Multi-party Updatable Delegated Private Set Intersection

Aydin Abadi[†1] Changyu Dong[‡2] Steven J. Murdoch[§1] Sotirios Terzis[¶3]

[1] University College London
[2] Newcastle University
[3] University of Strathclyde

Abstract. With the growth of cloud computing, the need arises for Private Set Intersection protocols (PSI) that can let parties outsource the storage of their private sets and securely delegate PSI computation to a cloud server. The existing delegated PSIs have two major limitations; namely, they cannot support (1) efficient updates on outsourced sets and (2) efficient PSI among multiple clients. This paper presents "Feather", the *first* lightweight delegated PSI that addresses both limitations simultaneously. It lets clients independently prepare and upload their private sets to the cloud once, then delegate the computation an unlimited number of times. We implemented Feather and compared its costs with the state of the art delegated PSIs. The evaluation shows that Feather is more efficient computationally, in both update and PSI computation phases.

1 Introduction

Private Set Intersection (PSI) is an interesting protocol that lets parties compute the intersection of their private sets without revealing anything about the sets beyond the intersection [24]. PSI has various applications. For instance, it has been used in COVID-19 contact tracing schemes [22], remote diagnostics [18], and Apple's child safety solution to combat "Child Sexual Abuse Material" (CSAM) [13]. PSI has been considered by the "Financial Action Task Force" (FATF) as one of the vital tools for enabling collaborative analytics between financial institutions to strengthen "Anti-Money Laundering" (AML) and "Countering the Financing of Terrorism" (CFT) compliance [23].

Traditionally, PSIs have been designed for the setting where parties locally maintain their sets and jointly compute the sets' intersection. Recently, it has been a significant interest in the *delegated* PSIs that let parties outsource the storage of their sets to cloud computing which later can compute the intersection without being able to learn the sets and their intersection. One of the reasons for this trend is that the cloud is becoming mainstream among individuals,

[†] aydin.abadi@ucl.ac.uk
[‡] changyu.dong@newcastle.ac.uk
[§] s.murdoch@ucl.ac.uk
[¶] sotirios.terzis@strath.ac.uk

businesses, and financial institutes. For instance, IDC's 2020 survey suggests that the banking industry is not only adopting but also accelerating the adoption of the cloud, based on its benefits proven in the market [45]. The cloud can serve as a hub that allows for large-scale storage and data analysis by pooling clients' data together, without the need for them to locally maintain the data, which lets them discover new knowledge that could provide fresh insights to their business.

However, there are two major limitations to the existing delegated PSIs; namely, *they cannot efficiently support (1) updates on outsourced private sets, and (2) PSI among multiple clients*. Particularly, they have been designed for static sets and do not let parties efficiently update their outsourced sets. For application areas involving large private sets frequently updated, like fintech (e.g., stock market trend analysis [47]), e-commerce (e.g., consumer behaviour prediction [48]), or e-health (e.g., cancer research on genomic datasets [10]), the cost of securely updating outsourced sets using these schemes is prohibitive; in particular, it is linear with the entire set's size, $O(c)$. Another limitation is that they cannot scale to multiple clients without sacrificing security or efficiency. Specifically, in the most efficient delegate PSI in [1], the cloud has to perform a high number of random polynomials' evaluations which leads to a performance bottleneck, when the number of clients is high. A PSI that supports more than two parties creates opportunities for much richer analytics than what is possible with two-party PSIs. For example, it can benefit (i) companies that wish to jointly launch an ad campaign and identify the target audience, (ii) multiple ISPs which have private audit logs and want to identify network attacks' sources, or (iii) the aforementioned Apple's solution in which different CSAM datasets are provided by distinct child safety organizations [8].

Our Contributions. In this paper, we:

- present Feather, the first multi-party delegated PSI that lets a client efficiently update its outsourced set by accessing only a tiny fraction of this set. The update in Feather imposes $O(d^2)$ computation cost, where d is a hash table's bin size, i.e., $d = 100$.
- implement Feather and make its source code public, in [2].
- perform a rigorous cost analysis of Feather. The analysis shows that (a) updates on a set of 2^{20} elements are over 1000 times, and (b) PSI's computations are over 2 times faster than the fastest delegated PSI. Moreover, during the PSI computation when two clients participate, Feather's cloud-side runtime is over 26 times faster than the cloud's runtime in the fastest delegated PSI and this gap would grow when the number of clients increases. In Feather, it only takes 4.7 seconds to run PSI with 1000 clients, where each client has 2^{11} elements.

Feather offers other features too; for instance, the cloud *learns nothing* about the sets and their intersection, each client can *independently* prepare its set, and can delegate the PSI computation an *unlimited* number of times. We define and prove Feather's security in the simulation-based paradigm.

2 Related Work

Since their introduction in [24], various PSIs have been designed. PSIs can be broadly divided into *traditional* and *delegated* ones. In *traditional* PSIs data owners interactively compute the result using their local data. So far, the protocol of Kolesnikov *et al.* in [39] is the fastest two-party PSI secure against a semi-honest/passive adversary. It relies on symmetric key operations and has a computation complexity linear with the set size, i.e., $O(c)$, where c is a set size. Recently, Pinkas *et al.* in [43] proposed an efficient PSI that is secure against a stronger (i.e., active) adversary, and has $O(c \log c)$ computation complexity. Recently, researchers propose two threshold PSIs in [13] that let the Apple server learn the intersection of CSAM and a user's set only if the intersection cardinality exceeds a threshold. These two PSIs involve $O(c)$ asymmetric key operations. Also, there have been efforts to improve the communication cost in PSIs, through homomorphic encryption and polynomial representation [9,17,20,27]. Recently, a new PSI has been proposed that achieves a better balance between communication and computation costs [19]. Also, researchers designed PSIs that let multiple (i.e., more than two) parties efficiently compute the intersection. The multi-party PSIs in [31,40] are secure against passive adversaries while those in [11,26,50] were designed to remain secure against active ones. To date, the protocols in [40] and [26] are the most efficient multi-party PSIs designed to be secure against passive and active adversaries respectively. The computation complexities of [40] and [26] are $O(c\xi^2 + c\xi)$ and $O(c\xi)$ respectively, where ξ is the number of clients. However, Abadi *et al.* [4] showed that the latter is susceptible to several attacks. The former uses inexpensive symmetric key primitives and performs well with a small number of clients, i.e., up to 15. But, as we will discuss, it imposes high costs when the number of clients is high.

Delegated PSIs use cloud computing for computation and/or storage, while preserving the privacy of the computation inputs and outputs from the cloud. They can be divided further into protocols that support *one-off* and *repeated* delegation of PSI computation. The former like [33,36,51] cannot reuse their outsourced encrypted data and require clients to re-encode their data locally for each computation. The most efficient such protocol is [33], which has been designed for the two-party setting and its computation complexity is $O(c)$. In contrast, the latter (i.e., repeated PSI delegation ones) let clients outsource the storage of their encrypted data to the cloud only once, and then with the data owners' consent run any number of computations.

Looking more closely at the repeated PSI delegation protocols, the ones in [41,44,52] are not secure, as illustrated in [1,5]. In contrast, the PSIs in [1,5,6,49] are secure. Those in [5,6,49] involve $O(c)$ asymmetric key operations. In these schemes, the entire set is represented as a polynomial outsourced to the cloud. The protocol in [1] is more efficient than the ones in [5,6,49] and involves only $O(c)$ symmetric key operations. It uses a hash table to improve the performance. However, all these four protocols have been designed for the two-party setting and only support static datasets. Even though the authors in [1,5,6] explain how their two-party protocols can be modified to support multi-party, the extensions are computationally expensive; they also (a) impose a bottleneck to the cloud, and (b) do not provide any empirical evaluation for their modified protocols. In these PSIs, for parties to update their sets and avoid serious data leakage, they need to locally re-encode their entire outsourced set that incurs high costs.

3 Preliminaries

In this section, we outline the primitives used in this paper. We present a notation table in Appendix A.

3.1 Pseudorandom Functions and Permutation

Informally, a pseudorandom function is a deterministic function that takes a key of length Λ and an input; and outputs a value indistinguishable from that of a truly random function. In this paper, we use two pseudorandom functions: $\mathtt{PRF} : \{0,1\}^{\Lambda} \times \{0,1\}^{*} \rightarrow \mathbb{F}_p$ and $\mathtt{PRF'} : \{0,1\}^{\Lambda} \times \{0,1\}^{*} \rightarrow \{0,1\}^{\Psi}$, where $|p| = \Omega$ and Λ, Ψ, Ω are the security parameters. In practice, a pseudorandom function can be obtained from an efficient block cipher [35].

A pseudorandom permutation, $\pi(k, \vec{\boldsymbol{v}})$, is a deterministic function that permutes the elements of a vector, $\vec{\boldsymbol{v}}$, pseudorandomly using a secret key k. In practice, Fisher-Yates shuffle algorithm [38] can permute a vector of m elements in time $O(m)$. Formal definitions of pseudorandom function and permutation can be found in [35].

3.2 Hash Tables

A hash table is an array of bins each of which can hold a set of elements. It is accompanied with a hash function. To insert an element, we first compute the element's hash, and then store the element in the bin whose index is the element's hash. In this paper, we set the table's parameters appropriately to ensure the number of elements in each bin does not exceed a predefined capacity. Given the maximum number of elements c and the bin's maximum size d, we can determine the number of bins, h, by analysing hash tables under the balls into the bins model [12]. In Appendix B, we explain how the hash table parameters are set.

3.3 Horner's Method

Horner's method [21] is an efficient way of evaluating polynomials at a given point, e.g., x_0. In particular, given a degree-n polynomial of the form: $\tau(x) = a_0 + a_1 x + a_2 x^2 + ... + a_n x^n$ and a point: x_0, one can efficiently evaluate the polynomial at the point iteratively from inside-out, in the following fashion:

$$\tau(x_0) = a_0 + x_0(a_1 + x_0(a_2 + ... + x_0(a_{n-1} + x_0 a_n)...)))$$

Evaluating a degree-n polynomial naively requires n additions and $\frac{(n^2+n)}{2}$ multiplications, whereas using Horner's method the evaluation requires only n additions and n multiplications. We use this method throughout the paper.

3.4 Bloom Filter

A Bloom filter [14] is a compact data structure that allows us to efficiently check an element membership. It is an array of m bits (initially all set to zero), that represents n elements. It is accompanied with k independent hash functions. To insert an element, all the hash values of the element are computed and their corresponding bits in the filter are set to 1. To check an element membership, all its hash values are re-computed and checked whether all are set to 1 in the filter. If all the corresponding bits are 1, then the element is probably in the filter; otherwise, it is not. In Bloom filters it is possible that an element is not in the set, but the membership query indicates it is, i.e., false positives. In this work, we ensure the false positive probability is negligible, e.g., 2^{-40}. In Appendix C, we explain how the Bloom filter parameters can be set.

3.5 Representing Sets by Polynomials

Freedman *et al.* in [24] put forth the idea of using a polynomial to represent a set elements. In this representation, set elements $S = \{s_1, ..., s_d\}$ are defined over a field, $\mathbb{F}_p$, and set S is represented as a polynomial of form: $\rho(x) = \prod_{i=1}^{d}(x - s_i)$, where $\rho(x) \in \mathbb{F}_p[X]$ and $\mathbb{F}_p[X]$ is a polynomial ring. Often a polynomial of degree d is represented in the "coefficient form" as: $\rho(x) = a_0 + a_1 \cdot x + ... + a_d \cdot x^d$. As shown in [37], for two sets $S^{(A)}$ and $S^{(B)}$ represented by polynomials $\rho^{(A)}$ and $\rho^{(B)}$ respectively, their product, i.e., polynomial $\rho^{(A)} \cdot \rho^{(B)}$, represents the set union, while their greatest common divisor, i.e., $gcd(\rho^{(A)}, \rho^{(B)})$, represents the set intersection. For two degree-d polynomials $\rho^{(A)}$ and $\rho^{(B)}$, and two degree-d random polynomials $\gamma^{(A)}$ and $\gamma^{(B)}$, it is proven in [37] that:

$$\theta = \gamma^{(A)} \cdot \rho^{(A)} + \gamma^{(B)} \cdot \rho^{(B)} = \mu \cdot gcd(\rho^{(A)}, \rho^{(B)}), \tag{1}$$

where μ is a uniformly random polynomial, and polynomial θ contains only information about the elements in $S^{(A)} \cap S^{(B)}$, and contains no information about other elements in $S^{(A)}$ or $S^{(B)}$. To find the intersection, one extracts θ's roots, which contain the roots of (i) random polynomial μ and (ii) the polynomial that represents the intersection, i.e., $gcd(\rho^{(A)}, \rho^{(B)})$. To distinguish errors (i.e., roots of μ) from the intersection, PSIs in [1,5,37] use a padding technique. In this technique, every element u_i in the set universe $\mathcal{U}$, becomes $s_i = u_i || \mathtt{G}(u_i)$, where $\mathtt{G}$ is a cryptographic hash function with sufficiently large output size. Given a field's arbitrary element, $s \in \mathbb{F}_p$, and $\mathtt{G}$'s output size, we can parse s into a and b, such that $s = a||b$ and $|b| = |\mathtt{G}(.)|$. Then, we check $b \stackrel{?}{=} \mathtt{G}(a)$. If $b = \mathtt{G}(a)$, then s is an element of the intersection; otherwise, it is not.

Polynomials can also be represented in the "point-value form". Specifically, a polynomial $\mathbf{p}(x)$ of degree d can be represented as a set of m ($m > d$) point-value pairs $\{(x_1, y_1), ..., (x_m, y_m)\}$ such that all x_i are distinct non-zero points and $y_i = \rho(x_i)$ for all i, $1 \leq i \leq m$. Polynomials in point-value form have been used previously in PSIs [1,27]. A polynomial in this form can be converted into coefficient form via polynomial interpolation, e.g., via Lagrange interpolation [7]. Usually, PSIs that rely on this representation assume that all x_i are picked from $\mathbb{F} \setminus \mathcal{U}$. Also, one can add or multiply two polynomials, in point-value form, by adding or multiplying their corresponding y-coordinates.

4 Feather: Multi-party Updatable Delegated PSI

In this section, we first outline Feather's model, followed by an overview of its three protocols: setup, update, and PSI computation. Then, we elaborate on each protocol.

4.1 An Overview of Feather's Definition

Similar to most PSIs, we consider the semi-honest adversaries; similar to the PSIs in [1,6,32], we assume that the adversaries do not collude with the cloud. However, all but one clients are allowed to collude with each other. Similar to the security model of searchable encryption [29,34], in our security model we let some information, i.e., the query and access patterns, be leaked to the cloud to achieve efficiency. Informally, we say the protocol is secure as long as the cloud does not learn anything about the computation inputs and outputs beyond the allowed leakage and clients do not learn anything beyond the intersection about the other clients' set elements. We formalise Feather's security in the simulation-based paradigm. We require the clients' and cloud's view during the execution of the protocol can be simulated given their input and output (as well as the leakage). We refer readers to Appendix D for a formal definition.

