0 < q \leq 1$ and $1 \leq x$. Next, r_i dummy elements are randomly sampled with replacement from the element domain $\mathcal{D}$, and these dummy elements, each along with a flag of 0, are padded into $\mathcal{S}'_i$.

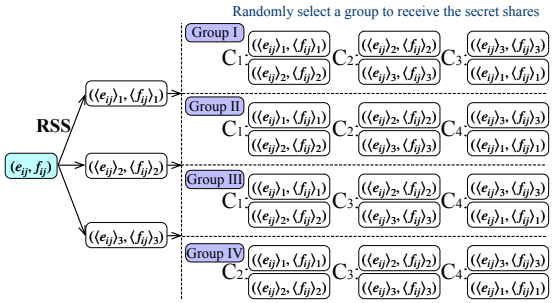

Figure 3: Set owner process for an (element, flag) pair.

(3) **Secret sharing:** Given each pair $(e_{ij}, f_{ij}) \in \mathcal{S}'_i$, set owner i splits it into $(\llbracket e_{ij} \rrbracket, \llbracket f_{ij} \rrbracket)$. C_{1234} are categorized into four groups: Group I - $(\mathsf{C}_1, \mathsf{C}_2, \mathsf{C}_3)$, Group II - $(\mathsf{C}_1, \mathsf{C}_2, \mathsf{C}_4)$, Group III - $(\mathsf{C}_1, \mathsf{C}_3, \mathsf{C}_4)$, and Group IV - $(\mathsf{C}_2, \mathsf{C}_3, \mathsf{C}_4)$. Then, set owner i randomly selects a group to receive the shares. Specifically, in the selected group: the first node receives $(\langle e_{ij} \rangle_1, \langle f_{ij} \rangle_1)$, $(\langle e_{ij} \rangle_2, \langle f_{ij} \rangle_2)$, the second computation node receives $(\langle e_{ij} \rangle_2, \langle f_{ij} \rangle_2)$, $(\langle e_{ij} \rangle_3, \langle f_{ij} \rangle_3)$, and the third node receives $(\langle e_{ij} \rangle_3, \langle f_{ij} \rangle_3)$, $(\langle e_{ij} \rangle_1, \langle f_{ij} \rangle_1)$. Figure 3 summarizes the process. In addition, for each $(\llbracket e_{ij} \rrbracket, \llbracket f_{ij} \rrbracket)$, set owner i ensures that the three computation nodes in the selected group are aware of each other to facilitate subsequent computations in the RSS domain. The RSS of the pairs held by Group $\mathsf{K} \in \{\mathsf{I}, \mathsf{II}, \mathsf{III}, \mathsf{IV}\}$ are denoted as $\llbracket \mathcal{S}^{\mathsf{K}} \rrbracket = \{(\llbracket e_i \rrbracket, \llbracket f_i \rrbracket)\}$.

In this manner, **Issue I** is addressed.

Creating MACs. To create the MACs, a naive method involves having each set owner authenticate (e_{ij}, f_{ij}) to produce the corresponding MACs. Subsequently, the MACs are secretly shared among the same selected group as that of (e_{ij}, f_{ij}). However, this requires all the set owners share the same MAC key. In this setting, the malicious computation node only needs to collude with a single set owner to obtain the MAC key. This issue also exists in the work [28]. Therefore, to establish malicious security under a stronger threat model, we adopt an alternative strategy: shifting the MAC creation from the set owners' side to the computation nodes' side. Specifically, the querier secretly shares the same MAC key $\alpha \in \mathbb{F}_p$ in RSS among each group. The secret-shared MAC key held by each group is uniformly denoted as $\llbracket \alpha \rrbracket$ for simplicity. After that, Group $\mathsf{K} \in \{\mathsf{I}, \mathsf{II}, \mathsf{III}, \mathsf{IV}\}$ authenticates the secret-shared pairs in $\llbracket \mathcal{S}^{\mathsf{K}} \rrbracket$ by multiplying them with $\llbracket \alpha \rrbracket$. The secret-shared MACs are denoted as $\llbracket \hat{\mathcal{S}}^{\mathsf{K}} \rrbracket$. Specifically, given $(\llbracket e_i \rrbracket, \llbracket f_i \rrbracket) \in \llbracket \mathcal{S}^{\mathsf{K}} \rrbracket$, Group K computes $\llbracket \hat{e}_i \rrbracket = \llbracket \alpha \rrbracket \cdot \llbracket e_i \rrbracket$; $\llbracket \hat{f}_i \rrbracket = \llbracket \alpha \rrbracket \cdot \llbracket f_i \rrbracket$, then adds $(\llbracket \hat{e}_i \rrbracket, \llbracket \hat{f}_i \rrbracket)$ into $\llbracket \hat{\mathcal{S}}^{\mathsf{K}} \rrbracket$.

The security of the online MACs creation is as follows. Each secret share held by the malicious computation node has a replica held by an honest computation node. Therefore, if the malicious computation node tampers with the secret shares immediately upon receipt, the "replicated form" of the secret shares will be compromised. Additionally, if the malicious computation node sends a tampered secret share of the MAC during the re-sharing at the end of the secret-shared multiplication, it cannot guarantee that the output will pass the MAC verification.

6.3 Set Elements Revelation with DP Guarantees

We next describe how to reveal the set elements to C_{1234}. Our solution lies in our maliciously secure shuffle protocol.

Our maliciously secure shuffle protocol. Our protocol is built upon the secure shuffle protocol in [2]. Our main contributions can be summarized in the following three aspects:

(1) We simplify the protocol [2] to output the shuffling results in the form of 3-out-of-3 ASS, because FastPSC does not require the RSS-formed shuffling results. This simplification reduces the communication rounds to 1, compared to the 2 rounds required by the protocol [2].

(2) We adapt the protocol [2] to support arithmetic values.

(3) As the malicious security of the protocol [2] is established based on the characteristics of binary values, our adaptation to accommodate arithmetic values precludes a direct inheritance of malicious security from it. Consequently, another contribution is the incorporation of information-theoretic MACs into our arithmetic values-based protocol to achieve malicious security.

Our oblivious shuffle protocol is denoted as $(\llbracket \mathbf{T}_\pi \rrbracket^A, \llbracket \hat{\mathbf{T}}_\pi \rrbracket^A) = \mathsf{maliShuffle}(\llbracket \mathbf{T} \rrbracket, \llbracket \hat{\mathbf{T}} \rrbracket)$. Here, $\llbracket \mathbf{T} \rrbracket$ and $\llbracket \hat{\mathbf{T}} \rrbracket$ are held by three parties P_1, P_2, and P_3. $\mathbf{T}$ is the table to be shuffled; $\hat{\mathbf{T}}$ is the table consisting of the MAC of each element in $\mathbf{T}$. Specifically, each table comprises $n \cdot m$ values, with each value belonging to $\mathbb{F}_p$. It holds that $\forall i \in [n], j \in [m], \hat{\mathbf{T}}[i, j] = \alpha \cdot \mathbf{T}[i, j]$, where α is the MAC key unknown to the parties. $\mathbf{T}_\pi$ and $\hat{\mathbf{T}}_\pi$ are the output tables resulting from the application of the *same* random permutation $\pi(\cdot)$—unknown to the parties—to the *rows* of $\mathbf{T}$ and $\hat{\mathbf{T}}$, respectively. The outputs $\llbracket \mathbf{T}_\pi \rrbracket^A$ and $\llbracket \hat{\mathbf{T}}_\pi \rrbracket^A$ are in the form of 3-out-of-3 ASS. Figure 4 depicts our maliciously secure shuffle protocol. Correctness holds since the sum of the new shares $\langle \mathbf{T}_\pi \rangle_1, \langle \mathbf{T}_\pi \rangle_2, \langle \mathbf{T}_\pi \rangle_3$ equals the shuffled table:

$$\begin{aligned}
\mathbf{A}_{12} + \mathbf{B}_{31} + \mathbf{Z} + \mathbf{U} =& \pi_{23}(\mathbf{X} + \mathbf{R}_{23}) + \pi_{23}(\pi_{31}(\mathbf{Y} - \mathbf{R}_{31}) - \mathbf{R}_{23}) \\
=& \pi_{23}(\mathbf{X} + \pi_{31}(\mathbf{Y} - \mathbf{R}_{31})) \\
=& \pi_{23}(\pi_{31}(\pi_{12}(\langle \mathbf{T} \rangle_1 + \langle \mathbf{T} \rangle_2 + \mathbf{R}_{12}) + \mathbf{Y})) \\
=& \pi_{23}(\pi_{31}(\pi_{12}(\langle \mathbf{T} \rangle_1 + \langle \mathbf{T} \rangle_2 + \langle \mathbf{T} \rangle_3))),
\end{aligned}$$

where $\pi_{23}(\pi_{31}(\pi_{12}(\cdot)))$ is the overall random permutation $\pi(\cdot)$. Similarly, correctness also holds for the MACs $\llbracket \hat{\mathbf{T}} \rrbracket$.

Shuffling and revealing the set elements. To apply $\mathsf{maliShuffle}(\cdot)$ on $\llbracket \mathcal{S}^{\mathsf{K}} \rrbracket$ and $\llbracket \hat{\mathcal{S}}^{\mathsf{K}} \rrbracket$, Group $\mathsf{K} \in \{\mathsf{I}, \mathsf{II}, \mathsf{III}, \mathsf{IV}\}$ needs to represent each of them as a table. Each row of the table corresponding to $\llbracket \mathcal{S}^{\mathsf{K}} \rrbracket$ is one pair $(\llbracket e_i \rrbracket, \llbracket f_i \rrbracket) \in \llbracket \mathcal{S}^{\mathsf{K}} \rrbracket$, and each row of the table corresponding to $\llbracket \hat{\mathcal{S}}^{\mathsf{K}} \rrbracket$ is one pair $(\llbracket \hat{e}_i \rrbracket, \llbracket \hat{f}_i \rrbracket) \in \llbracket \hat{\mathcal{S}}^{\mathsf{K}} \rrbracket$. The shuffled sets are denoted as $\llbracket \mathcal{S}^{\mathsf{K}}_\pi \rrbracket^A$ and $\llbracket \hat{\mathcal{S}}^{\mathsf{K}}_\pi \rrbracket^A$ (in 3-out-of-3 ASS), respectively. After that, for each $(\llbracket e_i \rrbracket^A, \llbracket f_i \rrbracket^A) \in \llbracket \mathcal{S}^{\mathsf{K}}_\pi \rrbracket^A$, the computation nodes of Group K broadcast their respective secret share of e_i in Group K to reveal e_i (without revealing its flag f_i). The oblivious shuffle operation breaks the link between the revealed elements and their owners. In addition, each computation node only has access to a random subset of the result set, thereby addressing **Issue II**. In Section 7.1, we will prove that the information each computation node learns during the revelation is bounded by our pure DP guarantee.