4.2 An Overview of Feather's Protocols

At a high level, Feather works as follows. In the setup, the cloud publishes a set of public parameters. Any time a client wants to outsource the storage of its set, it uses the parameters to create a hash table. It inserts its set's elements to the hash table's bins, encodes the bins' content such that the encoded bins leak no information. Next, it assigns random-looking metadata to each bin, and shuffles the bins and the metadata. It sends the shuffled hash table and metadata to the cloud. When the client wants to insert/delete an element to/from its outsourced set, it figures out to which bin the element belongs and asks the cloud to send only that bin to it. Then, the client locally updates that bin's content, encodes the updated bin, and sends it to the cloud. In the PSI computation phase, the result recipient client, i.e., client B, interacts with other clients' to have their permission. Those clients that want to participate in the PSI computation send a set of messages to the cloud and client B. Using the clients' messages, the cloud connects the clients' permuted bins with each other and then obliviously computes the sets intersection. It sends the result to client B which, with the assistance of other clients' messages, extracts the result. Figure 1 in Appendix E depicts the parties' interaction in Feather.

In Feather, we use various techniques to attain scalability and efficiency. For instance, by analysing the most efficient delegated PSI in [1], we identified a *performance bottleneck* that prevents this PSI to scale in the multi-party setting. Specifically, we observed that in this scheme, the cloud has to perform a high number of random polynomials' evaluations on the clients' behalf. To avoid this bottleneck, in Feather, each client locally evaluates its random polynomials and sends the result to the cloud, yielding a significant performance improvement on the cloud side. To attain efficiency, we (i) substitute previous schemes' padding technique with an efficient error detecting mechanism, (ii) use an efficient polynomial evaluation (i.e., Horner's) method, and (iii) utilise a novel combination of permuted hash tables, permutation mapping, labels, and resettable counters.

4.3 Feather Setup

In this section, we first explain the efficient error detecting technique and then present Feather's setup protocol.

An Efficient Error Detecting Technique. As we described in section 3.5, often in the PSIs that use the polynomial representation, during the setup, each set element is padded (with some values). This lets the result recipient distinguish actual set elements from errors. A closer look reveals that the minimum bit-size of the padding is $t+\epsilon$ (due to the union bound), where 2^t is the total number of roots and $2^{-\epsilon}$ is the maximum probability that at least one invalid root has a set element structure, e.g., $\epsilon \geq 40$. So, this padding scheme increases element size, and requires a larger field. This has a considerable effect on the performance of (all arithmetic operations in the field and) polynomial factorisation whose complexity is bounded by (i) the polynomial's degree and (ii) the logarithm of the number of elements in the field, i.e., $O(n^a \log_2 2^{|p|})$ or $O(n^a|p|)$, where $1 < a \leq 2$, n is polynomial's degree and $|p|$ is the field bit size [25].

We observed that to improve efficiency, the padding scheme can be replaced by Bloom filters. The idea is that each client generates a Bloom filter which encodes all its set elements, blinds, and then sends the blinded Bloom filter (BB) along with other data to the cloud. For PSI computation, the result recipient gets the result along with its *own* BB. After it extracts the result, i.e., polynomials' roots, it checks if the roots are already in the Bloom filter and only accepts those in it. The use of BB reduces an element size and requires a smaller field which improves the performance of all arithmetic operations in the field. Here, we highlight only the improvement during the factorisation, as it dominates the protocol's cost. After the modification, the factorisation complexity is reduced from $O(n^a(|p|+t+\epsilon))$ to $O(n^a|p|)$. For instance, for e elements, $e \in [2^{10}, 2^{20}]$, and the error probability 2^{-40}, we get a factor of 1.5-2.5 lower runtime, when $|p| \in [40, 100]$. In general, this improvement is at least a factor of 2, when $|p| \leq t+\epsilon$. The smaller element and field size *reduces the communication and cloud-side storage costs too.*

Feather Setup Protocol. Now, we present the setup protocol in Feather. Briefly, first the cloud generates and publishes a set of public parameters. Then, each client builds a hash table using these parameters. It maps its set elements into the hash table's bins and represents each bin's elements as a blinded polynomial. It assigns a Bloom filter to each bin such that a bin's Bloom filter encodes that bin's set elements. Next, it blinds each filter and assigns a unique label to each bin. It pseudorandomly permutes the (i) bins (containing the blinded polynomials), (ii) blinded Bloom filters, and (iii) labels. It sends the permuted: bins, blinded Bloom filters, and labels to the cloud. It can delete its local set at this point. Below, we present the setup protocol.

Cloud Setup: Sets c as an upper bound of sets' size and sets a hash table parameters, i.e., table's length: h, hash function: $\mathtt{H}$, and bin's capacity: d. It picks pseudorandom functions $\mathtt{PRF}$ (used to generate labels and masking) and $\mathtt{PRF}'$ (used to mask Bloom filters), and a pseudorandom permutation, π. It picks a vector $\vec{\boldsymbol{x}} = [x_1, .., x_n]$ of $n = 2d + 1$ distinct non-zero values. It publishes the parameters.
Client Setup: Let client $I \in \{A_1, ...A_\varsigma, B\}$ have set: $S^{(I)}$, $|S^{(I)}| < c$. Client I:

1. ***Gen. a hash table and Bloom filters:*** Builds a hash table $\mathtt{HT}^{(I)}$ and inserts its elements into it, i.e., $\forall s_i^{(I)} \in S^{(I)}$: $\mathtt{H}(s_i^{(I)}) = j$, then $s_i^{(I)} \rightarrow \mathtt{HT}_j^{(I)}$. If needed, it pads every bin to d elements (using dummy values). Then, for every j-th bin, it generates a polynomial representing the bin's elements: $\prod\limits_{l=1}^{d}(x - e_l^{(I)})$, and evaluates each polynomial at every element $x_i \in \vec{\boldsymbol{x}}$, where $e_l^{(I)}$ is either a set element or a dummy value. This yields a vector of n y-coordinates: $y_{j,i}^{(I)} = \prod\limits_{l=1}^{d}(x_i - e_l^{(I)})$, for that bin. It allocates a Bloom filter: $\mathtt{B}_j^{(I)}$ to bin $\mathtt{HT}_j^{(I)}$, and inserts only the set elements of the bin in the filter.
2. ***Blind Bloom filters:*** Blinds every Bloom filter, by picking a secret key: $bk^{(I)}$, extracting h pseudorandom values and using each value to blind each Bloom filter; i.e., $\forall j, 1 \leq j \leq h : \mathtt{BB}_j^{(I)} = \mathtt{B}_j^{(I)} \oplus \mathtt{PRF}'(bk^{(I)}, j)$, where $\oplus$ denotes XOR. Thus, a vector of blinded Bloom filters is computed: $\overrightarrow{\mathtt{BB}}^{(I)} = [\mathtt{BB}_1^{(I)}, ..., \mathtt{BB}_h^{(I)}]$.
3. ***Blind bins:*** To blind every $y_{j,i}^{(I)}$, it assigns a key to each bin by picking a master secret key $k^{(I)}$, and generating h pseudorandom keys: $\forall j, 1 \leq j \leq h$: $k_j^{(I)} = \mathtt{PRF}(k^{(I)}, j)$. Next, it uses each $k_j^{(I)}$ to generate n pseudorandom values $z_{j,i}^{(I)} = \mathtt{PRF}(k_j^{(I)}, i)$. Then, for each bin, it computes n blinded y-coordinates as follows: $\forall i, 1 \leq i \leq n : o_{j,i}^{(I)} = y_{j,i}^{(I)} + z_{j,i}^{(I)}$. Thus, d elements in each $\mathtt{HT}_j^{(I)}$ are represented as $\vec{\boldsymbol{o}}_j^{(I)} : [o_{j,1}^{(I)}, ..., o_{j,n}^{(I)}]$.
4. ***Gen. labels:*** Assigns a pseudorandom label to each bin, by picking a fresh key: $lk^{(I)}$ and then computing h values, i.e., $\forall j, 1 \leq j \leq h : l_j^{(I)} = \mathtt{PRF}(lk^{(I)}, j)$.
5. ***Shuffle:*** Pseudorandomly permutes the labeled hash table. To do that, it picks a fresh key, $pk^{(I)}$, and then calls π as follows: $\vec{\boldsymbol{o}}'^{(I)} = \pi(pk^{(I)}, \vec{\boldsymbol{o}}^{(I)})$, $\vec{\boldsymbol{l}}'^{(I)} = \pi(pk^{(I)}, \vec{\boldsymbol{l}}^{(I)})$, where $\vec{\boldsymbol{o}}^{(I)} = [\vec{\boldsymbol{o}}_1^{(I)}, ..., \vec{\boldsymbol{o}}_h^{(I)}]$ and $\vec{\boldsymbol{l}}^{(I)}$ contains the labels generated in step 4. Also, it pseudorandomly permutes $\overrightarrow{\mathtt{BB}}^{(I)}$ as: $\overrightarrow{\mathtt{BB}}'^{(I)} = \pi(pk^{(I)}, \overrightarrow{\mathtt{BB}}^{(I)})$.
6. ***Gen. resettable counters:*** Builds and maintains a vector: $\vec{\boldsymbol{c}}^{(I)}$ of counters $c_i^{(I)}$ initially zero, where each counter $c_i^{(I)}$ keeps track of the number of times a

bin $\mathtt{HT}_i^{(I)}$ in the outsourced hash table is retrieved by the client for an update. They will let the client efficiently regenerate the most recent blinding factors.

Outsourcing: every client I sends the permuted labeled hash table: $(\vec{\hat{\boldsymbol{o}}}^{(I)}, \vec{\hat{\boldsymbol{l}}}^{(I)})$ along with the permuted blinded Bloom filters: $\vec{\hat{\mathtt{BB}}}^{(I)}$ to the cloud.

4.4 Feather Update Protocol

In this section, we present the update protocol in Feather. Briefly, for client I to insert/delete an element, $s^{(I)}$, to/from its outsourced set, it asks the cloud to send to it a bin and that bin's blinded Bloom filter. To do that, it first determines to which bin the element belongs. It recomputes the bin's label and sends the label to the cloud which sends the bin and related blinded Bloom filter to it. Then, the client uses the counter and a secret key to remove the most recent blinding factors from the bin's content, applies the update, re-encodes the bin and filter. Next, it refreshes their blinding factors and sends the updated bin along with the updated filter to the cloud.

The efficiency of Feather's update protocol stems from three factors: (a) the ability of a client to (securely) update only a bin of its outsourced hash table, that leads to very low complexities, (b) the use of an efficient error detecting technique that yields communication and computation costs reduction, and (c) the use of the local counters that yields client-side storage cost reduction. Now, we explain the update protocol in detail.

1. ***Fetch a bin and its Bloom filter:*** Recomputes the label of the bin to which element $s^{(I)}$ belongs, by generating the bin's index: $\mathtt{H}(s^{(I)}) = j$, and computing the label: $l_j^{(I)} = \mathtt{PRF}(lk^{(I)}, j)$. It sends $l_j^{(I)}$ to the cloud which sends back the bin: $\vec{\boldsymbol{o}}_j^{(I)}$, and the blinded Bloom filter: $\mathtt{BB}_j^{(I)}$.
2. ***Unblind***: Removes the blinding factors from $\vec{\boldsymbol{o}}_j^{(I)}$ and $\mathtt{BB}_j^{(I)}$ as follows.
   a. ***Regen. blinding factors:*** To regenerate the blinding factors of the bin and its Bloom filter, it first regenerates the key for that bin, as $k_j^{(I)} = \mathtt{PRF}(k^{(I)}, j)$. Then, it uses $k_j^{(I)}$, $bk^{(I)}$, and $c_j^{(I)}$ to regenerate the bin's masking values:
      - If the bin has never been fetched (i.e., $c_j^{(I)} = 0$), then it computes
$$b_j^{(I)} = \mathtt{PRF}'(bk^{(I)}, j) \ \text{ and } \ \forall i, 1 \leq i \leq n : z_{j,i}^{(I)} = \mathtt{PRF}(k_j^{(I)}, i)$$
      - Otherwise (i.e., $c_j^{(I)} \neq 0$), it computes:
$$b_j^{(I)} = \mathtt{PRF}'(\mathtt{PRF}'(bk^{(I)}, j), c_j^{(I)}) \text{ and } \forall i, 1 \leq i \leq n : z_{j,i}^{(I)} = \mathtt{PRF}(\mathtt{PRF}(k_j^{(I)}, c_j^{(I)}), i)$$
   b. ***Unblind:*** Removes the blinding factors from the bin and its blinded Bloom filter, as follows. $\mathtt{B}_j^{(I)} = \mathtt{BB}_j^{(I)} \oplus b_j^{(I)}$, $\quad \forall i, 1 \leq i \leq n : y_{j,i}^{(I)} = o_{j,i}^{(I)} - z_{j,i}^{(I)}$. The result is a Bloom filter: $\mathtt{B}_j^{(I)}$ and a vector: $\vec{\boldsymbol{y}}_j^{(I)} = \{y_{j,1}^{(I)}, ..., y_{j,n}^{(I)}\}$.
3. ***Update the counter:*** Increments the corresponding counter: $c_j^{(I)} = c_j^{(I)} + 1$.
4. ***Update the bin's content:***
   - If update: **element insertion**

  - if the element, to be inserted, is not in the bin's Bloom filter, then it uses the n pairs of $(y^{(I)}_{j,i}, x_i)$ to interpolate a polynomial: $\psi_j(x)$ and considers valid roots of $\psi_j(x)$ as the set elements in that bin. Then, it generates a polynomial: $\prod_{m=1}^{d}(x - s'^{(I)}_m)$, where its roots consist of valid roots of $\psi_j(x)$, $s^{(I)}$, and some random elements to pad the bin. Next, it evaluates the polynomial at every $x_i \in \vec{\boldsymbol{x}}$. This yields $\vec{\boldsymbol{u}}^{(I)}_j = [u^{(I)}_{j,1}, ..., u^{(I)}_{j,n}]$. It discards $\mathtt{B}^{(I)}_j$ and builds a fresh one: $\mathtt{B}'^{(I)}_j$ encoding $s^{(I)}$ and valid roots of $\psi_j(x)$.
  - otherwise, i.e., if $s^{(I)} \in \mathtt{B}^{(I)}_j$, it sets $\vec{\boldsymbol{u}}^{(I)}_j = \vec{\boldsymbol{y}}^{(I)}_j$ and $\mathtt{B}'^{(I)}_j = \mathtt{B}^{(I)}_j$, where $\vec{\boldsymbol{y}}^{(I)}_j$ and $\mathtt{B}^{(I)}_j$ were computed in step 2.b. Note, in this case the element already exists in the set; therefore, the element is not inserted.
- If update: **element deletion**
  - if the element, to be deleted, is not in the bin's Bloom filter, then it sets $\vec{\boldsymbol{u}}^{(I)}_j = \vec{\boldsymbol{y}}^{(I)}_j$ and $\mathtt{B}'^{(I)}_j = \mathtt{B}^{(I)}_j$, where $\vec{\boldsymbol{y}}^{(I)}_j$ and $\mathtt{B}^{(I)}_j$ were computed in step 2.b. It means the element does not exist in the set, so no deletion is needed.
  - otherwise, if $s^{(I)} \in \mathtt{B}^{(I)}_j$, it uses pairs $(y^{(I)}_{j,i}, x_i)$ to interpolate a polynomial: $\psi_j(x)$. It constructs a polynomial: $\prod_{m=1}^{d}(x - s'^{(I)}_m)$, where its roots contains valid roots of $\psi_j(x)$, excluding $s^{(I)}$, and some random elements to pad the bin (if required). Then, it evaluates the polynomial at every $x_i \in \vec{\boldsymbol{x}}$. This yields $\vec{\boldsymbol{u}}^{(I)}_j = [u^{(I)}_{j,1}, ..., u^{(I)}_{j,n}]$. Also, it discards $\mathtt{B}^{(I)}_j$ and builds a fresh one: $\mathtt{B}'^{(I)}_j$ that encodes valid roots of $\psi_j(x)$ excluding $s^{(I)}$.