Verifying the revealed elements. The integrity of each revealed element e_i can be verified by having the t-th ($t \in [3]$) computation node in Group K broadcast its secret share $e_i \cdot \langle \alpha \rangle_t - \langle \hat{e}_i \rangle_t$ in the

Party	P_1	P_2	P_3
Input	$(\langle\mathbf{T}\rangle_1, \langle\mathbf{T}\rangle_2), (\langle\hat{\mathbf{T}}\rangle_1, \langle\hat{\mathbf{T}}\rangle_2)$ $\mathbf{R}_{12}, \mathbf{R}_{31}, \overline{\mathbf{R}}_{12}, \overline{\mathbf{R}}_{31}, \mathbf{A}_{12}, \mathbf{B}_{31}, \overline{\mathbf{A}}_{12}, \overline{\mathbf{B}}_{31}$ π_{12}, π_{31}	$(\langle\mathbf{T}\rangle_2, \langle\mathbf{T}\rangle_3), (\langle\hat{\mathbf{T}}\rangle_2, \langle\hat{\mathbf{T}}\rangle_3)$ $\mathbf{R}_{12}, \mathbf{R}_{23}, \overline{\mathbf{R}}_{12}, \overline{\mathbf{R}}_{23}, \mathbf{A}_{12}, \overline{\mathbf{A}}_{12}$ π_{12}, π_{23}	$(\langle\mathbf{T}\rangle_3, \langle\mathbf{T}\rangle_1), (\langle\hat{\mathbf{T}}\rangle_3, \langle\hat{\mathbf{T}}\rangle_1)$ $\mathbf{R}_{31}, \mathbf{R}_{23}, \overline{\mathbf{R}}_{31}, \overline{\mathbf{R}}_{23}, \mathbf{B}_{31}, \overline{\mathbf{B}}_{31}$ π_{31}, π_{23}
Step I	$\mathbf{X} = \pi_{31}(\pi_{12}(\langle\mathbf{T}\rangle_1 + \langle\mathbf{T}\rangle_2 + \mathbf{R}_{12}) + \mathbf{R}_{31})$ $\overline{\mathbf{X}} = \pi_{31}(\pi_{12}(\langle\hat{\mathbf{T}}\rangle_1 + \langle\hat{\mathbf{T}}\rangle_2 + \overline{\mathbf{R}}_{12}) + \overline{\mathbf{R}}_{31})$	$\mathbf{Y} = \pi_{12}(\langle\mathbf{T}\rangle_3 - \mathbf{R}_{12})$ $\overline{\mathbf{Y}} = \pi_{12}(\langle\hat{\mathbf{T}}\rangle_3 - \overline{\mathbf{R}}_{12})$	
Communication	$\mathbf{X}$ and $\overline{\mathbf{X}}$ — Send to →	$\mathbf{Y}$ and $\overline{\mathbf{Y}}$ — Send to →	
Step II		$\mathbf{Z} = \pi_{23}(\mathbf{X} + \mathbf{R}_{23}) - \mathbf{A}_{12}$ $\overline{\mathbf{Z}} = \pi_{23}(\overline{\mathbf{X}} + \overline{\mathbf{R}}_{23}) - \overline{\mathbf{A}}_{12}$	$\mathbf{U} = \pi_{23}(\pi_{31}(\mathbf{Y} - \mathbf{R}_{31}) - \mathbf{R}_{23}) - \mathbf{B}_{31}$ $\overline{\mathbf{U}} = \pi_{23}(\pi_{31}(\overline{\mathbf{Y}} - \overline{\mathbf{R}}_{31}) - \overline{\mathbf{R}}_{23}) - \overline{\mathbf{B}}_{31}$
Output	$\langle\mathbf{T}_\pi\rangle_1 = \mathbf{A}_{12} + \mathbf{B}_{31}, \langle\hat{\mathbf{T}}_\pi\rangle_1 = \overline{\mathbf{A}}_{12} + \overline{\mathbf{B}}_{31}$	$\langle\mathbf{T}_\pi\rangle_2 = \mathbf{Z}, \langle\hat{\mathbf{T}}_\pi\rangle_2 = \overline{\mathbf{Z}}$	$\langle\mathbf{T}_\pi\rangle_3 = \mathbf{U}, \langle\hat{\mathbf{T}}_\pi\rangle_3 = \overline{\mathbf{U}}$

Figure 4: Our maliciously secure shuffle protocol. Each capitalized boldface letter represents a table of dimensions $n \cdot m$, where each element belongs to the finite field $\mathbb{F}_p$. π represents a permutation with n rows. Each entity with subscript ij (e.g., $\mathbf{R}_{12}, \pi_{31}, \overline{\mathbf{B}}_{31}$), signifies its local generation by parties P_i and P_j through the same pseudo-random function, inputting their jointly held random seed. The inputs in gray indicate that they are not used in the protocol.

group to reveal $e_i \cdot \alpha - \hat{e}_i$. Here, $\langle\alpha\rangle_t$ is the share of MAC key α, with $\langle\alpha\rangle_1 + \langle\alpha\rangle_2 + \langle\alpha\rangle_3 = \alpha$, and $\hat{e}_i$ is the MAC of e_i. If $e_i \cdot \alpha - \hat{e}_i = 0$ holds, the honest computation nodes accept e_i; otherwise, they abort the process and return $\perp$ to the querier. However, since many such values will be revealed, simply employing the above method requires the computation nodes to broadcast $e_i \cdot \langle\alpha\rangle_t - \langle\hat{e}_i\rangle_t$ for each e_i, consuming a significant amount of additional communication cost. To avoid this, we use the batch MAC checking technique from [10], only requiring each computation node to broadcast a single message for all revealed elements. Specifically, the computation nodes in Group K reveal the random linear combination of $e_i \cdot \alpha - \hat{e}_i$ for all e_i in $\mathcal{S}_\pi^{\mathsf{K}}$. If the revealed value is non-zero, the honest computation nodes abort the process and return $\perp$ to the querier.

6.4 Secure Construction of Result Set

FastPSC has the computation nodes first compute the secret-shared count of each element and then construct the secret-shared result set based on these secret-shared counts.

Computing secret-shared element counts. As per the secure set elements revelation phase, each computation node $\mathsf{C}_t, t \in [4]$ sees tuples containing *a plaintext element*, a share of the flag (in 3-out-of-3 ASS), and a share of the MAC of the flag (in 3-out-of-3 ASS), denoted as $(e_i, \langle f_i\rangle, \langle\hat{f}_i\rangle)$ (the subscript of secret share is omitted for brevity). To compute the secret-shared count of each element $e \in \mathcal{D}$, the computation nodes compute over the secret shares of the flags. Recall that real elements have a flag with a value of 1, while dummies have a flag with a value of 0. Therefore, we design Algorithm 1, which enables C_t to locally compute the share of the sum of all flags (as well as the share of the sum of all flags' MACs) corresponding to each element $e \in \mathcal{D}$.

Algorithm 1 Secure Element Count Computation

Input: A set of tuples $(e_i, \langle f_i\rangle, \langle\hat{f}_i\rangle)$.
Output: The share of element counts: $(e, \langle c_e\rangle_t, \langle\hat{c}_e\rangle_t), e \in \mathcal{D}$.
1: Initialization: $\langle c_e\rangle_t = 0, \langle\hat{c}_e\rangle_t = 0, e \in \mathcal{D}$.
2: **for** each tuple $(e_i, \langle f_i\rangle, \langle\hat{f}_i\rangle)$ in the input set **do**
3: $\quad\langle c_{e_i}\rangle_t = \langle c_{e_i}\rangle_t + \langle f_i\rangle; \langle\hat{c}_{e_i}\rangle_t = \langle\hat{c}_{e_i}\rangle_t + \langle\hat{f}_i\rangle$.
4: **end for**

Algorithm 1 inputs $\mathsf{C}_{t\in[4]}$'s set of tuples $(e_i, \langle f_i\rangle, \langle\hat{f}_i\rangle)$ and outputs $(e, \langle c_e\rangle_t, \langle\hat{c}_e\rangle_t), e \in \mathcal{D}$. Here, c_e is the count of element e and $\hat{c}_e$ is c_e's MAC. Since each computation node locally aggregates the (3-out-of-3) secret shares of both the flags and their MACs associated with element e, both c_e and $\hat{c}_e$ are in the form of 4-out-of-4 ASS, i.e., $\langle c_e\rangle_1 + \langle c_e\rangle_2 + \langle c_e\rangle_3 + \langle c_e\rangle_4 = c_e$ and $\langle\hat{c}_e\rangle_1 + \langle\hat{c}_e\rangle_2 + \langle\hat{c}_e\rangle_3 + \langle\hat{c}_e\rangle_4 = \hat{c}_e$. Correctness holds because, based on the properties of ASS, the sum of all secret shares of the flags equals the sum of the flags themselves. Given that real elements have a flag of 1 and dummies have a flag of 0, c_e must be the count of each element e. In addition, real elements have a MAC equal to α and dummies have a MAC equal to 0, and thus $\hat{c}_e$ must be the MAC of e's count, i.e., $\hat{c}_e = c_e \cdot \alpha$.

Constructing the secret-shared result set. We then introduce how to construct the result set based on the secret-shared counts of all elements. Our method relies on the observation:

If an element is present in the intersection, its count is equal to N; if a specific element is present in the union, its count is greater than 0.

A viable approach is to have C_{1234} directly return the secret-shared counts of all elements to the querier for recovering and verification. The querier derives the intersection and union from the counts. This method, however, leads to **Issue III**. Therefore, we design a mechanism to randomize the counts, enabling the querier to only learn the intersection or union and nothing beyond that.

To construct the secret-shared *intersection*, for $(e, \langle c_e\rangle_t, \langle\hat{c}_e\rangle_t), e \in \mathcal{D}, \mathsf{C}_t, t \in [4]$ perform the following:

$$\mathsf{C}_1 : \langle c'_e\rangle_1 = -\langle c_e\rangle_1 \cdot a_e; \quad \langle\hat{c}'_e\rangle_1 = (N \cdot \langle\alpha\rangle_1 - \langle\hat{c}_e\rangle_1) \cdot a_e;$$
$$\mathsf{C}_2 : \langle c'_e\rangle_2 = -\langle c_e\rangle_2 \cdot a_e; \quad \langle\hat{c}'_e\rangle_2 = (N \cdot \langle\alpha\rangle_2 - \langle\hat{c}_e\rangle_2) \cdot a_e;$$
$$\mathsf{C}_3 : \langle c'_e\rangle_3 = -\langle c_e\rangle_3 \cdot a_e; \quad \langle\hat{c}'_e\rangle_3 = (N \cdot \langle\alpha\rangle_3 - \langle\hat{c}_e\rangle_3) \cdot a_e;$$
$$\mathsf{C}_4 : \langle c'_e\rangle_4 = (N - \langle c_e\rangle_4) \cdot a_e; \langle\hat{c}'_e\rangle_4 = -\langle\hat{c}_e\rangle_4 \cdot a_e,$$

where $a_e, e \in \mathcal{D}$ are non-zero numbers from $\mathbb{F}_p$. They can be locally generated by C_{1234} using the same pseudo-random function and the same random seed. $\langle\alpha\rangle_1, \langle\alpha\rangle_2, \langle\alpha\rangle_3$ are the secret shares of the MAC key α, i.e., $\langle\alpha\rangle_1 + \langle\alpha\rangle_2 + \langle\alpha\rangle_3 = \alpha$. It is important to note that our objective here is to randomize the element count c_e and its MAC $\hat{c}_e$ rather than the secret shares of c_e and $\hat{c}_e$. In addition, given that the querier can compute $\hat{c}'_e$ by $c'_e \cdot \alpha$ (assuming the malicious

computation node does not deviate from the protocol). Therefore, it is secure to reuse a_e for each secret share of c_e and $\hat{c}_e$.

To construct the secret-shared *union*, for $(e, \langle c_e \rangle_t, \langle \hat{c}_e \rangle_t)$, $e \in \mathcal{D}$, $\mathsf{C}_t, t \in [4]$ perform the following:

$$\begin{aligned}
&\mathsf{C}_1 : \langle c'_e \rangle_1 = \langle c_e \rangle_1 \cdot b_e; && \langle \hat{c}'_e \rangle_1 = \langle \hat{c}_e \rangle_1 \cdot b_e;\\
&\mathsf{C}_2 : \langle c'_e \rangle_2 = \langle c_e \rangle_2 \cdot b_e; && \langle \hat{c}'_e \rangle_2 = \langle \hat{c}_e \rangle_2 \cdot b_e;\\
&\mathsf{C}_3 : \langle c'_e \rangle_3 = \langle c_e \rangle_3 \cdot b_e; && \langle \hat{c}'_e \rangle_3 = \langle \hat{c}_e \rangle_3 \cdot b_e;\\
&\mathsf{C}_4 : \langle c'_e \rangle_4 = \langle c_e \rangle_4 \cdot b_e; && \langle \hat{c}'_e \rangle_4 = \langle \hat{c}_e \rangle_4 \cdot b_e,
\end{aligned}$$

where $b_e, e \in \mathcal{D}$ are non-zero numbers in $\mathbb{F}_p$. They can be generated by C_{1234} using the same pseudo-random function and random seed.

Finally, $\mathsf{C}_t, t \in [4]$ returns $(e, \langle c'_e \rangle_t, \langle \hat{c}'_e \rangle_t), e \in \mathcal{D}$ to the querier.

Parsing and verifying the result set. The querier can recover $(e, c'_e, \hat{c}'_e), e \in \mathcal{D}$, and then verify their integrity. As each computation node returns the tuple containing e, its integrity can be simply checked by comparing whether the value e received from different computation nodes are identical. Therefore, the querier only checks the integrity of $c'_e, e \in \mathcal{D}$. The integrity can be checked by evaluating whether $\forall e \in \mathcal{D}, c'_e \cdot \alpha - \hat{c}'_e = 0$ holds. If it does not hold, the querier will abort the result. The correctness is as follows.

• *Intersection*: If the malicious computation node does not deviate from the protocol, $c'_e = (N - c_e) \cdot a_e$, $\hat{c}'_e = (N \cdot \alpha - \hat{c}_e) \cdot a_e$, and $\hat{c}_e = c_e \cdot \alpha$. Therefore, $c'_e \cdot \alpha - \hat{c}'_e = (N - c_e) \cdot a_e \cdot \alpha - (N \cdot \alpha - c_e \cdot \alpha) \cdot a_e = 0$.

• *Union*: If the malicious computation node does not deviate from the protocol, $c'_e = c_e \cdot b_e$, $\hat{c}'_e = \hat{c}_e \cdot b_e$, and $\hat{c}_e = c_e \cdot \alpha$. Therefore, $c'_e \cdot \alpha - \hat{c}'_e = c_e \cdot b_e \cdot \alpha - c_e \cdot \alpha \cdot b_e = 0$.

If the result passes verification, the querier then parses it to obtain the intersection or union.

• *Intersection*: $\forall e \in \mathcal{D}$, if $c'_e = 0$, the querier can conclude that e is in the intersection; if $c'_e \neq 0$, the querier can conclude that e is not in the intersection. The correctness is as follows: If every set owner holds e, its count $c_e = N$, and thus $c'_e = (N - c_e) \cdot a_e = 0 \cdot a_e = 0$. If not every set owner holds e, its count $c_e < N$, and $(N - c_e) \neq 0$. As both a_e and $(N - c_e)$ are non-zero numbers in the finite field $\mathbb{F}_p$, it follows that $c'_e = (N - c_e) \cdot a_e \neq 0$. FastPSC also guarantees that if an element e is not in the intersection, in the querier's view, c'_e is a non-zero random value in $\mathbb{F}_p$. In other words, the querier only learns the intersection and nothing beyond that. In Section 7.2, we will provide a formal proof of this.

• *Union*: $\forall e \in \mathcal{D}$, if $c'_e \neq 0$, the querier can conclude that e is in the union; if $c'_e = 0$, the querier can conclude that e is not in the union. The correctness is as follows: If there exists at least one set owner holding e, its count $c_e \neq 0$, and thus $c'_e = c_e \cdot a_e \neq 0$. If no set owner holds e, its count $c_e = 0$, and thus $c'_e = c_e \cdot a_e = 0$. FastPSC also guarantees that the querier only learns whether a particular element is in the union or not, and cannot learn the count of set owners holding each element in the union. In other words, the querier only learns the union and nothing beyond that. In Section 7.2, we will provide a proof of this. Hence, **Issue III** is addressed.

7 PRIVACY ANALYSIS

7.1 Privacy Against the Computation Nodes

Recall that during the execution of FastPSC, in addition to the publicly available information, the computation nodes can only learn additional information during the revelation of set elements (Section 6.3). In this phase, a single computation node obtains a set of tuples $(e_i, \langle f_i \rangle, \langle \hat{f}_i \rangle)$. These shares reveal nothing about the corresponding values. Therefore, a single computation node only learns a noisy count about each possible element. These information can be described as a histogram, denoted as L. Let $\mathcal{M}$ represent the algorithm for revealing set elements in Section 6.3. Then L is the output of $\mathcal{M}$. We next demonstrate that $\mathcal{M}$ is ϵ_L-DP.

Theorem 1. *$\mathcal{M}$ is ε_L-DP, where $\varepsilon_L = \ln\left(\max\left\{\frac{q}{1-q} \cdot |\mathcal{D}| + 1, 4\right\}\right)$, q is the geometric distribution's parameter, $|\mathcal{D}|$ is the domain size.*

Proof. We define two multisets of elements owned by all set owners, $\mathcal{S}$ and $\mathcal{S}'$, as neighboring if they differ in at most the addition or deletion of one element. We represent the true histogram of the elements as the vector $c_{\mathcal{S}}$. Let $c_{\mathcal{S}}(e)$ represent the count of element $e \in \mathcal{D}$ on the input $\mathcal{S}$. Therefore, we have $\forall e \in \mathcal{D}, |c_{\mathcal{S}}(e) - c_{\mathcal{S}'}(e)| \leq 1$. We need to prove:

$$\frac{Pr[\mathcal{M}(\mathcal{S}) = L]}{Pr[\mathcal{M}(\mathcal{S}') = L]} = \frac{\prod_{e \in \mathcal{D}} Pr[\mathcal{M}_e(\mathcal{S}) = L_e]}{\prod_{e \in \mathcal{D}} Pr[\mathcal{M}_e(\mathcal{S}') = L_e]} \leq e^{\varepsilon_L}, \tag{1}$$

where $\mathcal{M}_e$ denotes the process for a single element $e \in \mathcal{D}$ and L_e denotes $\mathcal{M}_e$'s output, i.e., the count of element e observed by a computation node. The transition in Eq. 1 stems from the fact that $\mathcal{M}$ applies randomness independently to each element $e \in \mathcal{D}$. As $\mathcal{S}$ and $\mathcal{S}'$ differ by at most the addition or deletion of a single element, we have:

$$\frac{\prod_{e \in \mathcal{D}} Pr[\mathcal{M}_e(\mathcal{S}) = L_e]}{\prod_{e \in \mathcal{D}} Pr[\mathcal{M}_e(\mathcal{S}') = L_e]} = \frac{Pr[\mathcal{M}_e(\mathcal{S}) = L_e]}{Pr[\mathcal{M}_e(\mathcal{S}') = L_e]}, \tag{2}$$

where e is the distinct element between $\mathcal{S}$ and $\mathcal{S}'$.