5. ***Blind***: Blinds the updated bin: $\vec{\boldsymbol{u}}^{(I)}_j$ and Bloom filter: $\mathtt{B}'^{(I)}_j$ as follows.
   a. generates fresh blinding factors:
   $$b^{(I)}_j = \mathtt{PRF}'(\mathtt{PRF}'(bk^{(I)}, j), c^{(I)}_j), \quad \forall i, 1 \leq i \leq n : z^{(I)}_{j,i} = \mathtt{PRF}(\mathtt{PRF}(k^{(I)}_j, c^{(I)}_j), i)$$
   b. blinds the bin's content and Bloom filter, using the fresh blinding factors.
   $$\mathtt{BB}^{(I)}_j = \mathtt{B}'^{(I)}_j \oplus b^{(I)}_j \quad \text{and} \quad \forall i, 1 \leq i \leq n : o^{(I)}_{j,i} = u^{(I)}_{j,i} + z^{(I)}_{j,i}$$
6. ***Send update query***: Sends $\vec{\boldsymbol{o}}^{(I)}_j = [o^{(I)}_{j,1}, ..., o^{(I)}_{j,n}], \mathtt{BB}^{(I)}_j$, $l^{(I)}_j$, and **"Update"** to the cloud which replaces the bin's and Bloom filter's contents with the new ones.

4.5 Feather PSI Computation Protocol

In this section, we present the PSI computation protocol in Feather. Note, to let the cloud compute PSI correctly, clients need to tell it how to combine the bins of their hash tables (each of which permuted under a different key) without revealing the bins' original order to the cloud. Also, as the blinding values of some of the bins get refreshed (when updated), each client needs to efficiently regenerate the most recent ones in PSI delegation and update phases. To address those issues, we use two novel techniques: *permutation mapping*, and *resettable counter*, respectively. Now, we outline how the clients delegate the computation to the cloud. When client B wants the intersection of its set and clients $A_\sigma \in \{A_1, ..., A_\xi\}$ sets, it sends a message to each client A_σ to obtain its permission. If client A_σ agrees, it generates two sets of messages (with the help of the counter), one for client B and one for the cloud. It sends messages that include unblinding vectors to client B, and a message that includes a permutation map to the cloud. The vectors help client B to unblind the cloud's response. The map lets the cloud associate client A_σ's bins to client B's bins. The cloud uses the clients' messages and the outsourced datasets to compute the result that contains a set of blinded polynomials. It sends them to client B which unblinds them and retrieves the intersection. Below, we present the PSI computation protocol in more detail.

1. **Computation Delegation:** It is initiated by B which is interested in the intersection.
   a. **Gen. a permission query:** Client B performs as follows.
      i. ***Regen. blinding factors:*** regenerates the most recent blinding factors: $\vec{\boldsymbol{z}}^{(B)} = [\vec{\boldsymbol{z}}^{(B)}_1, ..., \vec{\boldsymbol{z}}^{(B)}_h]$ (as explained in step 2.a. of the update). Then, it shuffles the vector: $\pi(pk^{(B)}, \vec{\boldsymbol{z}}^{(B)})$.
      ii. ***Mask blinding factors:*** to mask the shuffled vector, it picks a fresh temporary key: $tk^{(B)}$, uses it to allocate a key to each bin, i.e., $\forall g, 1 \leq g \leq h :$ $tk^{(B)}_g = \texttt{PRF}(tk^{(B)}, g)$. Then, using each key, it generates fresh pseudorandom values and uses them to blind the vector's elements, as below:

$$\forall g, 1 \leq g \leq h,\ \forall i, 1 \leq i \leq n : r^{(B)}_{g,i} = z^{(B)}_{a,i} + \texttt{PRF}(tk^{(B)}_g, i)$$

      Let $\vec{\boldsymbol{r}}^{(B)}_g = [r^{(B)}_{g,1}, ..., r^{(B)}_{g,n}]$. Note, $\vec{\boldsymbol{z}}^{(B)}_a$ at index a $(1 \leq a \leq h)$ in $\vec{\boldsymbol{z}}^{(B)}$ moved to index g after it was shuffled in the previous step.
      iii. ***Send off secret values:*** sends $lk^{(B)}, pk^{(B)}, \vec{\boldsymbol{r}}^{(B)} = [\vec{\boldsymbol{r}}^{(B)}_1, ..., \vec{\boldsymbol{r}}^{(B)}_h]$, and its id: $\mathit{ID}^{(B)}$, to every client A_σ. Also, it sends $tk^{(B)}$ to the cloud.
   b. **Grant the computation:** Each client $A_\sigma \in \{A_1, ..., A_\xi\}$ performs as follows.
      i. ***Gen. a mapping:*** computes a mapping vector that will allow the cloud to match client A_σ's bins to client B's ones. To do so, it first generates $\vec{\boldsymbol{M}}_{A_\sigma \rightarrow B}$ whose elements, m_g, are computed as follows.

$$\forall g, 1 \leq g \leq h : \quad l^{(A_\sigma)}_g = \texttt{PRF}(lk^{(A_\sigma)}, g), l^{(B)}_g = \texttt{PRF}(lk^{(B)}, g), m_g = (l^{(A_\sigma)}_g, l^{(B)}_g)$$

      It permutes the elements of $\vec{\boldsymbol{M}}_{A_\sigma \rightarrow B}$. This yields mapping vector $\vec{\hat{\boldsymbol{M}}}_{A_\sigma \rightarrow B}$
      ii. ***Regen. blinding factors:*** regenerates the most recent blinding factors: $\vec{\boldsymbol{z}}^{(A_\sigma)} = [\vec{\boldsymbol{z}}^{(A_\sigma)}_1, ..., \vec{\boldsymbol{z}}^{(A_\sigma)}_h]$ where each $\vec{\boldsymbol{z}}^{(A_\sigma)}_g$ contains n blinding factors. After that, it pseudorandomly permutes the vector as: $\pi(pk^{(A_\sigma)}, \vec{\boldsymbol{z}}^{(A_\sigma)})$.
      iii. ***Gen. random masks and polynomials:*** assigns n fresh random values: $a^{(A_\sigma)}_{g,i}$ and two random degree-d polynomials: $\omega^{(A_\sigma)}_g, \omega^{(B_\sigma)}_g$ to each bin: $\texttt{HT}_g$.
      iv. ***Gen. mask removers:*** generates $\vec{\boldsymbol{q}}^{(A_\sigma)}$ that will assist client B to remove the blinding factors from the result provided by the cloud. To do that, it first multiplies each element at position g in $\pi(pk^{(A)}, \vec{\boldsymbol{z}}^{(A)})$ and in $\vec{\boldsymbol{r}}^{(B)}$, by $\omega^{(A_\sigma)}_g$ and $\omega^{(B_\sigma)}_g$, respectively, i.e., $\forall g, 1 \leq g \leq h$ and $\forall i, 1 \leq i \leq n :$

$$v^{(A_\sigma)}_{g,i} = \omega^{(A_\sigma)}_{g,i} \cdot z^{(A_\sigma)}_{j,i} \quad \text{and} \quad v^{(B_\sigma)}_{g,i} = \omega^{(B_\sigma)}_{g,i} \cdot r^{(B_\sigma)}_{g,i} = \omega^{(B_\sigma)}_{g,i} \cdot (z^{(B)}_{a,i} + \texttt{PRF}(tk^{(B)}_g, i))$$

      Then, given permutation keys: $pk^{(A_\sigma)}$ and $pk^{(B_\sigma)}$, for each value $v^{(A_\sigma)}_{g,i}$ it finds its matched value: $v^{(B_\sigma)}_{e,i}$, such that the blinding factors $z^{(A_\sigma)}_{j,i}$ and $z^{(B)}_{j,i}$ of the two values belong to the same bin, $\texttt{HT}_j$. Specifically, for each

$v^{(A_\sigma)}_{g,i} = \omega^{(A_\sigma)}_{g,i} \cdot z^{(A_\sigma)}_{j,i}$ it finds $v^{(B_\sigma)}_{e,i} = \omega^{(B_\sigma)}_{e,i} \cdot (z^{(B)}_{j,i} + \texttt{PRF}(tk^{(B)}_{e}, i))$. Next, it combines and blinds the matched values, i.e., $\forall g, 1 \le g \le h$ and $\forall i, 1 \le i \le n$:

$$q^{(A_\sigma)}_{e,i} = -(v^{(A_\sigma)}_{g,i} + v^{(B_\sigma)}_{e,i}) + a^{(A_\sigma)}_{g,i} = -(\omega^{(A_\sigma)}_{g,i} \cdot z^{(A_\sigma)}_{j,i} + \omega^{(B_\sigma)}_{e,i} \cdot (z^{(B)}_{j,i} + \texttt{PRF}(tk^{(B)}_{e}, i))) + a^{(A_\sigma)}_{g,i}$$

v. ***Send values:*** sends $\vec{\boldsymbol{q}}^{(A_\sigma)} = [\vec{\boldsymbol{q}}_1^{(A_\sigma)}, ..., \vec{\boldsymbol{q}}_h^{(A_\sigma)}]$ to client B, where each $\vec{\boldsymbol{q}}_e^{(A_\sigma)}$ contains $q^{(A_\sigma)}_{e,i}$. It sends to the cloud $\textit{ID}^{(B)}$, $\textit{ID}^{(A_\sigma)}$, $\vec{\hat{\boldsymbol{M}}}_{A_\sigma \rightarrow B}$, the blinding factors: $a^{(A_\sigma)}_{g,i}$, **"Compute"**, and random polynomials' y-coordinates, i.e., all $\omega^{(A_\sigma)}_{g,i}$, $\omega^{(B_\sigma)}_{g,i}$.

2. ***Cloud-side Result Computation***: The cloud uses each mapping vector: $\vec{\hat{\boldsymbol{M}}}_{A_\sigma \rightarrow B}$ to match the bins' of clients A_σ and B. Specifically, for each e-th bin in $\vec{\hat{\boldsymbol{o}}}^{(B)}$ it finds g_σ-th bin in $\vec{\hat{\boldsymbol{o}}}^{(A_\sigma)}$, where both bins would have the same index, e.g., j, before they were permuted. Next, it generates the elements of $\vec{\boldsymbol{t}}_e$, i.e., $\forall e, 1 \le e \le h$ and $\forall i, 1 \le i \le n$:

$$t_{e,i} = (\sum_{\sigma=1}^{\xi} \omega^{(B_\sigma)}_{e,i}) \cdot (o^{(B)}_{e,i} + \texttt{PRF}(tk^{(B)}_{e}, i)) - \sum_{\sigma=1}^{\xi} a^{(A_\sigma)}_{g_\sigma,i} + \sum_{\sigma=1}^{\xi} \omega^{(A_\sigma)}_{g_\sigma,i} \cdot o^{(A_\sigma)}_{g_\sigma,i}$$

where $o^{(A_\sigma)}_{g_\sigma,i} \in \vec{\boldsymbol{o}}^{(A_\sigma)}_{g_\sigma} \in \vec{\hat{\boldsymbol{o}}}^{(A_\sigma)}$. It sends to B its blinded Bloom filters: $\vec{\hat{\texttt{BB}}}^{(B)}$ and result $\vec{\boldsymbol{t}} = [\vec{\boldsymbol{t}}_1, ..., \vec{\boldsymbol{t}}_h]$, where each $\vec{\boldsymbol{t}}_e$ has values $t_{e,i}$.

3. ***Client-side Result Retrieval***: Client B unblinds the permuted Bloom filters using the key $bk^{(B)}$. This yields a vector of permuted Bloom filters $\vec{\hat{\texttt{B}}}^{(B)}$. Then, it uses the elements of vectors $\vec{\boldsymbol{q}}^{(A_\sigma)}$ to remove the blinding from the result sent by the cloud, i.e., $\forall e, 1 \le e \le h$ and $\forall i, 1 \le i \le n$:

$$f_{e,i} = t_{e,i} + \sum_{\sigma=1}^{\xi} q^{(A_\sigma)}_{e,i} = (\sum_{\sigma=1}^{\xi} \omega^{(B_\sigma)}_{e,i}) \cdot (u^{(B)}_{j,i}) + \sum_{\sigma=1}^{\xi} \omega^{(A_\sigma)}_{g_\sigma,i} \cdot u^{(A_\sigma)}_{j,i}$$

Given vectors $\vec{\boldsymbol{f}}_e$ and $\vec{\boldsymbol{x}}$, it interpolates h polynomials: $\phi_e(x)$, for all e. Then, it extracts the roots of each polynomial. It considers the roots encoded in $\texttt{B}^{(B)}_e \in \vec{\hat{\texttt{B}}}^{(B)}$ as valid, and the union of all valid roots as the sets' intersection.