Recall that each set owner $i \in [N]$ draws a random number r_i from a geometric distribution $\mathbb{G}(x)$ and then pads r_i dummies randomly sampled from $\mathcal{D}$ into their sets. Given that each set owner independently samples the random number, the probability mass function (PMF) for the sum (denoted as R and $R \geq N$) of $r_i, i \in [N]$ can be represented as a *negative binomial distribution*. More specifically, the negative binomial distribution is a probability distribution that characterizes the number of independent and identically distributed Bernoulli trials needed for a predetermined number of successes to take place. Consequently, the PMF of R follows a negative binomial distribution with N ($N \geq 1$) successes and a success probability of q, $\mathbb{N}(R = x)$:

$$\mathbb{N}(x) = \binom{x-1}{N-1} \cdot q^N \cdot (1-q)^{x-N}. \tag{3}$$

In addition, as the dummies are randomly sampled from $\mathcal{D}$, the PMF of the count (denoted as d_e) of element e's dummies can be modeled as the binomial distribution $\mathbb{B}_1(d_e = x; R = y)$:

$$\mathbb{B}_1(x; y) = \binom{y}{x} \cdot \left(\frac{1}{|\mathcal{D}|}\right)^x \cdot \left(1 - \frac{1}{|\mathcal{D}|}\right)^{y-x}. \tag{4}$$

A single computation node receives the secret share of an element in the padded set with a probability of 3/4 (i.e., each computation node belongs to three groups, and the set owner randomly selects a group from the four groups). Therefore, the PMF of the count of element e observed by a single computation node can be modeled as the binomial distribution $\mathbb{B}_2(L_e = x; c_{\mathcal{S}}(e) + d_e = y)$:

$$\mathbb{B}_2(x, y) = \binom{y}{x} \cdot \left(\frac{3}{4}\right)^x \cdot \left(\frac{1}{4}\right)^{y-x},$$

where $c_{\mathcal{S}}(e) + d_e$ is the count of real element e and its dummies in the padded sets. Then we can convert Eq. 2 to:

$$\frac{\sum_{R=N}^{\infty}\sum_{d_e=0}^{\infty}\mathbb{N}(R)\cdot\mathbb{B}_1(d_e;R)\cdot\mathbb{B}_2(L_e;c_{\mathcal{S}}(e)+d_e)}{\sum_{R=N}^{\infty}\sum_{d_e=0}^{\infty}\mathbb{N}(R)\cdot\mathbb{B}_1(d_e;R)\cdot\mathbb{B}_2(L_e;c_{\mathcal{S}'}(e)+d_e)}. \tag{5}$$

We now consider the two worst cases under our assumption that $|c_{\mathcal{S}}(e) - c_{\mathcal{S}'}(e)| \le 1$: (1) $c_{\mathcal{S}}(e) - c_{\mathcal{S}'}(e) = 1$; (2) $c_{\mathcal{S}'}(e) - c_{\mathcal{S}}(e) = 1$. In the first case, we can convert Eq. 5 to:

$$\begin{aligned}&\frac{\sum_{R=N}^{\infty}\sum_{d_e=0}^{\infty}\mathbb{N}(R)\cdot\mathbb{B}_1(d_e;R)\cdot\mathbb{B}_2(L_e;c_{\mathcal{S}'}(e)+d_e+1)}{\sum_{R=N}^{\infty}\sum_{d_e=0}^{\infty}\mathbb{N}(R)\cdot\mathbb{B}_1(d_e;R)\cdot\mathbb{B}_2(L_e;c_{\mathcal{S}'}(e)+d_e)}\\ =&\frac{\sum_{R=N}^{\infty}\sum_{d_e=1}^{\infty}\mathbb{N}(R)\cdot\mathbb{B}_1(d_e-1;R)\cdot\mathbb{B}_2(L_e;c_{\mathcal{S}'}(e)+d_e)}{\sum_{R=N}^{\infty}\sum_{d_e=0}^{\infty}\mathbb{N}(R)\cdot\mathbb{B}_1(d_e;R)\cdot\mathbb{B}_2(L_e;c_{\mathcal{S}'}(e)+d_e)}.\end{aligned} \tag{6}$$

Given the property of combinatorial numbers: $\binom{y}{x} = \binom{y-1}{x} + \binom{y-1}{x-1}$, we can derive $\binom{y+1}{x} = \binom{y}{x} + \binom{y}{x-1}$. Then we can convert the binomial distribution $\mathbb{B}(x-1;y)$ with a success probability p to:

$$\begin{aligned}&\binom{y}{x-1}\cdot p^{x-1}\cdot(1-p)^{y-x+1}\\ =&\left(\binom{y+1}{x}-\binom{y}{x}\right)\cdot p^{x-1}\cdot(1-p)^{y-x+1}\\ =&\frac{1}{p}\cdot\binom{y+1}{x}\cdot p^{x}\cdot(1-p)^{y-x+1}-\frac{1-p}{p}\cdot\binom{y}{x}\cdot p^{x}\cdot(1-p)^{y-x}\\ =&\frac{1}{p}\cdot\mathbb{B}(x;y+1)-\frac{1-p}{p}\cdot\mathbb{B}(x;y).\end{aligned}$$

Therefore, returning to Eq. 6, we have

$$\mathbb{B}_1(d_e-1;R) = |\mathcal{D}|\cdot\mathbb{B}_1(d_e;R+1) - (|\mathcal{D}|-1)\cdot\mathbb{B}_1(d_e;R).$$

We first consider the first part $\mathbb{B}_1(d_e;R+1)$. Since

$$\begin{aligned}&\sum_{R=N}^{\infty}\sum_{d_e=1}^{\infty}\mathbb{N}(R)\cdot\mathbb{B}(d_e;R+1)\cdot\mathbb{C}(L_e;c_{\mathcal{S}'}(e)+d_e)\\ =&\sum_{R=N+1}^{\infty}\sum_{d_e=1}^{\infty}\mathbb{N}(R-1)\cdot\mathbb{B}(d_e;R)\cdot\mathbb{C}(L_e;c_{\mathcal{S}'}(e)+d_e)\\ \le&\sum_{R=N+1}^{\infty}\sum_{d_e=1}^{\infty}\frac{1}{1-q}\cdot\mathbb{N}(R)\cdot\mathbb{B}(d_e;R)\cdot\mathbb{C}(L_e;c_{\mathcal{S}'}(e)+d_e),\end{aligned}$$

we have

$$\begin{aligned}&\frac{\sum_{R=N}^{\infty}\sum_{d_e=1}^{\infty}\mathbb{N}(R)\cdot\mathbb{B}_1(d_e;R+1)\cdot\mathbb{B}_2(L_e;c_{\mathcal{S}'}(e)+d_e)}{\sum_{R=N}^{\infty}\sum_{d_e=0}^{\infty}\mathbb{N}(R)\cdot\mathbb{B}_1(d_e;R)\cdot\mathbb{B}_2(L_e;c_{\mathcal{S}'}(e)+d_e)}\\ \le&\frac{\sum_{R=N+1}^{\infty}\sum_{d_e=1}^{\infty}\frac{1}{1-q}\cdot\mathbb{N}(R)\cdot\mathbb{B}_1(d_e;R)\cdot\mathbb{B}_2(L_e;c_{\mathcal{S}'}(e)+d_e)}{\sum_{R=N}^{\infty}\sum_{d_e=0}^{\infty}\mathbb{N}(R)\cdot\mathbb{B}_1(d_e;R)\cdot\mathbb{B}_2(L_e;c_{\mathcal{S}'}(e)+d_e)}\\ \le&\frac{\sum_{R=N+1}^{\infty}\sum_{d_e=1}^{\infty}\frac{1}{1-q}\cdot\mathbb{N}(R)\cdot\mathbb{B}_1(d_e;R)\cdot\mathbb{B}_2(L_e;c_{\mathcal{S}'}(e)+d_e)}{\sum_{R=N+1}^{\infty}\sum_{d_e=1}^{\infty}\mathbb{N}(R)\cdot\mathbb{B}_1(d_e;R)\cdot\mathbb{B}_2(L_e;c_{\mathcal{S}'}(e)+d_e)}\\ \le&\frac{1}{1-q}.\end{aligned}$$

Finally, we have Eq. $6 \le |\mathcal{D}|\cdot\frac{1}{1-q} - (|\mathcal{D}|-1)\cdot 1 = \frac{q}{1-q}\cdot|\mathcal{D}| + 1$, i.e., under $c_{\mathcal{S}}(e) - c_{\mathcal{S}'}(e) = 1$,

$$\frac{Pr[\mathcal{M}(\mathcal{S}) = L]}{Pr[\mathcal{M}(\mathcal{S}') = L]} \le \frac{q}{1-q}\cdot|\mathcal{D}| + 1. \tag{7}$$

In the second case, we can convert Eq. 5 to:

$$\frac{\sum_{R=N}^{\infty}\sum_{d_e=0}^{\infty}\mathbb{N}(R)\cdot\mathbb{B}_1(d_e;R)\cdot\mathbb{B}_2(L_e;c_{\mathcal{S}'}(e)+d_e-1)}{\sum_{R=N}^{\infty}\sum_{d_e=0}^{\infty}\mathbb{N}(R)\cdot\mathbb{B}_1(d_e;R)\cdot\mathbb{B}_2(L_e;c_{\mathcal{S}'}(e)+d_e)}. \tag{8}$$

Given the property of binomial distribution: $(1-p)\cdot\mathbb{B}(x;y-1) < \mathbb{B}(x;y)$ where p is the success probability (as proven in Appendix A), we have $\frac{1}{4}\cdot\mathbb{B}_2(L_e;c_{\mathcal{S}'}(e)+d_e-1) < \mathbb{B}_2(L_e;c_{\mathcal{S}'}(e)+d_e)$. Namely, under $c_{\mathcal{S}'}(e) - c_{\mathcal{S}}(e) = 1$,

$$\frac{Pr[\mathcal{M}(\mathcal{S}) = L]}{Pr[\mathcal{M}(\mathcal{S}') = L]} < 4. \tag{9}$$

Combining the two cases from Eq. 7 and Eq. 9 gives

$$\frac{Pr[\mathcal{M}(\mathcal{S}) = L]}{Pr[\mathcal{M}(\mathcal{S}') = L]} \le \max\left\{\frac{q}{1-q}\cdot|\mathcal{D}| + 1, 4\right\} = e^{\varepsilon_L},$$

where $\varepsilon_L = \ln\left(\max\left\{\frac{q}{1-q}\cdot|\mathcal{D}| + 1, 4\right\}\right)$. □

A notable feature of our protocol is the existence of a lower bound on ε_L, i.e., $\min\varepsilon_L = \ln 4 = 1.39$. However, this lower bound may be deemed unacceptable in certain scenarios. In Appendix B, we demonstrate that this lower bound can be further diminished by introducing additional computation nodes.