Theorem 1. *If* PRF *and* PRF′ *are pseudorandom functions, and π is a pseudorandom permutation, then Feather is secure in the presence of (a) a semi-honest cloud, or (b) semi-honest clients where all but one clients collude with each other.*

Proof outline. In the following, we provide an overview of the proof and we refer readers to Appendix G for an elaborated one. We conduct the security analysis for three cases where one of the parties is corrupt at a time. In *corrupt cloud* case, we show that given the leakage function output, i.e. query and access patterns, we can construct a simulator that produces a view indistinguishable from the one in the real model. The proof includes (1) simulating each client's outsourced data, (2) simulating clients queries (in PSI and update) by using query pattern (and access pattern in the update), and (3) arguing that the simulated values are indistinguishable from their counter-party in the real model, mainly based on the indistinguishability of pseudorandom functions and permutation outputs. In *corrupt client B* case, the proof includes (1) simulating each authoriser client's input and query, (2) simulating cloud's result, and (3) arguing that the simulated values are indistinguishable from their counter-party in the real model and it cannot learn anything beyond the intersection; the argument is based on the indistinguishability of randomised polynomials (in Sec. 3.5) and the indistinguishability of pseudorandom functions and permutation output. In *corrupt client* A_σ case, the proof comprises (1) simulating client B's queries and (2) arguing that the simulated values are indistinguishable from those in the real model, according to the indistinguishability of pseudorandom functions output. □

In Appendix F, we provide several remarks on Feather's protocols and explain why naive solutions cannot offer Feather's features. Also, in Appendix H, we present various extensions of Feather that outline how to: (a) reduce authorizers' storage space, (b) reset the counters, (c) further delegate grating the computation to a semi-honest third-party, and (d) further reduce communication cost.

5 Asymptotic Cost Analysis

In this section, we analyse and compare the complexities of Feather with those of delegated and traditional PSIs that support multi-client in the semi-honest model. Table 1 summarizes the results. We do not take the update cost of the traditional multi-party PSIs, i.e., in [31,40], into account, as they are designed for the cases where parties maintain locally their set elements and do not (need to) support data update. We present a full analysis in Appendix I.

Table 1: Comparison of the multi-party PSIs. Note, c: set cardinality upper bound, $\xi + 1$: total number of clients, $d = 100$, and all costs are in big O.

Property	Feather	[1]	[5]	[6]	[49]	[40]	[31]
Repeated Delegated PSI	✓	✓	✓	✓	✓	×	×
Supporting Multi-party	✓	✓	✓	✓	✓	✓	✓
Mainly Symmetric Key Primitives	✓	✓	×	×	×	✓	✓
Total PSI Comm. Complexity	$c\xi$	$c\xi$	$c\xi$	$c\xi$	$c\xi$	$c\xi^2$	$c\xi^2$
Total PSI Comp. Complexity	$c\xi + c$	$c\xi + c$	$c\xi + c^2$	$c\xi + c^2$	$c\xi + c^2$	$c\xi^2 + c\xi$	$c\xi^2 + c\xi$
Update Comm. Complexity	d	c	c	c	c	-	-
Update Comp. Complexity	d^2	c	c	c	c	-	-

5.1 Communication Complexity

In PSI Computation. Below, we analyse the protocols' communication cost during the PSI computation. Briefly, in Feather, client B's cost is $O(c\xi)$, each client A_σ's cost is $O(c)$, and the cloud's cost is $O(c)$. Thus, Feather's total communication cost during the computation of PSI is $O(c\xi)$. The cost of each PSI in

[1,5,6,49] is $O(c\xi)$, where the majority of the messages in [5,6,49] are the output of a public-key encryption scheme, whereas those in [1] and Feather are random elements of a finite field, that have much shorter bit-length. Also, each scheme's complexity in [31,40] is $O(c\xi^2)$.

In Update. In Feather, for a client to update its set, it sends to the cloud two labels, a vector of $2d+1$ elements, and a Bloom filter. So, in total its complexity is $O(d)$. The cloud sends a vector of $2d+1$ elements and a Bloom filter to the client that costs $O(d)$. Therefore, the update in Feather imposes $O(d)$ communication cost. The protocols in [1,5,6,49] offer no efficient update mechanism. Therefore, for a client to securely update its set, it has to download and locally update the entire set, which costs $O(c)$.

5.2 Computation Complexity

In PSI Computation. Next, we analyse the schemes computation complexity during the PSI computation. First, we analyse Feather's complexity. In short, client B's and cloud's complexity is $O(c\xi + c)$ while each client A_σ's complexity is $O(c)$. During the PSI computation, the main operations that the parties perform are modular addition, multiplication, and polynomial factorization. Thus, Feather's complexity during the PSI computation is $O(c\xi + c)$. In the delegated PSIs in [5,6,49], the cost is dominated by asymmetric key operations and polynomial factorization. These protocols' cost is $O(c\xi + c^2)$. Moreover, the cost of running PSI in the delegated PSI in [1] is $O(c\xi + c)$. Now, we turn our attention to the traditional PSIs in [31,40]. Each PSI in [31,40] has $O(c\xi^2 + c\xi)$ complexity and involves mainly symmetric key operations.

In Update. In Feather, to update an element, a client (i) performs $O(d)$ modular additions and multiplications, (ii) interpolates a polynomial that costs $O(d)$, (iii) extracts a bin's elements that costs $O(d^2)$, and (iv) evaluates a polynomial which costs $O(d)$. So, the client's total cost is $O(d^2)$. To update a set element in the PSIs in [5,49], a client has to encode the element as a polynomial, evaluate the polynomial on $2c + 1$ points, and perform $O(c)$ multiplications. The cloud performs the same number of multiplications to apply the update. So, each protocol's update complexity is $O(c)$. In [6], the client has to download the entire set, remove blinding factors, and apply the change locally that costs $O(c)$. Although the PSI in [1] use a hash table, if a client updates a single bin, then the cloud would learn which elements are updated (with a non-negligible probability); Because the bins are in their original order and each bin's address is the hash value of an element in that bin. Thus, in [1], for a client to securely update its set, it has to locally re-encode the entire set that costs $O(c)$.

6 Concrete Cost Evaluation

In this section, we first explain how we choose the optimal parameters of a hash table. Then, we provide a concrete evaluation of three protocols: Feather and the PSIs in [1,40]. The reason we only consider [1,40] is that [40] is the fastest *traditional multi-party* PSI while [1] is the fastest *delegated* PSI among the PSIs studied in section 5. We consider protocols in the semi-honest model.

6.1 Choice of Parameters

In Feather, with the right choice of the hash table's parameters, the cloud can keep the overall costs optimal. In this section, we briefly show how these parameters can be chosen. As before, let c be the upper bound of the set cardinality, d be the bin size, and h be the number of bins. Recall, in Feather the overall cost depends on the product, hd, i.e., the total number of elements, including set elements and random values stored in the hash table. Also, the computation cost is dominated by factorizing h polynomials of degree $n = 2d + 1$. For the cloud to keep the costs optimal, given c, it uses Inequality 2 (in Appendix B) to find the right balance between parameters d and h, in the sense that the *cost of factorizing a polynomial of degree n is minimal, while hd is* close *to c*. At a high level, to find the right parameters, we take the following steps. First, we measure the average time, t, taken to factorize a polynomial of degree n, for different values of n. Then, for each c, we compute h for different values of d. Next, for each d we compute ht, after that for each c we look for minimal d whose ht is at the lowest level. After conducting the above experiments, we can see that the cloud can set $d = 100$ for all values of c. In this setting, hd is at most $4c$ and only with a negligibly small probability, 2^{-40}, a bin receives more than d elements. We present a full analysis in Appendix J.1.

6.2 Concrete Communication Cost Analysis

In PSI Computation. Below, we compare the three PSIs' concrete communication costs during the PSI computation. Briefly, Feather has 8-496 times lower cost than the PSI in [40], while it has 1.6-2.2 times higher cost than the one in [1], for 40-bit elements. The PSI's cost in [40] grows much faster than Feather's and the scheme in [1], when the number of clients increases. Feather has a slightly higher cost than the one in [1], as Feather lets each client A_σ send to the cloud $2hn$ y-coordinates of random polynomials yielding a significant computation *improvement.* Table 2 compares the three PSIs' cost. Table 5, in Appendix J.2, provides a detailed analysis of Feather's communication cost.

In Update. In Feather, a client downloads and uploads only one bin, that makes its cost of update 0.003 MB for all set sizes, when each element bit-size is 40. In [1], for a client to securely update its data, it has to download the entire set, locally update and upload it. Via this approach, the update's total communication cost, in MB, is in the range $[0.13, 210]$ when the set size is in the range $[2^{10}, 2^{20}]$ and each element bit-size is 40. Thus, Feather's communication cost is from 45 to 70254 times lower than [1].

Table 2: Concrete communication cost comparison (in MB)

Protocols	Elem. size	Set's Cardinality			Number of Clients				
		2^{12}	2^{16}	2^{20}	3	4	10	15	100
[40]	40, 64-bit	✓			24	45	300	679	30278
			✓		407	762	5015	11341	505658
				✓	6719	12571	82697	186984	8335520
[1]	40-bit	✓			0.8	1	2	4	29
			✓		18	25	62	93	625
				✓	300	400	1001	1501	10011
	64-bit	✓			1.3	1.7	4	6	43
			✓		28	37	94	141	941
				✓	452	602	1506	2260	15069
Feather	40-bit	✓			1	2	5	8	61
			✓		30	44	123	189	1311
				✓	494	705	1973	3028	20979
	64-bit	✓			2	3	9	14	97
			✓		48	69	196	301	2096
				✓	773	1111	3138	4828	33549

6.3 Concrete Computation Cost Analysis

In this section, we provide an empirical computation evaluation of Feather using a prototype implementation developed in C++. Feather's source code can be found in [2]. We compare the concrete computation cost of Feather with the two protocols in [1,40]. All experiments were run on a macOS laptop, with an Intel $i5$@2.3 GHz CPU, and 16 GB RAM. In Appendix J.3, we provide full detail about the system's parameters used in the experiment.

In PSI Computation. We first compare the runtime of Feather and the PSI in [1] in a two-client setting, as the latter was designed and implemented in this setting. Briefly, Feather is 2-2.5 times faster than the PSI in [1] (as Figure 6 in Appendix J.3 shows). The cloud-side runtime in Feather is 26-34 times faster than the one in [1]. Because Feather lets each client compute and send y-coordinates of random polynomials to the cloud, so the cloud does not need to re-evaluate them. Tables 6 and 7, in Appendix J.3, compare these PSIs' runtime in the setup and PSI computation respectively. Briefly, for a small number of clients, the performance of the PSI in [40] is better than Feather, e.g., about 40-4 times when the number of clients is 3- 15. But, the performance of the one in [40] gets *significantly* worse when the number of clients is large, e.g., 100-150; as its cost is quadratic with the number of clients. Thus, Feather outperforms the PSI in [40] when the number of clients is large. We provide a more detailed analysis in Appendix J.3. We also conducted experiments when a very large number of clients participate in Feather, i.e., up to 16000 clients. To provide a concrete value here, in Feather it takes 4.7 seconds to run PSI with 1000 clients where each client has 2^{11} elements. Table 9, in Appendix J.3, provides more detail.

In Update. Now, we compare the runtime of Feather and the PSI in [1] during the update. As the PSI in [1] does not provide a way for an update, we developed a prototype implementation of it that lets clients securely update their sets. The implementation's source code is in [3]. The update runtime of Feather is much faster than that of in [1]. The update runtime of the latter scheme, for 40-bit elements, grows from 0.07 to 27 seconds when the set size increases from 2^{10} to 2^{20}; whereas in Feather, the update runtime remains 0.023 seconds for all set sizes. Hence, the update in Feather is 3-1182 times faster than the one in [1]. Table 3 provides the update's runtime detailed comparison.

Table 3: The update runtime comparison between Feather and [1] (in sec.).

Protocols	Elem. size	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
[1]	40-bit	0.07	0.09	0.13	0.21	0.37	0.68	1.72	3.41	6.88	13.75	27.2
	64-bit	0.08	0.11	0.14	0.22	0.38	0.69	1.76	3.43	7.12	13.94	28.15
Feather	40-bit						$\leftarrow$ 0.023 $\rightarrow$					
	64-bit						$\leftarrow$ 0.035 $\rightarrow$					

7 Conclusion

Private set intersection (PSI) is an elegant protocol with numerous applications. Nowadays, due to cloud computing's growing popularity, there is a demand for an efficient PSI that can securely operate on multiple outsourced sets that are updated frequently. In this paper, we presented Feather. It is the first efficient delegated PSI that lets multiple clients (i) securely store their private sets in the cloud, (ii) efficiently perform data updates, and (iii) securely compute PSI on the outsourced sets. We implemented Feather and performed a rigorous cost analysis. The analysis indicates that Feather's performance during the update is over 10^3 times, and during PSI computation is over 2 times faster than the most efficient delegated PSI. Feather has low communication costs too.

Recently, it has been shown that the most efficient multi-party PSI in [26] supposed to be secure against active adversaries, suffers from serious issues. Hence, to fill the void, future research could investigate how to enhance Feather so it remains secure against *active* adversaries while preserving its efficiency.

Acknowledgments

Aydin Abadi was supported in part by REPHRAIN: The National Research Centre on Privacy, Harm Reduction and Adversarial Influence Online, under UKRI grant: EP/V011189/1. Steven J. Murdoch is supported by REPHRAIN and The Royal Society under grant UF160505. This work was also partially funded by EPSRC Doctoral Training Grant studentship and EPSRC research grants EP/M013561/2 and EP/N028198/1.

References

1. Abadi, A., Terzis, S., Metere, R., Dong, C.: Efficient delegated private set intersection on outsourced private datasets. IEEE Transactions on Dependable and Secure Computing (2018)
2. Abadi, A.: The implementation of multi-party updatable delegated private set intersection (2021), `https://github.com/AydinAbadi/Feather/tree/master/Feather-implementation`
3. Abadi, A.: The implementation of the update phase in efficient delegated private set intersection on outsourced private datasets (2021), `https://github.com/AydinAbadi/Feather/tree/master/Update-Simulation-code`
4. Abadi, A., Murdoch, S.J., Zacharias, T.: Polynomial representation is tricky: Maliciously secure private set intersection revisited. In: ESORICS (2021)
5. Abadi, A., Terzis, S., Dong, C.: O-PSI: delegated private set intersection on outsourced datasets. In: IFIP SEC (2015)
6. Abadi, A., Terzis, S., Dong, C.: VD-PSI: verifiable delegated private set intersection on outsourced private datasets. In: FC (2016)
7. Aho, A.V., Hopcroft, J.E.: The Design and Analysis of Computer Algorithms. Pearson Education India (1974)
8. Apple inc.: Security threat model review of Apple's child safety features. Online (2021), `https://www.apple.com/child-safety/pdf/Security_Threat_Model_Review_of_Apple_Child_Safety_Features.pdf`
9. Badrinarayanan, S., Miao, P., Raghuraman, S., Rindal, P.: Multi-party threshold private set intersection with sublinear communication. In: PKC (2021)
10. Baldi, P., Baronio, R., De Cristofaro, E., Gasti, P., Tsudik, G.: Countering gattaca: efficient and secure testing of fully-sequenced human genomes. In: CCS (2011)
11. Ben-Efraim, A., Nissenbaum, O., Omri, E., Paskin-Cherniavsky, A.: Psimple: Practical multiparty maliciously-secure private set intersection. IACR Cryptol. ePrint Arch. (2021)
12. Berenbrink, P., Czumaj, A., Steger, A., Vöcking, B.: Balanced allocations: the heavily loaded case. In: STOC (2000)
13. Bhowmick, A., Boneh, D., Myers, S., Talwar, K., Tarbe, K.: The apple PSI system. Online (2021), `https://www.apple.com/child-safety/pdf/Apple_PSI_System_Security_Protocol_and_Analysis.pdf`
14. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. (1970)
15. Boneh, D., Gentry, C., Halevi, S., Wang, F., Wu, D.J.: Private database queries using somewhat homomorphic encryption. In: ACNS (2013)
16. Bose, P., Guo, H., Kranakis, E., Maheshwari, A., Morin, P., Morrison, J., Smid, M.H.M., Tang, Y.: On the false-positive rate of bloom filters. Inf. Process. Lett. (2008)
17. Branco, P., Döttling, N., Pu, S.: Multiparty cardinality testing for threshold private set intersection. IACR Cryptol. ePrint Arch. (2020)
18. Brickell, J., Porter, D.E., Shmatikov, V., Witchel, E.: Privacy-preserving remote diagnostics. In: CCS (2007)
19. Chase, M., Miao, P.: Private set intersection in the internet setting from lightweight oblivious PRF. In: CRYPTO (2020)
20. Chen, H., Laine, K., Rindal, P.: Fast private set intersection from homomorphic encryption. In: ACM CCS (2017)

21. Dorn, W.S.: Generalizations of horner's rule for polynomial evaluation. IBM Journal of Research and Development (1962)
22. Duong, T., Phan, D.H., Trieu, N.: Catalic: Delegated PSI cardinality with applications to contact tracing. In: ASIACRYPT (2020)
23. Financial Action Task Force (FATF): Stocktake on data pooling, collaborative analytics and data protection (2021), `https://www.fatf-gafi.org/publications/digitaltransformation/documents/data-pooling-collaborative-analytics-data-protection.html`
24. Freedman, M.J., Nissim, K., Pinkas, B.: Efficient private matching and set intersection. In: EUROCRYPT (2004)
25. von zur Gathen, J., Panario, D.: Factoring polynomials over finite fields: A survey. J. Symb. Comput. (2001)
26. Ghosh, S., Nilges, T.: An algebraic approach to maliciously secure private set intersection. In: EUROCRYPT (2019)
27. Ghosh, S., Simkin, M.: The communication complexity of threshold private set intersection. In: CRYPTO (2019)
28. Goldreich, O.: The Foundations of Cryptography - Volume 2, Basic Applications. Cambridge University Press (2004)
29. Hahn, F., Kerschbaum, F.: Searchable encryption with secure and efficient updates. In: ACM CCS (2014)
30. Hazay, C., Venkitasubramaniam, M.: Scalable multi-party private set-intersection. In: PKC (2017)
31. Inbar, R., Omri, E., Pinkas, B.: Efficient scalable multiparty private set-intersection via garbled bloom filters. In: SCN (2018)
32. Kamara, S., Mohassel, P., Raykova, M.: Outsourcing multi-party computation. ePrint (2011)
33. Kamara, S., Mohassel, P., Raykova, M., Sadeghian, S.: Scaling private set intersection to billion-element sets. In: FC (2014)
34. Kamara, S., Papamanthou, C.: Parallel and dynamic searchable symmetric encryption. In: FC (2013)
35. Katz, J., Lindell, Y.: Introduction to Modern Cryptography. CRC Press (2007)
36. Kerschbaum, F.: Outsourced private set intersection using homomorphic encryption. In: ASIACCS (2012)
37. Kissner, L., Song, D.X.: Privacy-preserving set operations. In: CRYPTO (2005)
38. Knuth, D.E.: The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition. Addison-Wesley (1981)
39. Kolesnikov, V., Kumaresan, R., Rosulek, M., Trieu, N.: Efficient batched oblivious PRF with applications to private set intersection. In: CCS (2016)
40. Kolesnikov, V., Matania, N., Pinkas, B., Rosulek, M., Trieu, N.: Practical multi-party private set intersection from symmetric-key techniques. In: CCS (2017)
41. Liu, F., Ng, W.K., Zhang, W., Giang, D.H., Han, S.: Encrypted set intersection protocol for outsourced datasets. In: IC2E (2014)
42. Papamanthou, C., Tamassia, R., Triandopoulos, N.: Authenticated hash tables (2008)
43. Pinkas, B., Rosulek, M., Trieu, N., Yanai, A.: PSI from paxos: Fast, malicious private set intersection. In: EUROCRYPT (2020)
44. Qiu, S., Liu, J., Shi, Y., Li, M., Wang, W.: Identity-based private matching over outsourced encrypted datasets. IEEE Transactions on Cloud Computing (2018)
45. Silva, J.: Banking on the cloud: Results from the 2020 cloudpath survey (2020), `https://www.idc.com/getdoc.jsp?containerId=US45822120`

46. Stefanov, E., van Dijk, M., Shi, E., Fletcher, C.W., Ren, L., Yu, X., Devadas, S.: Path oram: an extremely simple oblivious ram protocol. In: CCS (2013)
47. Tsai, C.F., Hsiao, Y.C.: Combining multiple feature selection methods for stock prediction: Union, intersection, and multi-intersection approaches. Decis. Support Syst. (2010)
48. Varma Citrin, A., Sprott, D.E., Silverman, S.N., Stem Jr, D.E.: Adoption of internet shopping: the role of consumer innovativeness. Industrial management & data systems (2000)
49. Yang, X., Luo, X., Wang, X.A., Zhang, S.: Improved outsourced private set intersection protocol based on polynomial interpolation. Concurrency and Computation (2018)
50. Zhang, E., Liu, F., Lai, Q., Jin, G., Li, Y.: Efficient multi-party private set intersection against malicious adversaries. In: CCSW (2019)
51. Zhao, Y., Chow, S.S.M.: Can you find the one for me? privacy-preserving matchmaking via threshold PSI. IACR Cryptology ePrint Archive (2018)
52. Zheng, Q., Xu, S.: Verifiable delegated set intersection operations on outsourced encrypted data. In: IC2E (2015)

A Notations

In Table 4, we summarise the notations used in Feather's protocols.

Table 4: Notation table.

Setting	Symbol	Description
Generic	c	Set cardinality upper bound
	h	The total number of bins
	d	A bin's capacity
	n	$n = 2d + 1$
	$e^{(I)}$	Value e belongs to client I
	$\mathtt{HT}_j$	j-th bin of hash table: $\mathtt{HT}$
	$\mathtt{B}_j$	Bloom filter allocated to $\mathtt{HT}_j$
	$\mathtt{BB}_j$	Blinded $\mathtt{B}_j$
	$z_{j,i}$	The most recent i-th PR value assigned to $\mathtt{HT}_j$
	l_j	PR label of $\mathtt{HT}_j$
	$\vec{\hat{\boldsymbol{v}}}$	After (elements of) $\vec{\boldsymbol{v}}$ is permuted
	$\vec{\hat{\boldsymbol{o}}}^{(I)}$	Client I outsourced blinded y-coordinates
	$\vec{\hat{\boldsymbol{l}}}^{(I)}$	Client I outsourced labels
	$\overrightarrow{\hat{\mathtt{BB}}}^{(I)}$	Client I outsourced blinded $\mathtt{B}_j$'s
	$\vec{\boldsymbol{c}}$	A vector of resettable counters: c_i
	pk	π key used to permute a vector
	lk, k	$\mathtt{PRF}$ keys used to gen. labels, and blind y's
Setup	$s \to \mathtt{X}$	Element s is inserted to $\mathtt{X}$
	$S^{(I)}$	Client I's set
Update	b_j	$\mathtt{B}_j$'s most recent blinding factor
	$\mathtt{B}'_j$	Updated Bloom filter
	$\vec{\boldsymbol{u}}_j$	(Updated) y-coordinates of $\mathtt{HT}_j$
PSI Computation	g_σ	Client A_σ g-th bin in the permuted hash table
	$\vec{\boldsymbol{t}}$	Cloud result, vector of combined bins
	$\overrightarrow{\hat{\boldsymbol{M}}}_{A\to B}$	Permuted mapping vector
	m_g	An element of $\overrightarrow{\hat{\boldsymbol{M}}}_{A\to B}$
	$r^{(B)}_{g,i}$	Masked blinding factor
	ω_g	Random polynomial
	$\vec{\boldsymbol{q}}^{(A)}$	A vector of blinding factor removers: $q^{(A)}_{e,i}$
	$\vec{\boldsymbol{f}}_e$	An unblinded combined bin

B Hash Tables

In this paper, a hash table is utilised for two reasons; to achieve *efficiency* when (a) computing PSI, and (b) updating outsourced set. We set the table's parameters appropriately to ensure the number of elements in each bin does not exceed a predefined capacity. Given the maximum number of elements c and the bin's maximum size d, we can determine the number of bins by analysing hash tables under the balls into bins model [12].

Theorem 2. ***(Upper Tail in Chernoff Bounds)*** *Let X_i be a random variable defined as $X_i = \sum_{i=1}^{c} Y_i$, where $Pr[Y_i = 1] = p_i$, $Pr[Y_i = 0] = 1 - p_i$, and all Y_i are independent. Let the expectation be $\mu = \mathrm{E}[X_i] = \sum_{i=1}^{h} p_i$, then $Pr[X_i > d = (1+\sigma)\cdot\mu] < \left(\frac{e^{\sigma}}{(1+\sigma)^{(1+\sigma)}}\right)^{\mu}, \forall\sigma > 0$*

In this model, the expectation is $\mu = \frac{c}{h}$, where c is the number of balls and h is the number of bins. The above inequality provides the probability that bin i gets more than $(1+\sigma)\cdot\mu$ balls. Since there are h bins, the probability that at least one of them is overloaded is bounded by the union bound:

$$Pr[\exists i, X_i > d] \leq \sum_{i=1}^{h} Pr[X_i > d] = h \cdot \left(\frac{e^{\sigma}}{(1+\sigma)^{(1+\sigma)}}\right)^{\frac{c}{h}} \tag{2}$$

Thus, for a hash table of length $h = O(c)$, there is always an *almost constant* expected number of elements, d, mapped to the same bin with a high probability [42], e.g., $1 - 2^{-40}$.

C Bloom Filter

In this work, we use Bloom filters to let parties (in Feather) identify real set elements from errors. A Bloom filter [14] is a compact data structure for probabilistic efficient elements' membership checking. A Bloom filter is an array of m bits that are initially all set to zero. It represents n elements. A Bloom filter comes along with k independent hash functions. To insert an element, all the hash values of the element are computed and their corresponding bits in the filter are set to 1. To check an element's membership, all its hash values are re-computed and checked whether all are set to one in the filter. If all the corresponding bits are one, then the element is probably in the filter; otherwise, it is not. In Bloom filters false positives are possible, i.e. it is possible that an element is not in the set, but the membership query shows that it is. According to [16], the upper bound of the false positive probability is: $q = p^k(1 + O(\frac{k}{p}\sqrt{\frac{\ln m - k\ln p}{m}}))$, where p is the probability that a particular bit in the filter is set to 1 and calculated as: $p = 1 - (1 - \frac{1}{m})^{kn}$. The efficiency of a Bloom filter depends on m and k. The lower bound of m is $n\log_2 e \cdot \log_2 \frac{1}{q}$, where e is the base of natural logarithms, while the optimal number of hash functions is $\log_2 \frac{1}{q}$, when m is optimal. In this paper, we only use optimal k and m. In practice, we would like to have a predefined acceptable upper bound on false positive probability, e.g. $q = 2^{-40}$. Thus, given q and n, we can determine the rest of the parameters.

D Feather's Security Definition

In this section, we provide Feather's security definition. Similar to the majority of previous PSI's, we consider the semi-honest model. In particular, we consider a static semi-honest adversary who controls one of the parties at a time, i.e., non-colluding semi-honest adversaries [28,32]. For the sake of simplicity, we consider three kinds of party, cloud C and clients $A_\sigma \in \{A_1, ..., A_\xi\}$, and B engage in the protocol, where each A_σ authorizes the computation and client B is interested in the result. We assume parties use a secure communication channel. Similar to the security model of searchable encryption [34,29], in our security model we allow some information, i.e., query and access patterns, to be leaked to the cloud. This is an inevitable tradeoff if we want to retain efficiency, because to hide those patterns we would have to use primitives such as ORAM [46], which would make the protocol inefficient. Informally, we say the protocol is secure as long as the cloud does not learn anything about the computation inputs and output beyond the allowed leakage and clients do not learn anything beyond the intersection about the other clients' set elements. The leakage includes query and access patterns. The protocol involves two types of operations: set update: Upd, and delegated PSI computation: $\mathsf{D\text{-}PSI}$.

Query Pattern. Intuitively, in the protocol clients need to explicitly ask the cloud for a certain operation it wants to perform on its outsourced data. Therefore, the cloud learns, whether the client's i^{th} query is for update or PSI computation. In other words, the query pattern includes a list of clients' queries, that includes update and PSI computation queriers, sent to the cloud. Formally, we define the query pattern $\overrightarrow{\boldsymbol{T}}$ as a vector of strings where every element of the vector is defined as $T_i \in \{\mathsf{Upd}_t^{(I)}, \mathsf{PSI\text{-}Com}\}$, where $1 \leq t \leq \beta$, and value β is the total number of update queries issued by each client $I \in \{A_1, ..., A_\xi, B\}$. Also, $|\overrightarrow{\boldsymbol{T}}| = poly(\lambda) = \Upsilon$, where λ is a security parameter.

Access Pattern. In our protocol, a set is encoded as a hash table and each bin of the hash table is tagged with a unique *deterministic* label, in a form of a pseudorandom binary string of length l, where l is a security parameter. Without loss of generality, we assume in each update query only one element is inserted/removed to/from the set. Each update query always requires the client to send a label to the cloud, receive the bin (tagged with the label), and then rewrite the contents of the bin. Thus, in the update process, the cloud can see what part of the outsourced data is updated. But, it cannot associate that part with the sets' elements. In particular, given a sequence of client's queries, the cloud can see that a bin is updated but it does not learn the original address of the bin because the bins are pseudorandomly permuted. Also, it cannot figure out whether the update is an insertion or a deletion.