Colluding with set owners. It is noted that Theorem 1 holds for $N \ge 1$, indicating that at least one set owner should effectively pad dummies into its set. This means that even in the worst-case scenario where the malicious computation node colludes with $N-1$ set owners, Theorem 1 remains valid.

Remark. In scenarios where the required number of computation nodes (i.e., four) cannot be fully met in practice, we might delegate the roles of missing computation nodes to (non-colluding) set owners. In such scenarios, the delegated set owners act as substitutes for the absent computation nodes, receiving shares from other set owners and collaborating with the remaining computation nodes to

execute the PSI and PSU protocols. As Theorem 1 demonstrates that the privacy guarantees of FastPSC hold even if a malicious computation node colludes with $N-1$ set owners, this delegation strategy does not compromise the privacy guarantees of the protocol.

7.2 Privacy Against the Querier

Recall that the querier receives a set of tuples $(e, c'_e, \hat{c}'_e), e \in \mathcal{D}$ from the computation nodes. As the MAC $\hat{c}'_e$ does not incorporate additional information (i.e., $\hat{c}'_e = c'_e \cdot \alpha$), the tuples can be simplified to $(e, c'_e), e \in \mathcal{D}$. We then prove that the querier only learns the result set from the (masked) counts and nothing beyond that.

Case of PSI. In the case of PSI, the masked count c'_e of an element e present in the intersection is equal to 0, while the masked count c'_e of an element e not present in the intersection is not equal to 0. Hence, to prevent the querier from learning additional information beyond the result set, we need to prove that the masked count c'_e of an element e not present in the intersection appears random in the view of the querier. Recall that $c'_e = (N - c_e) \cdot a_e$ in the PSI protocol. a_e is independent of other values involved in the generation of c'_e and is uniformly random in the querier's view. Hence, if $N - c_e \neq 0$, c'_e is also uniformly random in the querier's view [1].

Case of PSU. In the case of PSU, the masked count c'_e of an element e present in the union is not equal to 0, while the masked count c'_e of an element e not present in the union is equal to 0. Therefore, we need to prove that the masked count c'_e of an element e present in the union appears random in the view of the querier. Recall that $c'_e = c_e \cdot b_e$ in the PSU protocol. The value of b_e is independent of other values involved in the generation of c'_e and is uniformly random in the querier's view. Hence, if $c_e \neq 0$, c'_e is also uniformly random in the querier's view [1].

8 SECURITY ANALYSIS

We use the simulation paradigm [29] to analyze FastPSC's security guarantees. We aim to ensure that an adversary $\mathcal{A}$ with the ability to *statically* corrupt at most one of C_{1234} and at most $N-1$ set owners cannot obtain additional knowledge beyond the DP leakage L. Proving security within the simulation paradigm involves establishing two worlds: the real world, where the actual protocol is executed by honest parties, and the ideal world, where an ideal functionality $\mathcal{F}$ takes inputs from the parties and directly outputs the result to the relevant party. To ensure that $\mathcal{A}$ cannot distinguish between the real and ideal worlds, a simulator SM "simulates" messages that are similar to what the uncorrupted set owners and honest computation nodes send to the corrupted parties in the real world. If $\mathcal{A}$ cannot distinguish between the real and ideal worlds, we conclude that FastPSC is secure.

Ideal functionality $\mathcal{F}$. $\mathcal{F}$ is defined as follows:

- $\mathsf{Init}(\mathbb{F}_p, \mathcal{D})$: The querier sends the finite field $\mathbb{F}_p$ and the element domain $\mathcal{D}$ to $\mathcal{F}$. $\mathcal{F}$ executes the initialization process with $\mathbb{F}_p$ and $\mathcal{D}$.
- $\mathsf{Colle}(\mathcal{S}_i, i \in [N])$: The set owners send sets $\mathcal{S}_{i\in[N]}$ to $\mathcal{F}$.
- $\mathsf{Set}(\mathsf{Op})$: The querier sends the set operation Op (intersection or union) to $\mathcal{F}$. $\mathcal{F}$ executes the set operation Op on $\mathcal{S}_i, i \in [N]$ and returns the result to the querier.

We allow $\mathcal{F}$ to leak $\mathcal{L} = (\mathbb{F}_p, \mathcal{D}, \mathsf{Op}, L)$ to C_{1234}, where L is the DP leakage analyzed in Section 7.1.

Definition 2. *Let $\prod$ denote the protocol for private set computation. Let $\mathcal{A}$ be an adversary who statically corrupts one of C_{1234} and at most $N-1$ set owners, and let $\mathsf{View}^{\prod}_{\mathsf{Real}}$ be the view of the corrupted parties during the protocol run. In the ideal world, a simulator $\mathcal{S}$ generates a simulated view $\mathsf{View}^{\mathcal{S},\mathcal{L}}_{\mathsf{Ideal}}$ given only the leakage $\mathcal{L}$ of $\mathcal{F}$. We say that $\prod$ is secure, if there exists a probabilistic polynomial time simulator $\mathcal{S}$ such that $\mathsf{View}^{\mathcal{S},\mathcal{L}}_{\mathsf{Ideal}}$ is indistinguishable from $\mathsf{View}^{\prod}_{\mathsf{Real}}$.*

Theorem 2. *Based on Definition 2, FastPSC securely realizes $\mathcal{F}$ with the leakage $\mathcal{L}$.*

We prove Theorem 2 in Appendix C. Next, we demonstrate that the honest computation nodes or the querier can detect the occurrence of deviation behaviors.

Theorem 3. *In FastPSC, a cheating computation node can be caught by the honest computation nodes or the querier with a probability of $1 - \frac{2}{|\mathbb{F}_p|}$.*

We prove Theorem 3 in Appendix D.

9 EXPERIMENTS

9.1 Experimental Setup

We develop a prototype implementation of FastPSC using Python. All experiments are conducted on a workstation equipped with 24 Intel Xeon Gold 6240R CPU cores and 2 TB of external SSD storage, running Ubuntu 20.04.3 LTS. Our implementation confines each computation node to a single thread. The communication between the computation nodes takes place through a loopback device on the local area network, allowing for the manipulation of traffic flow to simulate both LAN and WAN configurations. In particular, the LAN configuration permits a throughput of 1.25 GBps with a latency of 0.25 ms, whereas the WAN configuration allows for an average throughput of 12.5 MBps and a latency of 40 ms. These settings align with those employed in the state-of-the-art work [31] and closely resemble the settings used in the state-of-the-art work [3]. The prime number p for the finite field $\mathbb{F}_p$ is set to $2^{31}-1$. If not specifically stated, the parameter q for geometric distribution is set to $1 \cdot 10^{-5}$, and the element domain $\mathcal{D}$ is set to $\{1, \ldots, 1 \cdot 10^8\}$, i.e., $|\mathcal{D}| = 1 \cdot 10^8$. We consider the number of set owners, $N \in \{20, 40, 60, 80, 100\}$ and set size, $M \in \{2^{16}, 2^{18}, 2^{20}, 2^{22}, 2^{24}\}$. We randomly sample sets from the specified element domain $\mathcal{D}$.

Prior works without DP leakage. In our experiments we compare the performance of FastPSC against two state-of-the-art PSI/PSU solutions [3, 31] under the outsourced setting. We demonstrate that, when differentially private leakage is allowed, FastPSC can achieve significant performance improvements over these solutions. The results for [31] are obtained by executing its official prototype implementation, available at https://github.com/ladnir/aby3. However, since the code for the protocol in [3] is not publicly available, we estimate its performance based on the best-reported results in [3]. Given the limitations of the works [19, 28] discussed in Section 2, we do not include them in our comparison. Additionally, since the prior works [18, 24, 35, 36] with DP leakage all focus on the multi-party setting, and their protocols cannot be easily adapted to the outsourcing setting (as analyzed in Section 1), we also exclude them from our comparison.

Table 1: Comparison of running time (in seconds).

			$N = 60, M =$					$M = 2^{20}, N =$				
	Setting	Protocol	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	20	40	60	80	100
PSI	LAN	[31]	12.8	56.6	281.1	1401.8	6661	91.5	195.1	281.1	383.9	485.1
		[3]	37.6	150.4	627.9	2511.6	10046.4	202.2	415	627.9	840.7	1053.6
		FastPSC	**8.3**	**25.6**	**87.1**	**338.7**	**1355.1**	**28.2**	**57.3**	**87.1**	**116**	**148.7**
	WAN	[31]	378.6	1410.1	2762.9	26130.3	48675	781.1	1581.1	2762.9	3211.9	4051.2
		[3]	>37.6	>150.4	>627.9	>2511.6	>10046.4	>202.2	>415	>627.9	>840.7	>1053.6
		FastPSC	**8.8**	**27**	**91.5**	**355.2**	**1420**	**29.7**	**60.2**	**91.5**	**121.8**	**156.1**
PSU	LAN	[31]	41.5	175	701.9	2832.1	11328.7	171.4	501.9	701.9	1011.7	1214
		[3]	>37.6	>150.4	>627.9	>2511.6	>10046.4	>202.2	>415	>627.9	>840.7	>1053.6
		FastPSC	**8.1**	**25.2**	**86.5**	**336**	**1353.2**	**27.1**	**56.2**	**86.5**	**114.1**	**145.2**
	WAN	[31]	331.6	1511.6	3512.2	24211.6	99186.4	1136.1	2191.2	3512.2	4250.9	6018.5
		[3]	>37.6	>150.4	>627.9	>2511.6	>10046.4	>202.2	>415	>627.9	>840.7	>1053.6
		FastPSC	**8.5**	**25.6**	**90.1**	**353.7**	**1411.4**	**28.5**	**58.9**	**90.1**	**116.2**	**152.4**

Note: The numbers in gray represent estimates based on the best possible results. $>^*$ indicates that the result in the corresponding case must exceed the value of *. This is because the PSI protocol of [3] is expected to perform slower in the WAN setting compared to the LAN setting. Furthermore, since the PSU protocol of [3] is built upon its PSI protocol, its PSU protocol is expected to be slower than its PSI protocol.

Table 2: Comparison of communication cost (in GB).