Definition 1. *(Access Pattern) Let $\mathtt{HT}^{(I)}$ be client I's hash table containing h bins where each bin, $\mathtt{HT}_i^{(I)}$, is tagged with a unique label $l_i^{(I)}$. Also, let $\overrightarrow{\boldsymbol{o}}'^{(I)} = \pi(k^{(I)}, \overrightarrow{\boldsymbol{o}}^{(I)})$ be shuffled data, where $k^{(I)}$ is a secret key and $\overrightarrow{\boldsymbol{o}}^{(I)} = [(\mathtt{HT}_1^{(I)}, l_1^{(I)}), ..., (\mathtt{HT}_h^{(I)}, l_h^{(I)})]$. The access pattern, for the shuffled data, induced by β-update queries is a symmetric binary matrix $\mathcal{M}^{(I)}$ such that for $1 \leq i, j \leq \beta$, the element in the i^{th} row and j^{th} column is 1, $\mathcal{M}_{i,j}^{(I)} = 1$, if the i^{th} query equals j^{th} query (i.e. both queries have the same label) and 0 otherwise (where $I \in \{A_1, ..., A_\xi, B\}$).*

The multi-party delegated PSI protocol, D-PSI, computes a function that maps the inputs to some outputs. We define this function as: $F : \bot \times \underbrace{2^{\mathcal{U}} \times ... \times 2^{\mathcal{U}}}_{\xi+1} \rightarrow \bot \times ... \times \bot \times f_{\cap}$ where $\bot$ denotes the empty string, $2^{\mathcal{U}}$ denotes the powerset of the set universe and $f_{\cap}$ denotes the set intersection function. For inputs $\bot, S^{(A_1)}, ..., S^{(A_\xi)}$ and $S^{(B)}$ belonging to $C, A_1, ..., A_\xi$ and B respectively, the function outputs nothing to $C, A_1, ..., A_\xi$, but outputs $f_{\cap}(S^{(A_1)}, ..., S^{(A_\xi)}, S^{(B)}) = S^{(A_1)} \cap ... \cap S^{(A_\xi)} \cap S^{(B)}$ to B. Note, for PSI computation we do not have any data leakage. In the security model, we define the leakage function as $\mathsf{leak}(\overrightarrow{\boldsymbol{o}}'^{(I)}, \overrightarrow{\boldsymbol{upd}}^{(I)}) = [\mathcal{M}^{(I)}, \overrightarrow{\boldsymbol{T}}]$ that captures precisely what is being leaked by the update operation. The function takes as input clients' outsourced data, and β update queries. It outputs two different types of information; namely, the access pattern (i.e., the matrix) and the query pattern of the clients. We say the protocol is secure if (1) nothing beyond the leakage is revealed to the cloud; (2) whatever can be computed by a client in the protocol can be obtained from its input and output only. This is formalized by the simulation paradigm. We require a client's *view* during the execution of D-PSI to be simulatable given its input and output. As one client's update pattern is not leaked to the other client, the scheme is secure as long as the PSI computation result does not leak any information to the client. Also, we require that the cloud's view of both operations, i.e., Feather = (Upd, D-PSI), can be simulated given the leakage. The party I's view on input tuple (x, y, z) is denoted by $\mathsf{VIEW}_I^{\mathbf{t}}(x, y, z)$, and equals $(w, r^{(i)}, m_1^{(i)}, ..., m_g^{(i)})$, where if $I \in \{A_1, ..., A_\xi, B\}$, then $\mathbf{t}$: D-PSI, if $I = C$, then $\mathbf{t}$: Feather, $w \in (x, y, z)$ is the input of I, $r^{(i)}$ is the outcome of i^{th} internal coin tosses and $m_j^{(i)}$ is the j^{th} message it received.

Definition 2. *Let* $\mathbf{S} = \{S^{(A_1)}, ..., S^{(A_\xi)}\}$ *and also Feather = (Upd, D-PSI) be the scheme defined above. We say that Feather is secure at the client-side in the presence of static semi-honest adversaries if there exist probabilistic polynomial-time algorithms* SIM_{A_σ} *and* SIM_B *that given the input and output of a client, can simulate a view that is computationally indistinguishable from the client's view in the protocol:*

$$\{\mathsf{SIM}_{A_\sigma}(S^{(A_\sigma)}, \bot)\}_{\mathbf{S}, S^{(B)}} \overset{c}{\equiv} \{\mathsf{VIEW}_{A_\sigma}^{\textsf{D-PSI}}(\bot, \mathbf{S}, S^{(B)})\}_{\mathbf{S}, S^{(B)}}$$

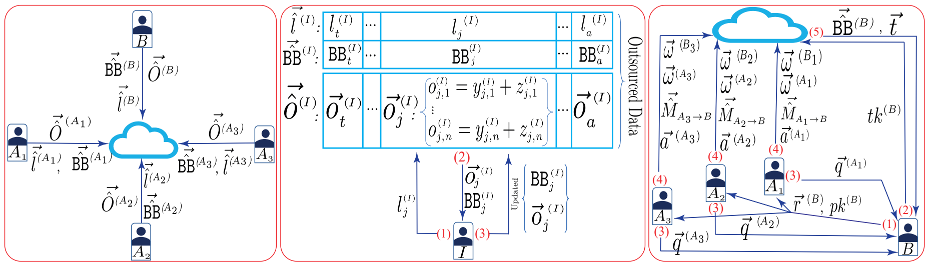
$$\{\mathsf{SIM}_B(S^{(B)}, f_{\cap}(\mathbf{S}, S^{(B)}))\}_{\mathbf{S}, S^{(B)}} \overset{c}{\equiv} \{\mathsf{VIEW}_B^{\textsf{D-PSI}}(\bot, \mathbf{S}, S^{(B)})\}_{\mathbf{S}, S^{(B)}},$$

where D-PSI was defined above and $A_\sigma \in \{A_1, ..., A_\xi\}$. *Also, Feather is secure, at the cloud-side, in the presence of static semi-honest adversaries if there exists probabilistic polynomial-time algorithm* SIM_C *that given the leakage function, can simulate a view that is computationally indistinguishable from the cloud's view in the protocol:*

$$\{\mathsf{SIM}_C^{\mathsf{leak}()}(\bot, \bot)\}_{\mathbf{S}, S^{(B)}} \overset{c}{\equiv} \{\mathsf{VIEW}_C^{\mathsf{Feather}}(\bot, \mathbf{S}, S^{(B)})\}_{\mathbf{S}, S^{(B)}}$$

E Feather's Protocols Workflow

Figure 1 outlines parties' interaction in Feather' protocols.



(a) Setup (b) Update (c) PSI Computation

Fig. 1: Workflow in Feather's protocols

F Further Remarks on Feather

In this section, we provide several remarks on Feather's update and PSI computation protocols.

Remark 1. In Feather, if multiple elements are updated at once, then multiple bins need to be accessed by a client (with a certain probability). This means the efficiency gap between Feather and the fasted delegated PSI in [1] in the update phase will reduce depending on the number of bins retrieved. To alleviate it, the client can use the lazy update technique, in the sense that it waits and buffers enough elements locally until some of them end up in the same bin. Then, it fetches the related bins and updates them. In this case, the client retrieves a lower number of bins and has less computation and communication overheads than the naive scheme where it updates the elements immediately. In general, when m elements are supposed to be updated in one go, the probability that a certain bin will receive less than a certain number of elements, say d', can be calculated via the lower tail in Chernoff bounds: $Pr[X_i < d' = (1-\sigma)\cdot\mu] < \left(\frac{e^{-\sigma}}{(1-\sigma)^{(1-\sigma)}}\right)^{\mu}$, where $\sigma \in (0,1]$ and $\mu = \frac{m}{h}$. The rest of the parameters are the same as those defined in Section 3.2.

Remark 2. In the following, we outline why the result's *correctness* holds, in step 3 in the Feather's PSI protocol. For the sake of simplicity, first we consider the two-client setting, where only client A_1 and B engage in the protocol. In this case, it holds that $f'_{e,i} = t_{e,i} + q^{(A_1)}_{e,i} = \omega^{(B_1)}_{e,i}\cdot u^{(B)}_{j,i} + \omega^{(A_1)}_{g_1,i}\cdot u^{(A_1)}_{j,i}$, where $u^{(B)}_{j,i}$ and $u^{(A_1)}_{j,i}$ are (y-coordinates of) polynomial representation of clients B and A_1 sets respectively, while $\omega^{(B_1)}_{e,i}$ and $\omega^{(A_1)}_{g_1,i}$ are (y-coordinates of) two random polynomials of degree d. As stated in Section 3.5, the result, $f'_{e,i}$, is (y-coordinates of) a polynomial that encodes the intersection of the two parties' sets. The same holds if there are ξ clients , i.e., $f''_{e,i} = \omega^{(B_\sigma)}_{e,i}\cdot u^{(B)}_{j,i} + \sum_{\sigma=1}^{\xi}\omega^{(A_\sigma)}_{g_\sigma,i}\cdot u^{(A_\sigma)}_{j,i}$. The only difference between $f''_{e,i}$ and $f_{e,i}$, in terms of presentation, is that in the former, $u^{(B)}_{j,i}$ is multiplied by only one random polynomial of degree d while in the latter, $u^{(B)}_{j,i}$ is multiplied by the sum of ξ polynomials of degree at most d. We know that the sum of ξ random polynomials of degree d is a random polynomial of degree d. Thus, the result in step 3, $f_{e,i}$, encodes the intersection of the parties' set elements.

Remark 3. Another reason that Feather's PSI scales better than [1] is that it removes a bottleneck from the cloud by requiring clients to send the evaluated random polynomials to it in step 1(b.)v. This relieves the cloud of re-evaluating them, and improves the cloud performance by up to 34 times. See Section J.3 for more detail.

Remark 4. Since the client, in the update phase, refreshes the blinding factors each time it retrieves a bin, the cloud cannot figure out whether it inserts or deletes an element. Also, as the original index of each bin is hidden from the cloud, it cannot learn which element has been updated in the bin.

Remark 5. In the protocol, the client does not need to recompute the hash table as long as each client's set cardinality remains smaller than the upper bound: c, irrespective of the number of updates performed on its outsourced data. Only in the case where a bin exceeds its capacity the client would need to recompute the table. However, as shown in section 3.2, given c, we can set the hash table parameters (i.e. total number of bins and bin capacity) in such a way that a bin overflows only with a negligibly small probability.

Remark 6. The bit-size of the field elements are different from the bit-size of each Bloom filter. Therefore, to mask them two different pseudorandom functions (i.e. $\mathtt{PRF}$ and $\mathtt{PRF}'$) with appropriate output sizes are needed.

Remark 7. One might be tempted to use an efficient traditional two-party PSI instead of Feather, such that clients store their data in the cloud, then for each PSI, they download the data and each of them runs a two-party PSI locally with the result recipient at a time. To record updates, a client encrypts the element (to be updated) and stores it in the cloud. In this setting, each time the client downloads the data to run PSI, it also downloads the update records, applies the change locally and uploads the entire data updated to the cloud. However, this approach has serious security issues in PSI computation, as it leaks more information than the intersection to the result recipient, e.g. pair-wise intersection. Also, this approach for update introduces additional cost: (a) the size of outsourced data grows with the number of update queries (even if the update is deletion) and (b) the client may store multiple copies of an item in the cloud, as there is no efficient way for it to figure out whether the set element exists in its outsourced set. Alternatively, one may use traditional multi-party PSI, to address the above security issues. In this case, a client might want to still use the above naive technique to update data. But, as shown in Sections 5 and J, this approach will: (a) have a much higher communication cost than Feather's, (b) have a slower run-time than Feather's, for a large number of clients, and (c) inherit the same issues stated above for the update.

G Feather Security Proof

In this section, we present Feather's security proof in the presence of static semi-honest adversaries. We conduct the security analysis for the three cases where one of the parties is corrupted at a time.

Theorem 3. *If* $\mathtt{PRF}$ *and* $\mathtt{PRF}'$ *are pseudorandom functions, and* π *is a pseudorandom permutation, then Feather is secure in the presence of (a) a semi-honest cloud, or (b) semi-honest clients where all but one clients collude with each other.*

Proof. We will prove the theorem by considering in turn the case where each of the parties has been corrupted. In each case, we invoke the simulator with the corresponding party's input and output. Our focus is on the case where party A_σ wants to engage in the computation of the intersection, i.e. it authorizes the computation. If party A_σ does not want to proceed with the protocol, the views can be simulated in the same way up to the point where the execution stops.

Case 1: Corrupted Cloud. We show that given the leakage function (output) we can construct a simulator SIM_C that can produce a view computationally indistinguishable from the one in the real model. In the real execution, the cloud's view is: $\mathsf{VIEW}_C^{\mathsf{Feather}}(\perp, \mathbf{S}, S^{(B)}) = \{\perp, r_C, \vec{\hat{\boldsymbol{o}}}, \vec{\hat{\boldsymbol{l}}}, \overrightarrow{\hat{\mathtt{BB}}}, \vec{\hat{\boldsymbol{o}}}^{(B)}, \vec{\hat{\boldsymbol{l}}}^{(B)}, \overrightarrow{\hat{\mathtt{BB}}}^{(B)}, (Q_1, ..., Q_\Upsilon), \perp\}$.

In the above view, $\mathbf{S}$ is authorizer clients' sets (i.e. $\mathbf{S} = \{S^{(A_1)}, ..., S^{(A_\xi)}\}$) and r_C is the outcome of internal random coins of the cloud. Moreover, $\vec{\hat{\boldsymbol{o}}} = \{\vec{\hat{\boldsymbol{o}}}^{(A_1)}, ..., \vec{\hat{\boldsymbol{o}}}^{(A_\xi)}\}$, $\vec{\hat{\boldsymbol{l}}} = \{\vec{\hat{\boldsymbol{l}}}^{(A_1)}, ..., \vec{\hat{\boldsymbol{l}}}^{(A_\xi)}\}$, and $\overrightarrow{\hat{\mathtt{BB}}} = \{\overrightarrow{\hat{\mathtt{BB}}}^{(A_1)}, ..., \overrightarrow{\hat{\mathtt{BB}}}^{(A_\xi)}\}$, where $\vec{\hat{\boldsymbol{o}}}^{(I)}$, $\vec{\hat{\boldsymbol{l}}}^{(I)}$, and $\overrightarrow{\hat{\mathtt{BB}}}^{(I)}$ are client I's outsourced permuted blinded y-coordinates, permuted labels, and permuted blinded Bloom filters respectively, where $I \in \{A_1, ..., A_\xi, B\}$. In the case where the *PSI computation* is delegated to the cloud, Q_b $(1 \leq b \leq \Upsilon)$ has the form: $Q_b = \{\vec{\boldsymbol{a}}, \vec{\boldsymbol{\omega}}^{(A)}, \vec{\boldsymbol{\omega}}^{(B)}, tk^{(B)}, \mathit{ID}, \mathit{ID}^{(B)}, \overrightarrow{\hat{\boldsymbol{M}}}, \textbf{“Compute”}\}$, otherwise (i.e. if client I sends an update query), $Q_b = \{\vec{\boldsymbol{o}}_g^{(I)}, l_g^{(I)}, \mathtt{BB}_g^{(I)}, \textbf{“Update”}\}$. Note that for PSI delegation, $\vec{\boldsymbol{a}} = \{\vec{\boldsymbol{a}}^{(A_1)}, ..., \vec{\boldsymbol{a}}^{(A_\xi)}\}, \vec{\boldsymbol{\omega}}^{(A)} = \{\vec{\boldsymbol{\omega}}^{(A_1)}, ..., \vec{\boldsymbol{\omega}}^{(A_\xi)}\}, \vec{\boldsymbol{\omega}}^{(B)} = \{\vec{\boldsymbol{\omega}}^{(B_1)}, ..., \vec{\boldsymbol{\omega}}^{(B_\xi)}\}$, $\mathit{ID} = \{\mathit{ID}^{(A_1)}, ..., \mathit{ID}^{(A_\xi)}\}$, and $\overrightarrow{\hat{\boldsymbol{M}}} = \{\overrightarrow{\hat{\boldsymbol{M}}}_{A_1 \to B}, ..., \overrightarrow{\hat{\boldsymbol{M}}}_{A_\xi \to B}\}$. Also, each $\vec{\boldsymbol{a}}^{(A_\sigma)} \in \vec{\boldsymbol{a}}$ contains $h \cdot n$ random elements: $a_{g,i}^{(A_\sigma)}$; each $\vec{\boldsymbol{\omega}}^{(A_\sigma)} \in \vec{\boldsymbol{\omega}}^{(A)}$ and $\vec{\boldsymbol{\omega}}^{(B_\sigma)} \in \vec{\boldsymbol{\omega}}^{(B)}$ contains $h \cdot n$ y-coordinates: $\omega_{g,i}^{(A_\sigma)}$ and $\omega_{g,i}^{(B_\sigma)}$, of random polynomials, respectively $(\forall g, i, 1 \leq i \leq n$ and $1 \leq g \leq h)$. Furthermore, each $\overrightarrow{\hat{\boldsymbol{M}}}_{A_\sigma \to B} \in \overrightarrow{\hat{\boldsymbol{M}}}$ contains h tuples of the form: $(l_g^{(A_\sigma)}, l_g^{(B)})$. Recall that when the query is update, $\vec{\boldsymbol{o}}_g^{(I)}$ contains n values blinded by fresh blinding factors, $l_g^{(I)}$ is a bin's label and $\mathtt{BB}_g^{(I)}$ is a Bloom filter blinded by a fresh blinding factor, where $1 \leq g \leq h$. Now we construct the simulator SIM_C in the ideal model.

1. **Simulate clients' outsourced data.**
   (a) ***Simulate hash tables:*** It uses the public parameters and the hash function to construct $\xi + 1$ hash tables: $\mathtt{HT}'^{(A_1)}, ..., \mathtt{HT}'^{(A_\xi)}, \mathtt{HT}'^{(B)}$. It fills each bin of the hash tables with n uniformly random values picked from $\mathbb{F}_p$. So, each bin $\mathtt{HT}_j'^{(I)}$ contains a vector $\vec{\boldsymbol{o}}_j'^{(I)}$ of n random values. Also, it creates an empty view and appends $\perp$ and uniformly random coin r'_C to the view.

(b) ***Simulate labels and blinded Bloom filters:*** Assigns a pseudorandom label to each bin $\mathtt{HT}'^{(I)}_j$. So, it picks fresh label-keys, $lk'^{(I)}$, and computes the labels as $\forall I, I \in \{A_1, ..., A_\xi, B\}$ and $\forall j, 1 \leq j \leq h : l'^{(I)}_j = \mathtt{PRF}(lk'^{(I)}, j)$. Also, it allocates a random bit string of length Ψ, as a blinded Bloom filter: $\mathtt{BB}'^{(I)}_j$, to each bin.

(c) ***Simulate permuted outsourced dataset:*** For each client, it pairs every bin with its label and blinded Bloom filter and then randomly permutes the pairs. To do that, for each client, it constructs vector: $[(\vec{\boldsymbol{o}}'^{(I)}_1, l'^{(I)}_1, \mathtt{BB}'^{(I)}_1)$ $, ..., (\vec{\boldsymbol{o}}'^{(I)}_h, l'^{(I)}_h, \mathtt{BB}'^{(I)}_h)]$. Then, it randomly permutes the vector. Next, it inserts each element $\vec{\boldsymbol{o}}'^{(I)}_g$ $(\forall g, 1 \leq g \leq h)$ of the permuted vector into $\vec{\boldsymbol{o}}'^{(I)}$. Also, it inserts each element $l'^{(I)}_g$ and $\mathtt{BB}'^{(I)}_g$ of the permuted vector into $\vec{\boldsymbol{l}}'^{(I)}$ and $\vec{\mathtt{BB}}'^{(I)}$ respectively. For each client, it appends the three vectors, (i.e., $\vec{\boldsymbol{o}}'^{(I)}$, $\vec{\boldsymbol{l}}'^{(I)}$ and $\vec{\mathtt{BB}}'^{(I)}$) to the view. Note that, the three vectors are the simulation of each client's outsourced data in the real world.

2. **Simulate labels' vector used for update queries.** As defined in Section D, for each client, a leakage: $[\mathcal{M}^{(I)}, \vec{\boldsymbol{T}}]$ includes access pattern: matrix $\mathcal{M}^{(I)}$, and query pattern: $\vec{\boldsymbol{T}}$. Recall that in the real world, a client fetches a bin by sending to the cloud the bin's label and it may happen multiple times. As a result, in total a vector of labels is sent to the cloud for the updates. In the ideal world, the simulator, given the matrix and the outsourced data (generated above), needs to simulate a labels vector: $\vec{\boldsymbol{v}}^{(I)}$ that will contain a set of labels $l'^{(I)} \in \vec{\boldsymbol{l}}'^{(I)}$, and the vector has the same access pattern as the one indicated by the matrix. The technique used here has a resemblance to the one utilised in searchable encryption schemes. To generate the vector, the simulator first constructs vector $\vec{\boldsymbol{v}}^{(I)}$ of zeros, where $|\vec{\boldsymbol{v}}^{(I)}| = \beta$. Then, for every row i $(1 \leq i \leq \beta)$ of the matrix $\mathcal{M}^{(I)}$, it performs the following:

(a) If there exists at least one element set to 1 in the row and if the i^{th} element in the vector $\vec{\boldsymbol{v}}^{(I)}$ is zero, then it finds a set G such that $\forall g \in G : \mathcal{M}^{(I)}_{i,g} = 1$. Next, it picks a label $l'^{(I)} \in \vec{\boldsymbol{l}}'^{(I)}$ and inserts it into all i^{th}, g^{th} positions of the vector $\vec{\boldsymbol{v}}^{(I)}$. The label must be distinct from the ones used for the previous rows i', where $i' < i$. Otherwise, if the i^{th} element in the vector is non-zero, it moves on to the next row.

(b) If all the elements in the row are zero and if the i^{th} element in the vector $\vec{\boldsymbol{v}}^{(I)}$ is zero, then it picks a label $l'^{(I)} \in \vec{\boldsymbol{l}}'^{(I)}$ and inserts it at position i^{th} of the vector, where the label is distinct from the ones used for the previous rows i', where $i' < i$. Otherwise, i.e., if i^{th} element in the vector is non-zero, it moves on to the next row.

3. **Simulate clients queries.** Uses the query pattern: $\vec{\boldsymbol{T}}$, to generate a set of queries depending on the *type* of each query in $\vec{\boldsymbol{T}}$, i.e. for PSI computation or update. Specifically, it checks $T_b \in \vec{\boldsymbol{T}}$ $(\forall b, 1 \leq b \leq \Upsilon)$, and performs as follows:

- ***Simulate clients queries for PSI computation:*** if $T_b = \mathsf{PSI\text{-}Com}$, then constructs $\vec{\boldsymbol{a}}' = \{\vec{\boldsymbol{a}}'^{(A_1)}, ..., \vec{\boldsymbol{a}}'^{(A_\xi)}\}$ where each $\vec{\boldsymbol{a}}'^{(A_\sigma)}$ contains n random elements: $a'^{(A_\sigma)}_{g,i}$ of the field, $\mathbb{F}_p$. Also, it constructs $\vec{\boldsymbol{\omega}}'^{(A)} = \{\vec{\boldsymbol{\omega}}'^{(A_1)}, ..., \vec{\boldsymbol{\omega}}'^{(A_\xi)}\}$

and $\vec{\boldsymbol{\omega}}'^{(B)} = \{\vec{\boldsymbol{\omega}}'^{(B_1)}, ..., \vec{\boldsymbol{\omega}}'^{(B_\xi)}\}$, where each $\vec{\boldsymbol{\omega}}'^{(A_\sigma)} \in \vec{\boldsymbol{\omega}}'^{(A)}$ and $\vec{\boldsymbol{\omega}}'^{(B_\sigma)} \in \vec{\boldsymbol{\omega}}'^{(B)}$ contains $h \cdot n$ y-coordinates: $\omega'^{(A_\sigma)}_{g,i}$ and $\omega'^{(B_\sigma)}_{g,i}$, of random d-degree polynomials, respectively. Then, it constructs $\vec{\hat{\boldsymbol{M}}}'$ that comprises ξ vectors: $\vec{\hat{\boldsymbol{M}}}'_{A_\sigma \rightarrow B}$ $(1 \leq \sigma \leq \xi)$ randomly permuted, such that each $\vec{\hat{\boldsymbol{M}}}'_{A_\sigma \rightarrow B}$ contains h tuples of the form: $(l'^{(A_\sigma)}_g, l'^{(B)}_g)$. It picks a random key: $tk'^{(B)}$. It computes $\mathit{ID} = \{\mathit{ID}^{(A_1)}, ..., \mathit{ID}^{(A_\xi)}\}$, $\mathit{ID}^{(B)}$ and **"Compute"**. Next, it constructs: $Q'_b = \{\vec{\boldsymbol{a}}', \vec{\boldsymbol{\omega}}'^{(A)}, \vec{\boldsymbol{\omega}}'^{(B)}, tk'^{(B)}, \mathit{ID}, \mathit{ID}^{(B)}, \vec{\hat{\boldsymbol{M}}}',$ **"Compute"**$\}$, and appends Q'_b to the view.

- ***Simulate a client's query for update:*** if $T_b = \mathsf{Upd}^{(I)}_t$, then sets: $Q'_b = \{\vec{\boldsymbol{o}}'^{(I)}, l'^{(I)}_t, \mathsf{BB}'^{(I)},$ **"Update"**$\}$, where $\vec{\boldsymbol{o}}'^{(I)}$ contains n fresh random values, $l'^{(I)}_t$ is the t^{th} element in the vector $\vec{\boldsymbol{v}}^{(I)}$, and $\mathsf{BB}'^{(I)}$ is a fresh random string. After that, it appends Q'_i to the view.

4. Appends $\perp$ to its view and outputs the view.

We are ready now to show why the two views are indistinguishable. Briefly, in the following, we argue that (a) the outsourced data, (b) clients' queries for PSI computation, and (c) clients' queries for the update, in real and ideal models are indistinguishable (with the use of the leakage).
Outsourced data indistinguishability: In both views, the input and output parts (i.e. $\perp$) are identical and the random coins are both uniformly random, so they are indistinguishable. In the real world, each vector $\vec{\boldsymbol{o}}^{(I)}_g \in \vec{\hat{\boldsymbol{o}}}^{(I)}$ $(1 \leq g \leq h)$ contains n values blinded with fresh pseudorandom values, the outputs of a pseudorandom function; each blinded Bloom filter: $\mathsf{BB}^{(I)}_j \in \vec{\hat{\mathsf{BB}}}^{(I)}$ is also masked with a fresh pseudorandom value of length Ψ. In the ideal world, each vector $\vec{\boldsymbol{o}}'^{(I)}_j \in \vec{\hat{\boldsymbol{o}}}'^{(I)}$ $(1 \leq j \leq h)$ contains n fresh random values sampled uniformly from the same field and each $\mathsf{BB}'^{(I)}_j \in \vec{\hat{\mathsf{BB}}}'^{(I)}$ contains a Ψ-bit long fresh random string. Since the blinded values and random values are not distinguishable, the elements of vectors $\vec{\hat{\boldsymbol{o}}}^{(I)}$ and $\vec{\hat{\boldsymbol{o}}}'^{(I)}$ as well as vectors $\vec{\hat{\mathsf{BB}}}^{(I)}$ and $\vec{\hat{\mathsf{BB}}}'^{(I)}$ are indistinguishable either. Also, labels $l^{(I)}_j \in \vec{\hat{\boldsymbol{l}}}^{(I)}$ and $l'^{(I)}_j \in \vec{\hat{\boldsymbol{l}}}'^{(I)}$ are the outputs of a pseudorandom function and they are indistinguishable. Therefore, the elements of vectors $\vec{\hat{\boldsymbol{l}}}^{(I)}$ and $\vec{\hat{\boldsymbol{l}}}'^{(I)}$ are indistinguishable. Furthermore, since a pseudorandom permutation is indistinguishable from a random permutation, permuted vectors $\vec{\hat{\boldsymbol{o}}}^{(I)}, \vec{\hat{\mathsf{BB}}}^{(I)}$ and $\vec{\hat{\boldsymbol{l}}}^{(I)}$, in the real model, and permuted vectors $\vec{\hat{\boldsymbol{o}}}'^{(I)}, \vec{\hat{\mathsf{BB}}}'^{(I)}$ and $\vec{\hat{\boldsymbol{l}}}'^{(I)}$, in the ideal model, are indistinguishable respectively.