		$N = 60, M =$					$M = 2^{20}, N =$				
	Protocol	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	20	40	60	80	100
PSI	[31]	2.3	10.8	40.4	167.6	704.5	13.2	32.1	40.4	58.7	70.4
	[3]	54.2	162.7	767.8	1461.6	4393.2	279	573.4	767.8	1158.6	1454.3
	FastPSC	**0.5**	**1.2**	**3.8**	**14.4**	**56.6**	**1.3**	**2.6**	**3.8**	**5.1**	**6.4**
PSU	[31]	4	17.7	69.4	284.3	1144.6	22.6	40.1	69.4	92.5	120.6
	[3]	>54.2	>162.7	>767.8	>1461.6	>4393.2	>279	>573.4	>767.8	>1158.6	>1454.3
	FastPSC	**0.5**	**1.2**	**3.8**	**14.4**	**56.6**	**1.3**	**2.6**	**3.8**	**5.1**	**6.4**

9.2 Comparison with Baselines in PSI and PSU

The running time of PSI and PSU. Table 1 shows the comparison between FastPSC and the two baselines [3, 31] in terms of the running time for PSI and PSU. We can observe that FastPSC significantly outperforms both baselines in all experimental cases. Specifically, FastPSC achieves a speedup ranging from 1.5× to 52.2× compared to [31], and (at least) a speedup ranging from 4.2× to 7.4× compared to [3]. We can observe that the network configurations have little effect on the running time of FastPSC. This is attributed to FastPSC's low communication complexity, characterized by a round complexity of $O(1)$ and traffic complexity of $O(N \cdot (M + \frac{1}{q}))$.

The running time of [31] exhibits a sup-linear increase with the set size, whereas the running time of FastPSC demonstrates a sub-linear increase in the aspect. For instance, when the set size M grows from 2^{16} to 2^{24} (for PSI with $N = 60$ under the LAN setting), the running time of FastPSC only increases from 8.3 seconds to 1355.1 seconds. In contrast, the running time of [31] increases significantly from 12.8 seconds to 6661 seconds over the same set size expansion. Therefore, the advantages of FastPSC over [31] are expected to amplify further as the set size increases. Given that the protocol of [3] is slower than that of [31], the advantage of FastPSC over [3] will be even more pronounced.

The communication cost of PSI and PSU. Table 2 compares FastPSC and [3, 31] in terms of the communication cost for PSI and PSU. We can observe that FastPSC achieves a significantly lower communication cost compared to the baselines in all experimental cases. Specifically, FastPSC reduces 78%−98% of the communication cost of [31] and 99% of the communication cost of [3].

9.3 Scalability of FastPSC

The running time of PSI and PSU. In Table 3, we present the running time of FastPSC while varying the parameter q for the geometric distribution and the domain size $|\mathcal{D}|$. It can be observed that the running time appears to be minimally affected by variations in domain size $|\mathcal{D}|$. This is attributed to the fact that only Algorithm 1 is pertinent to the domain size, and its running time constitutes only a minor portion of the total running time.

The communication cost of PSI and PSU. In Table 4, we present the communication cost of FastPSC. Note that since the domain size $|\mathcal{D}|$ has no impact on the communication cost, we do not present the communication cost for varying the domain size $|\mathcal{D}|$. We can observe that when the set size is small, the influence of the parameter q on the communication cost is significant. For example, the communication cost is 0.3 GB with $q = 5 \cdot 10^{-5}$, $N = 60$, and $M = 2^{16}$, whereas it is 0.8 GB with $q = 1 \cdot 10^{-6}$, $N = 60$, and $M = 2^{16}$. The communication cost has almost tripled. However, when the set size is large, the effect of parameter q on the communication cost is minimal. For example, with $N = 60$ and $M = 2^{24}$, as q decreases from $5 \cdot 10^{-5}$ to $1 \cdot 10^{-6}$, the communication cost only rises from 56.3 GB to 56.9 GB. The reason is that the parameter q only influences the number of dummies padded into each set. Therefore, when the set size is small, the size of the padded set will increase significantly compared to that of the original set.

Table 3: Running time (in seconds) of FastPSC for PSI and PSU.

	Setting		$N = 60, M =$ 2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	$M = 2^{20}, N =$ 20	40	60	80	100
PSI ($\lvert\mathcal{D}\rvert = 1 \cdot 10^8$)	LAN	$q = 5 \cdot 10^{-5}$	6.9	20	78.9	319.9	1347.7	27.4	56.2	78.9	102	139.6
		$q = 1 \cdot 10^{-5}$	8.3	25.6	87.1	338.7	1355.1	28.2	57.3	87.1	116	148.7
		$q = 5 \cdot 10^{-6}$	10.1	34.3	93.4	353.5	1375.3	28.9	60.5	93.4	130.7	161.1
	WAN	$q = 5 \cdot 10^{-5}$	7.3	21.1	83	336.1	1362.3	28.8	58.8	83	118.5	151.2
		$q = 1 \cdot 10^{-5}$	8.8	27	91.5	355.2	1420	29.7	60.2	91.5	121.8	156.1
		$q = 5 \cdot 10^{-6}$	11.2	36.2	98.2	370.2	1490.6	30.4	63.7	98.2	137.4	169.2
PSI ($q = 1 \cdot 10^{-5}$)	LAN	$\lvert\mathcal{D}\rvert = 5 \cdot 10^7$	8.2	24.1	84.2	328.9	1316.1	25.8	52.9	84.2	110.6	141.2
		$\lvert\mathcal{D}\rvert = 2 \cdot 10^8$	8.6	26.2	89.6	342.9	1393.1	29.1	56.0	89.6	116.5	157.1
	WAN	$\lvert\mathcal{D}\rvert = 5 \cdot 10^7$	8.5	26.3	88.6	346.1	1388.9	26.6	57.9	88.6	115.9	149.2
		$\lvert\mathcal{D}\rvert = 2 \cdot 10^8$	9.2	28	92.3	361.6	1452.6	31.6	61.5	92.3	124.4	168.6
PSU ($\lvert\mathcal{D}\rvert = 1 \cdot 10^8$)	LAN	$q = 5 \cdot 10^{-5}$	6.5	19.2	76.1	307.5	1301.8	26.3	51.9	76.1	99.7	136.5
		$q = 1 \cdot 10^{-5}$	8.1	25.2	86.5	336	1353.2	27.1	56.2	86.5	114.1	145.2
		$q = 5 \cdot 10^{-6}$	9.5	31.8	91.4	346.9	1369.3	28.2	58.7	91.4	128.1	155.3
	WAN	$q = 5 \cdot 10^{-5}$	6.8	19.7	80.2	321.9	1359.7	27.5	55.7	80.2	105.9	144.2
		$q = 1 \cdot 10^{-5}$	8.5	25.6	90.1	353.7	1411.4	28.5	58.9	90.1	116.2	152.4
		$q = 5 \cdot 10^{-6}$	10.6	35.1	95.5	361.1	1481.3	29.1	61.9	95.5	130.1	162.3
PSU ($q = 1 \cdot 10^{-5}$)	LAN	$\lvert\mathcal{D}\rvert = 5 \cdot 10^7$	7.8	23.9	83	322	1303.2	25.4	52.6	83	109.4	140.1
		$\lvert\mathcal{D}\rvert = 2 \cdot 10^8$	8.5	26.9	91.3	348.1	1398.1	28.9	59.9	91.3	121.5	151.5
	WAN	$\lvert\mathcal{D}\rvert = 5 \cdot 10^7$	8.3	24.8	87.5	343.3	1370.7	27.6	57.2	87.5	112.7	148.2
		$\lvert\mathcal{D}\rvert = 2 \cdot 10^8$	8.9	27.4	91.8	355.1	1433.5	29.4	60.7	91.8	121.6	159.6

Table 4: Communication cost (in GB) of FastPSC for PSI and PSU.

Setting ($\lvert\mathcal{D}\rvert = 1 \cdot 10^8$)	$N = 60, M =$ 2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	$M = 2^{20}, N =$ 20	40	60	80	100
$q = 5 \cdot 10^{-5}$	0.3	0.9	3.6	14.1	56.3	1.2	2.4	3.6	4.8	6
$q = 1 \cdot 10^{-5}$	0.5	1.2	3.8	14.4	56.6	1.3	2.6	3.8	5.1	6.4
$q = 5 \cdot 10^{-6}$	0.8	1.6	4.2	14.6	56.9	1.3	2.8	4.2	5.8	7.1

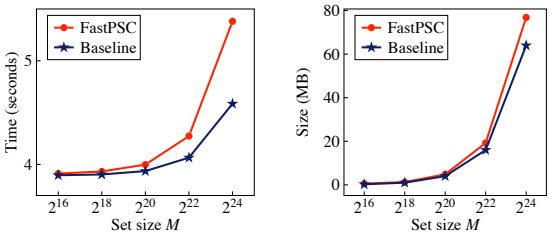

Figure 5: The cost of set secret sharing.

9.4 Evaluation of Set Secret Sharing

In Figure 5, we compare the performance of FastPSC with the baseline [31] in terms of the time of secretly sharing a set and the size of a secret-shared set, while varying the set size. Note that we do not compare with the baseline [3] since its encoding and secret sharing methods for the MAC are unclear. We can observe that despite FastPSC achieving stronger malicious security compared to the baseline [31], its cost remains nearly unchanged (e.g., 3.91 seconds vs 3.89 seconds). This is attributed to FastPSC's online MACs creation protocol, allowing computation nodes to authenticate the secret-shared sets, eliminating the need for each set owner to locally create MACs for their sets.

Table 5: Running time (in Seconds) of result set verification.

Domain size $\lvert\mathcal{D}\rvert$	$2 \cdot 10^7$	$4 \cdot 10^7$	$6 \cdot 10^7$	$8 \cdot 10^7$	$1 \cdot 10^8$
Intersection	0.36	0.75	1.13	1.59	1.83
Union	0.48	0.92	1.35	1.81	2.36

9.5 Evaluation of Result Set Verification

In Table 5, we show the performance of FastPSC in terms of result set parsing and verification. We can observe that the process is very fast. In particular, despite a large domain size of $\lvert\mathcal{D}\rvert = 1 \cdot 10^8$, the parsing and verification time for the intersection is a mere 1.83 seconds, and for the union, it is only 2.36 seconds.

10 CONCLUSION

We present FastPSC, a maliciously secure set computation system in the honest-majority four-party setting. FastPSC contributes new techniques to enable much faster secure set computation on multi-owner outsourced datasets over prior art, with differentially private leakage as the trade-off. In contrast to the state-of-the-art work [31] under the semi-honest threat model, FastPSC achieves better runtime by 1.5×-52.2× and cuts down communication cost by 78%-98%. In comparison to the state-of-the-art work [3] under the malicious threat model, FastPSC is 4.2×-7.4× faster and significantly reduces communication cost by 99%.

REFERENCES

[1] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. 2016. High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority. In *Proc. of ACM CCS*. 805–817.

[2] Toshinori Araki, Jun Furukawa, Kazuma Ohara, Benny Pinkas, Hanan Rosemarin, and Hikaru Tsuchida. 2021. Secure Graph Analysis at Scale. In *Proc. of ACM CCS*. 610–629.

[3] Gilad Asharov, Koki Hamada, Ryo Kikuchi, Ariel Nof, Benny Pinkas, and Junichi Tomida. 2023. Secure Statistical Analysis on Multiple Datasets: Join and Group-By. In *Proc. of ACM CCS*. 3298–3312.

[4] Jonas Böhler and Florian Kerschbaum. 2020. Secure multi-party computation of differentially private median. In *Proc. of USENIX Security*. 2147–2164.

[5] Jonas Böhler and Florian Kerschbaum. 2021. Secure multi-party computation of differentially private heavy hitters. In *Proc. of ACM CCS*. 2361–2377.

[6] Melissa Chase, Esha Ghosh, and Oxana Poburinnaya. 2020. Secret-Shared Shuffle. In *Proc. of ASIACRYPT*. 342–372.

[7] Harsh Chaudhari, Rahul Rachuri, and Ajith Suresh. 2020. Trident: Efficient 4PC Framework for Privacy Preserving Machine Learning. In *Proc. of NDSS*. 1–18.

[8] Kelong Cong, Radames Cruz Moreno, Mariana Botelho da Gama, Wei Dai, Ilia Iliashenko, Kim Laine, and Michael Rosenberg. 2021. Labeled PSI from homomorphic encryption with reduced computation and communication. In *Proc. of ACM CCS*. 1135–1150.

[9] Anders Dalskov, Daniel Escudero, and Marcel Keller. 2021. Fantastic four: Honest-Majority Four-Party secure computation with malicious security. In *Proc. of USENIX Security*. 2183–2200.

[10] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. 2012. Multiparty computation from somewhat homomorphic encryption. In *Proc. of CRYPTO*. 643–662.

[11] Emma Dauterman, Mayank Rathee, Raluca Ada Popa, and Ion Stoica. 2022. Waldo: A Private Time-Series Database from Function Secret Sharing. In *Proc. of IEEE S&P*. 2450–2468.

[12] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In *Proc. of NDSS*. 1–15.

[13] Cynthia Dwork. 2006. Differential Privacy. In *Proc. of ICALP*. 1–12.

[14] Saba Eskandarian and Dan Boneh. 2022. Clarion: Anonymous communication from multiparty shuffling protocols. In *Proc. of NDSS*. 1–20.

[15] Muhammad Faisal, Jerry Zhang, John Liagouris, Vasiliki Kalavri, and Mayank Varia. 2023. TVA: A multi-party computation system for secure and expressive time series analytics. In *Proc. of USENIX Security*. 5395–5412.

[16] Fireblocks. 2023. Remove the complexity of working with digital assets. online at https://www.fireblocks.com. [Online; Accessed 1-Jan-2025].

[17] Gayathri Garimella, Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. 2021. Oblivious key-value stores and amplification for private set intersection. In *Proc. of CRYPTO*. 395–425.

[18] Adam Groce, Peter Rindal, and Mike Rosulek. 2019. Cheaper private set intersection via differentially private leakage. *Proceedings on Privacy Enhancing Technologies* 2019, 3 (2019), 6–25.

[19] Yu Guo, Yuxin Xi, Yifang Zhang, Mingyue Wang, Shengling Wang, and Xiaohua Jia. 2024. Towards Efficient and Reliable Private Set Computation in Decentralized Storage. *IEEE Transactions on Services Computing* 17, 5 (2024), 2945–2958.

[20] Christoph Hagen, Christian Weinert, Christoph Sendner, Alexandra Dmitrienko, and Thomas Schneider. 2020. All the numbers are US: Large-scale abuse of contact discovery in mobile messengers. In *Proc. of NDSS*. 1–18.

[21] Thomas Humphries, Rasoul Akhavan Mahdavi, Shannon Veitch, and Florian Kerschbaum. 2022. Selective MPC: Distributed Computation of Differentially Private Key-Value Statistics. In *Proc. of ACM CCS*. 1459–1472.

[22] Mihaela Ion, Ben Kreuter, Ahmet Erhan Nergiz, Sarvar Patel, Shobhit Saxena, Karn Seth, Mariana Raykova, David Shanahan, and Moti Yung. 2020. On deploying secure computing: Private intersection-sum-with-cardinality. In *Proc. of EuroS&P*. 370–389.

[23] Yanxue Jia, Shi-Feng Sun, Hong-Sheng Zhou, Jiajun Du, and Dawu Gu. 2022. Shuffle-based private set union: Faster and more secure. In *Proc. of USENIX Security*. 2947–2964.

[24] Bailey Kacsmar, Basit Khurram, Nils Lukas, Alexander Norton, Masoumeh Shafieinejad, Zhiwei Shang, Yaser Baseri, Maryam Sepehri, Simon Oya, and Florian Kerschbaum. 2020. Differentially private two-party set operations. In *Proc. of IEEE EuroS&P*. 390–404.

[25] Shiva Prasad Kasiviswanathan, Homin K. Lee, Kobbi Nissim, Sofya Raskhodnikova, and Adam D. Smith. 2011. What Can We Learn Privately? *SIAM J. Comput.* 40, 3 (2011), 793–826.

[26] Lea Kissner and Dawn Song. 2005. Privacy-preserving set operations. In *Proc. of CRYPTO*. 241–257.

[27] Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. 2020. Efficient Oblivious Database Joins. *Proceedings of the VLDB Endowment* 13, 11 (2020), 2132–2145.

[28] Yin Li, Dhrubajyoti Ghosh, Peeyush Gupta, Sharad Mehrotra, Nisha Panwar, and Shantanu Sharma. 2021. Prism: private verifiable set computation over multi-owner outsourced databases. In *Proc. of ACM SIGMOD*. 1116–1128.

[29] Yehuda Lindell. 2017. How to Simulate It - A Tutorial on the Simulation Proof Technique. In *Tutorials on the Foundations of Cryptography*. 277–346.

[30] Meta. 2023. The value of secure multi-party computation. online at https://privacytech.fb.com/multi-party-computation/. [Online; Accessed 1-Jan-2025].

[31] Payman Mohassel, Peter Rindal, and Mike Rosulek. 2020. Fast Database Joins and PSI for Secret Shared Data. In *Proc. of ACM CCS*. 1271–1287.

[32] MPCVault. 2023. Multisig crypto wallet for business. online at https://mpcvault.com. [Online; Accessed 1-Jan-2025].

[33] Srinivasan Raghuraman and Peter Rindal. 2022. Blazing fast PSI from improved OKVS and subfield VOLE. In *Proc. of ACM CCS*. 2505–2517.

[34] Mike Rosulek and Ni Trieu. 2021. Compact and malicious private set intersection for small sets. In *Proc. of ACM CCS*. 1166–1181.

[35] Pinghui Wang, Yitong Liu, Zhicheng Li, and Rundong Li. 2024. An LDP Compatible Sketch for Securely Approximating Set Intersection Cardinalities. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–27.

[36] Pinghui Wang, Chengjin Yang, Dongdong Xie, Junzhou Zhao, Hui Li, Jing Tao, and Xiaohong Guan. 2023. An effective and differentially private protocol for secure distributed cardinality estimation. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–24.

[37] Zuan Wang, Xiaofeng Ding, Hai Jin, and Pan Zhou. 2022. Efficient secure and verifiable location-based skyline queries over encrypted data. *Proceedings of the VLDB Endowment* 15, 9 (2022), 1822–1834.

[38] Yuhan Wu, Siyuan Dong, Yi Zhou, Yikai Zhao, Fangcheng Fu, Tong Yang, Chaoyue Niu, Fan Wu, and Bin Cui. 2023. Kvsagg: Secure aggregation of distributed key-value sets. In *Proc. of ICDE*. 1775–1789.

A PROOF OF $(1-p)\cdot\mathbb{B}(x;y-1) < \mathbb{B}(x;y)$

Proof. Given the binomial distribution $\mathbb{B}(x;y-1)$, we have

$$\binom{y-1}{x}\cdot(p)^x\cdot(1-p)^{y-x-1}.$$

Given the property of combinatorial numbers: $\binom{y-1}{x} < \binom{y}{x}$, we have

$$\begin{aligned}\mathbb{B}(x;y-1) &< \binom{y}{x}\cdot(p)^x\cdot(1-p)^{y-x-1}\\ &= \binom{y}{x}\cdot(p)^x\cdot(1-p)^{y-x}\cdot\frac{1}{1-p}.\end{aligned}$$

Hence, $\mathbb{B}(x;y-1) < \frac{1}{1-p}\cdot\mathbb{B}(x;y) \Rightarrow (1-p)\cdot\mathbb{B}(x;y-1) < \mathbb{B}(x;y)$ □

B REDUCING PRIVACY LEAKAGE BY MORE COMPUTATION NODES

We prove that we can further diminish this lower bound given in Theorem 1 by introducing more computation nodes.

Theorem 4. *$\mathcal{M}$ is ε_L-DP with $\varepsilon_L = \ln\left(\max\left\{\frac{q}{1-q}\cdot|\mathcal{D}|+1, \frac{V}{V-3}\right\}\right)$, where V ($V \geq 4$) denotes the number of computation nodes and q denotes the parameter for geometric distribution.*

Proof. We define two multisets of elements owned by all set owners, $\mathcal{S}$ and $\mathcal{S}'$, as neighboring if they differ in at most the addition or deletion of one element. We represent the true histogram of the elements as the vector $c_{\mathcal{S}}$. Let $c_{\mathcal{S}}(e)$ represent the count of element $e \in \mathcal{D}$ on the input $\mathcal{S}$. Therefore, we have $\forall e \in \mathcal{D}, |c_{\mathcal{S}}(e) - c_{\mathcal{S}'}(e)| \leq 1$. We need to prove:

$$\frac{Pr[\mathcal{M}(\mathcal{S})=L]}{Pr[\mathcal{M}(\mathcal{S}')=L]} = \frac{\prod_{e\in\mathcal{D}} Pr[\mathcal{M}_e(\mathcal{S})=L_e]}{\prod_{e\in\mathcal{D}} Pr[\mathcal{M}_e(\mathcal{S}')=L_e]} \leq e^{\varepsilon_L}.$$

As $\mathcal{S}, \mathcal{S}'$ differ by the addition or deletion of an element, we have:

$$\frac{\prod_{e\in\mathcal{D}} Pr[\mathcal{M}_e(\mathcal{S})=L_e]}{\prod_{e\in\mathcal{D}} Pr[\mathcal{M}_e(\mathcal{S}')=L_e]} = \frac{Pr[\mathcal{M}_e(\mathcal{S})=L_e]}{Pr[\mathcal{M}_e(\mathcal{S}')=L_e]}, \tag{10}$$

Table 6: The lower bound of ε_L with different values of V

V	4	5	6	7	8	9	10
$\min \varepsilon_L$	1.39	0.92	0.69	0.56	0.47	0.41	0.36
V	11	12	13	14	15	16	17
$\min \varepsilon_L$	0.32	0.29	0.26	0.24	0.22	0.21	0.19

where e is the distinct element between $\mathcal{S}$ and $\mathcal{S}'$.

Note that with V computation nodes, a single computation node receives the secret share of an element with a probability of $3/V$. Therefore, the PMF of the count of element e observed by a single computation node can be modeled as the binomial distribution $\mathbb{B}_3(L_e = x; c_{\mathcal{S}}(e) + d_e = y)$:

$$\mathbb{B}_3(x, y) = \binom{y}{x} \cdot \left(\frac{3}{V}\right)^x \cdot \left(\frac{V-3}{V}\right)^{y-x},$$

where $c_{\mathcal{S}}(e) + d(e)$ is the count of true element e and its dummies in the padded sets. Then we can convert Eq. 10 to:

$$\frac{\sum_{R=N}^{\infty} \sum_{d_e=0}^{\infty} \mathbb{N}(R) \cdot \mathbb{B}_1(d_e; R) \cdot \mathbb{B}_3(L_e; c_{\mathcal{S}}(e) + d_e)}{\sum_{R=N}^{\infty} \sum_{d_e=0}^{\infty} \mathbb{N}(R) \cdot \mathbb{B}_1(d_e; R) \cdot \mathbb{B}_3(L_e; c_{\mathcal{S}'}(e) + d_e)}, \tag{11}$$

where $\mathbb{N}(\cdot)$ is defined by Eq. 3 and $\mathbb{B}_1(\cdot)$ is defined by Eq. 4. We now consider the two worst cases of under our assumption that $|c_{\mathcal{S}}(e) - c_{\mathcal{S}'}(e)| \leq 1$: (1) $c_{\mathcal{S}}(e) - c_{\mathcal{S}'}(e) = 1$; (2) $c_{\mathcal{S}'}(e) - c_{\mathcal{S}}(e) = 1$.

In the first case, we have Eq. 11 $\leq \frac{q}{1-q} \cdot |\mathcal{D}| + 1$. The process of proof is identical to that of Theorem 1. In the second case, we can convert Eq. 11 to:

$$\frac{\sum_{R=N}^{\infty} \sum_{d_e=0}^{\infty} \mathbb{N}(R) \cdot \mathbb{B}_1(d_e; R) \cdot \mathbb{B}_3(L_e; c_{\mathcal{S}'}(e) + d_e - 1)}{\sum_{R=N}^{\infty} \sum_{d_e=0}^{\infty} \mathbb{N}(R) \cdot \mathbb{B}_1(d_e; R) \cdot \mathbb{B}_3(L_e; c_{\mathcal{S}'}(e) + d_e)}. \tag{12}$$

Since $\mathbb{B}_3(L_e; c_{\mathcal{S}'}(e) + d_e - 1) \leq \frac{V}{V-3} \cdot \mathbb{B}_3(L_e; c_{\mathcal{S}}(e) + d_e)$, we have Eq. 12 $\leq \frac{V}{V-3}$. Combining the two cases gives

$$\frac{Pr[\mathcal{M}(\mathcal{S}) = L]}{Pr[\mathcal{M}(\mathcal{S}') = L]} \leq \max\left\{\frac{q}{1-q} \cdot |\mathcal{D}| + 1, \frac{V}{V-3}\right\} = e^{\varepsilon_L},$$

where $\varepsilon_L = \ln\left(\max\left\{\frac{q}{1-q} \cdot |\mathcal{D}| + 1, \frac{V}{V-3}\right\}\right)$. □

Table 6 shows the impact of varying V on the lower bound of ε_L.

C PROOF OF THEOREM 2

Proof. We first build simulator SM for the ideal world. We operate in the hybrid model [29], where invocations of sub-protocols can be replaced with the corresponding functionalities, provided that the sub-protocol is proven to be secure.

Simulator construction. Without loss of generality, let C_2 be the corrupted computation node. With access to the leakage $\mathcal{L}$, SM performs the following:

(1) On receiving $(\mathsf{Init}, \mathbb{F}_p, \mathcal{D})$ from $\mathcal{F}$: SM stores it locally and then forwards it to $\mathcal{A}$.

(2) On receiving (Colle, L) from $\mathcal{F}$: SM randomly samples $|\mathcal{U}|$ sets from $\mathcal{D}$, where $\mathcal{U}$ is the set of uncorrupted set owners and $|\mathcal{U}|$ is the size of the set $\mathcal{U}$. Each set's size can be obtained from L. SM assigns a random flag to each element in the sampled sets. SM samples RSS shares from $\mathbb{F}_p$ for each (element, flag) pair. SM then sends C_2's shares to $\mathcal{A}$.

(3) On receiving $(\mathsf{Set}, \mathsf{Op}, L)$: SM randomly samples two matrices (for the oblivious shuffle[2]) with the size of $2 \cdot \sum_{i \in \mathcal{U}} |\overline{\mathcal{S}}_i|$. Here, $|\overline{\mathcal{S}}_i|$ is the size of $\mathcal{S}_i$ in the view of C_2 and can be obtained from L. SM then forwards the two matrices to $\mathcal{A}$. SM forwards a random value (for the batch MAC checking) and the noisy count of each element (i.e., L) to $\mathcal{A}$. If any deviations are detected, SM aborts the process.

We now prove that the view generated by SM is indistinguishable from the real world through a sequence of hybrids $\mathcal{H}_0, \ldots, \mathcal{H}_3$.

Hybrid 0. We begin with the real world as our initial hybrid.

Hybrid 1. SM replaces the RSS shares for each (element, flag) pair with the output of the RSS simulator. Based on the security of the RSS scheme [1], $\mathcal{A}$ cannot distinguish between $\mathcal{H}_0$ and $\mathcal{H}_1$.

Hybrid 2. SM replaces the two matrices in the shuffle operations with the output of the pseudo-random function. Since $\mathcal{A}$ does not hold the random seed used in the generation of the two matrices, in the view of $\mathcal{A}$, the two matrices are random. Therefore, $\mathcal{A}$ cannot distinguish between $\mathcal{H}_1$ and $\mathcal{H}_2$.

Hybrid 3. SM replaces the RSS shares for the batch MAC checking with the output of the pseudo-random function. Additionally, since L is identical to what $\mathcal{A}$ receives in the real world, $\mathcal{A}$ cannot distinguish between $\mathcal{H}_2$ and $\mathcal{H}_3$. □

D PROOF OF THEOREM 3

Proof. Note that any error locally introduced by the malicious party cascades down in the subsequent computation across all parties due to the exchange of malformed messages [11]. This can be modeled as an equivalent additive error chosen by the malicious party in the messages it sends. Recall that in FastPSC, the detection of deviation behaviors occurs in two phases: the revelation of set elements at the computation nodes' side (Section 6.3) and the verification of the result set at the querier's side (Section 6.4).

For the revelation of set elements, our batch MAC checking can be formalized as $\sum(e_i \cdot \alpha - \hat{e}_i) \cdot \chi_i \stackrel{?}{=} 0$, where χ_i is a random value sampled from $\mathbb{F}_p$. Then we use ρ_i to represent the error introduced by the malicious computation node for e_i. Given that $\exists i, \rho_i \neq 0$ (indicating the presence of tampering), then passing the MAC checking requires: $\sum(e_i + \rho_i) \cdot \alpha \cdot \chi_i - \sum \hat{e}_i \cdot \chi_i + \Delta = 0$, where Δ is the corrective error the malicious computation node needs for the check to pass. By the analysis of [10], the success cheating probability is less than $\frac{2}{|\mathbb{F}_p|}$.

For the result set verification, our MAC checking can be formalized as $c_e \cdot \alpha - \hat{c}_e \stackrel{?}{=} 0, e \in \mathcal{D}$. Then we use ρ_e to represent the error introduced by the malicious computation node for $c_e, e \in \mathcal{D}$. Given that $\exists e, \rho_e \neq 0$ (indicating the presence of tampering), then passing the MAC checking requires: $(c_e + \rho_e) \cdot \alpha - \hat{c}_e + \Delta_e = 0$, where Δ_e is the corrective error the malicious computation node needs for the check to pass. This implies that successfully passing the check is equivalent to guessing α. However, as the malicious computation node lacks information about α, this occurs with a probability of only $\frac{1}{|\mathbb{F}_p|}$ for each $e \in \mathcal{D}$. Considering that the process aborts upon any deviation behavior, the MAC checks are interdependent. The overall probability of the malicious computation node passing

[2]Here, we only consider the execution of one oblivious shuffle operation for simplicity.

the check does not exceed $\max\{\frac{2}{|\mathbb{F}_p|}, \frac{1}{|\mathbb{F}_p|}, \cdots, \frac{1}{|\mathbb{F}_p|}\}$. Therefore, a cheating computation node can be caught with a probability of $1 - \frac{2}{|\mathbb{F}_p|}$. □