Now we show that Q_b is indistinguishable from Q'_b, $(\forall b, 1 \leq b \leq \Upsilon)$. Note that, since sequence $Q'_1, ..., Q'_\Upsilon$, in the ideal model, is generated given the leakage function, its access and query patterns are identical to the access and query patterns of $Q_1, ..., Q_\Upsilon$, in the real model.
Indistinguishability of clients' queries in PSI: We consider the case where $T_b = \mathsf{PSI\text{-}Com}$. In this case, each $\vec{\boldsymbol{a}}^{(A_\sigma)} \in \vec{\boldsymbol{a}}$ contains $h \cdot n$ random elements, so does each vector $\vec{\boldsymbol{a}}'^{(A_\sigma)} \in \vec{\boldsymbol{a}}'$; therefore, elements of vectors $\vec{\boldsymbol{a}}$ and $\vec{\boldsymbol{a}}'$ are indistinguishable. Moreover, each vector $\vec{\boldsymbol{\omega}}^{(A_\sigma)} \in \vec{\boldsymbol{\omega}}^{(A)}, \vec{\boldsymbol{\omega}}^{(B_\sigma)} \in \vec{\boldsymbol{\omega}}^{(B)}, \vec{\boldsymbol{\omega}}'^{(A_\sigma)} \in \vec{\boldsymbol{\omega}}'^{(A)}$, and $\vec{\boldsymbol{\omega}}'^{(B_\sigma)} \in \vec{\boldsymbol{\omega}}'^{(B)}$ contains $h \cdot n$ y-coordinates of h random d-degree polynomials

picked from the same field and evaluated at the same set of values. Thus, vectors $\vec{\boldsymbol{\omega}}^{(A)}$ and $\vec{\boldsymbol{\omega}}^{(B)}$ in the real world are indistinguishable from vectors: $\vec{\boldsymbol{\omega}}'^{(A)}$ and $\vec{\boldsymbol{\omega}}'^{(B)}$ in the ideal world, respectively. Also, keys $tk^{(B)}$ and $tk'^{(B)}$ are random values, so they are indistinguishable. Messages $\mathsf{ID}, \mathsf{ID}^{(B)}$ and "Compute" are identical in both views. In the real model, each pair in each $\overrightarrow{\hat{\boldsymbol{M}}}_{A_\sigma \rightarrow B} \in \overrightarrow{\hat{\boldsymbol{M}}}$ has the form $(l_g^{(A_\sigma)}, l_g^{(B)})$, where $l_g^{(I)} \in \vec{\boldsymbol{l}}^{(I)}$ and each $l_g^{(I)}$ is a pseudorandom string $(1 \leq g \leq h$ and $I \in \{A_1, ..., A_\xi, B\})$. In the ideal model, each pair in randomly permuted vector: $\overrightarrow{\hat{\boldsymbol{M}}'}_{A_\sigma \rightarrow B} \in \overrightarrow{\hat{\boldsymbol{M}}'}$ has the form $(l'^{(A_\sigma)}_{g'}, l'^{(B)}_{g'})$ where $l'^{(I)}_{g'} \in \vec{\boldsymbol{l}}'^{(I)}$ and each $l'^{(I)}_{g'}$ is a pseudorandom string; so $\overrightarrow{\hat{\boldsymbol{M}}}$ and $\overrightarrow{\hat{\boldsymbol{M}}'}$ are indistinguishable. We conclude that Q_b is indistinguishable from Q'_b.

Indistinguishability of client's query in update: Now we move on to the case where $T_b = \mathsf{Upd}_t^{(I)}$. In the real model, $\vec{\boldsymbol{o}}_g^{(I)}$ contains n elements blinded with fresh pseudorandom values; while in the ideal model, $\vec{\boldsymbol{o}}'^{(I)}$ comprises n random values sampled uniformly from the same field. Since the random values and blinded values are indistinguishable, $\vec{\boldsymbol{o}}_g^{(I)}$ and $\vec{\boldsymbol{o}}'^{(I)}$ are indistinguishable. In the real model, $l_j^{(I)}$ is a pseudorandom string and is an element of vector $\vec{\boldsymbol{l}}^{(I)}$. In the ideal model, $l'^{(I)}_j$ is a pseudorandom string and is an element of vector $\vec{\boldsymbol{l}}'^{(I)}$. So, the labels have the same distribution and are indistinguishable. In the real model, blinded Bloom filter: $\mathtt{BB}_g^{(I)}$ has been masked with a fresh pseudorandom blinding factor of length Ψ; in the ideal model, $\mathtt{BB}'^{(I)}$ is a fresh random string of the same length; therefore, $\mathtt{BB}_g^{(I)}$ and $\mathtt{BB}'^{(I)}$ are indistinguishable. Also, message "Update" is identical in both models. Thus, Q'_b and Q_b are indistinguishable in this case too. From the above, we conclude that the views are indistinguishable.

Case 2: Corrupted Client B**.** In the real execution client B's view is defined as:

$$\mathsf{VIEW}_B^{\mathsf{D\text{-}PSI}}(\bot, \mathbf{S}, S^{(B)}) = \{S^{(B)}, r_B, \vec{\boldsymbol{q}}, \vec{\boldsymbol{f}}, \vec{\hat{\mathtt{B}}}^{(B)}, f_\cap(\mathbf{S}, S^{(B)})\}$$

where $\mathbf{S} = \{S^{(A_1)}, ..., S^{(A_\xi)}\}$, $\vec{\boldsymbol{q}} = \{\vec{\boldsymbol{q}}^{(A_1)}, ..., \vec{\boldsymbol{q}}^{(A_\xi)}\}$ and $\vec{\hat{\mathtt{B}}}^{(B)}$ is a set of Bloom filters pseudorandomly permuted. The simulator SIM_B who receives $pk^{(B)}, lk^{(B)}, S^{(B)}$ and $f_\cap(\mathbf{S}, S^{(B)})$ performs as follows:

1. **Simulate clients inputs.** Creates an empty view, appends $S^{(B)}$ and uniformly at random chosen coins r'_B to it. It chooses the following arbitrary sets: $\mathbf{S}' = \{S'^{(A_1)}, ..., S'^{(A_\xi)}\}$ and $S'^{(B)} = S^{(B)}$ such that $\mathbf{S}' \cap S'^{(B)} = f_\cap(\mathbf{S}, S^{(B)})$ and $|S'^{(A_\sigma)}|$, $|S'^{(B)}| \leq c$, where c is a set cardinality's upper bound and $1 \leq \sigma \leq \xi$. It uses the table parameters to construct $\xi + 1$ hash tables: $\mathtt{HT}'^{(A_1)}, ..., \mathtt{HT}'^{(A_\xi)}, \mathtt{HT}'^{(B)}$. Next, it maps the elements in $S'^{(I)}$ to the bins of $\mathtt{HT}'^{(I)}$, where $I \in \{A_1, ..., A_\xi, B\}$; i.e., $\forall I : \mathtt{H}(s_i'^{(I)}) = j$, then $s_i'^{(I)} \rightarrow \mathtt{HT}_j'^{(I)}$, where $s_i'^{(I)} \in S'^{(I)}$ and $1 \leq j \leq h$. It assigns to each bin $\mathtt{HT}_j'^{(B)} \in \mathtt{HT}'^{(B)}$ a Bloom filter: $\mathtt{B}_j'^{(B)}$ that encodes the set elements in the bin. Note, in total we have a set of Bloom filters: $\vec{\mathtt{B}}'^{(B)} = \{\mathtt{B}_1'^{(B)}, ..., \mathtt{B}_h'^{(B)}\}$. It builds a polynomial representing the d elements of each bin. If a bin contains less than d ele-

ments first it is padded. $\forall I$ and $\forall j, 1 \leq j \leq h$: $\tau'^{(I)}_j(x) = \prod_{m=1}^{d}(x - e^{(I)}_m)$, where $e^{(I)}_m \in \mathtt{HT}'^{(I)}_j$.

2. **Simulate cloud's result.** Assigns a random polynomial: $\omega'^{(A_\sigma)}_j$ of degree d to each bin $\mathtt{HT}'^{(A_\sigma)}_j$, where $1 \leq \sigma \leq \xi$. Also, it assigns ξ random polynomials of degree d to each bin $\mathtt{HT}'^{(B)}_j$. Next, it constructs vector $\vec{\boldsymbol{f}}' = [\vec{\boldsymbol{f}}'_1, ..., \vec{\boldsymbol{f}}'_h]$ where elements of each vector $\vec{\boldsymbol{f}}'_j$ are computed as follows. $\forall j, 1 \leq j \leq h$ and $\forall i, 1 \leq i \leq n$: $f'_{j,i} = (\sum_{\sigma=1}^{\xi} \omega'^{(B_\sigma)}_j(x_i)) \cdot (\tau'^{(B))}_j(x_i)) + \sum_{\sigma=1}^{\xi} \omega'^{(A_\sigma)}_j(x_i) \cdot \tau'^{(A_\sigma)}_j(x_i) - v_{j,i}$ where $v_{j,i}$ is a fresh random element of the field. Then, it permutes $\vec{\boldsymbol{f}}'$ as: $\vec{\boldsymbol{f}}'' = \pi(pk^{(B)}, \vec{\boldsymbol{f}}')$.
3. **Simulate authorizer clients queries.** Generates vector $\vec{\boldsymbol{q}}' = [\vec{\boldsymbol{q}}'_1, ..., \vec{\boldsymbol{q}}'_h]$ where each vector $\vec{\boldsymbol{q}}'_j$ contains n elements of the form: $q'_{j,i} + v_{j,i}$ such that $q'_{j,i}$ is a fresh random value and $v_{j,i}$ was generated in the previous step. Next, it permutes $\vec{\boldsymbol{q}}'$ as $\vec{\boldsymbol{q}}'' = \pi(pk^{(B)}, \vec{\boldsymbol{q}}')$.
4. It appends $\vec{\boldsymbol{q}}''$, $\vec{\boldsymbol{f}}''$, $\vec{\mathtt{B}}'' = \pi(pk^{(B)}, \vec{\mathtt{B}}'^{(B)})$, $f_\cap(\mathbf{S}, S^{(B)})$ to the view and outputs it.

Now we show that the two views are computationally indistinguishable. In short, in the following, we argue the (a) authoriser clients' queries, (b) cloud's result in real and ideal models are indistinguishable, and (c) the result recipient only learns the intersection from the final result.

Indistinguishability of authorizer clients queries: In the real model, the elements in $\vec{\boldsymbol{q}}_j$ are blinded with fresh pseudorandom values. In the ideal model, the elements in $\vec{\boldsymbol{q}}''_j$ are fresh random values drawn uniformly from the same field. Moreover, both $\vec{\boldsymbol{q}}$ and $\vec{\boldsymbol{q}}''$ are permuted in the same way. Hence, the vectors $\vec{\boldsymbol{q}}$ and $\vec{\boldsymbol{q}}''$ are computationally indistinguishable. Also, the entries $S^{(B)}$ and $\perp$ are identical in both views.

Indistinguishability of cloud's result: Both vectors $\vec{\mathtt{B}}''$ and $\vec{\mathtt{B}}$ are permuted in the same way and encode the elements of the same set, i.e., $S^{(B)}$, so they are indistinguishable. In the real model, the elements in $\vec{\boldsymbol{t}}_j$ are blinded with fresh pseudorandom values. In the ideal model, the elements in $\vec{\boldsymbol{f}}''_j$ are fresh random values drawn uniformly from the same field. Moreover, both vectors $\vec{\boldsymbol{t}}$ and $\vec{\boldsymbol{f}}''$ are permuted in the same way. Hence, the vectors $\vec{\boldsymbol{t}}$ and $\vec{\boldsymbol{f}}''$ are computationally indistinguishable.

Indistinguishability of the final result: In the real model, given each $\vec{\boldsymbol{f}}_j$, the adversary interpolates a $2d$-degree polynomial that has form: $\phi_j(x) = (\sum_{\sigma=1}^{\xi} \omega^{(B_\sigma)}_j(x)) \cdot (\tau^{(B)}_j(x)) + \sum_{\sigma=1}^{\xi} \omega^{(A_\sigma)}_{g_\sigma}(x) \cdot \tau^{(A_\sigma)}_j(x) = \mu_j \cdot gcd(\tau^{(A_1)}_j(x), ..., \tau^{(A_\xi)}_j(x), \tau^{(B)}_j(x)$, where polynomial $gcd(\tau^{(A_1)}_j(x)\ , ..., \tau^{(A_\xi)}_j(x), \tau^{(B)}_j(x))$ represents intersection of the sets in the corresponding bin, $\mathtt{HT}_j$. Similarly, in the ideal model, after unblinding each bin (i.e. summing $\vec{\boldsymbol{f}}''_j$ with $\vec{\boldsymbol{q}}''_j$ component-wise) the adversary uses the unblinded y-coordinates to interpolate a $2d$-degree polynomial: $\phi'_j(x)$ that has the form

$\phi'_j(x) = (\sum_{\sigma=1}^{\xi} \omega_j'^{(B_\sigma)}(x)) \cdot (\tau_j'^{(B)}(x)) + \sum_{\sigma=1}^{\xi} \omega_j'^{(A_\sigma)}(x) \cdot \tau_j'^{(A_\sigma)}(x) = \mu'_j \cdot gcd(\tau_j'^{(A_1)}(x)$ $, ..., \tau_j'^{(A_\xi)}(x), \tau_j'^{(B)}(x))$, where polynomial $gcd(\tau_j'^{(A_1)}(x)\ , ..., \tau_j'^{(A_\xi)}(x), \tau_j'^{(B)}(x))$ represents intersection of the sets in the corresponding bin: $\mathtt{HT}'_j$. As stated in section 3.5, μ_j and μ'_j are uniformly random polynomials and the probability that their roots represent set elements is negligible, thus $\phi_j(x)$ and $\phi'_j(x)$ only contain information about the set intersection and have the same distribution in both models [15,37]. Note, even if client B colludes with $\xi - 1$ clients, it cannot learn anything about the non-colluding client's set elements. Because, as proven in [37], polynomial $\sum_{\sigma=1}^{\xi} \omega_j^{(B_\sigma)}(x)$ is always a random polynomial even if only one of the polynomials, e.g. $\omega_j^{(B_\sigma)}(x)$, is contributed by an honest client and is a uniformly random polynomial not known to client B. Also, since the same hash table parameters are used, the same elements would reside in the same bins in both models, therefore polynomials $gcd(\tau_j^{(A_1)}(x)\ , ..., \tau_j^{(A_\xi)}(x), \tau_j^{(B)}(x))$ and $gcd(\tau_j'^{(A_1)}(x)$ $, ..., \tau_j'^{(A_\xi)}(x), \tau_j'^{(B)}(x))$ represent the set elements of the intersection for that bin. Also, the output: $f_\cap(\mathbf{S}, S^{(B)})$ is identical in both views. Thus, we conclude that the two views are computationally indistinguishable.

Case 3: Corrupted Client A_σ. In the real execution, the view of each client $A_\sigma \in \{A_1, ..., A_\xi\}$ is defined as: $\mathsf{VIEW}_{A_\sigma}^{\mathsf{D\text{-}PSI}}(\perp, \mathbf{S}, S^{(B)}) = \{S^{(A_\sigma)}, r_{A_\sigma}, lk^{(B)}, pk^{(B)}, \vec{\boldsymbol{r}}^{(B)},$ $\mathsf{ID}^{(B)}, \perp\}$. The simulator, SIM_A, who receives $S^{(A_\sigma)}$ performs as follows. It constructs an empty view, and adds $S^{(A_\sigma)}$ and uniformly at random chosen coins r'_{A_σ} to the view. It simulates client B's queries by picking two random keys $lk'^{(B)}$, $pk'^{(B)}$ and constructing $\vec{\boldsymbol{r}}'^{(B)} = [\vec{\boldsymbol{r}}_1'^{(B)}, ..., \vec{\boldsymbol{r}}_h'^{(B)}]$, where each vector $\vec{\boldsymbol{r}}_j'^{(B)}$ contains n random values picked from the field. It appends the two keys, $\vec{\boldsymbol{r}}'^{(B)}$, $\mathsf{ID}^{(B)}$ and $\perp$ to the view and outputs the view.

In the following, we will explain why the two views are indistinguishable. In brief, we show that all messages, i.e. queries, it receives is indistinguishable in the real and ideal models. In both views, $S^{(A_\sigma)}$, $\mathsf{ID}^{(B)}$ and $\perp$ are identical. Also r_{A_σ} and r'_{A_σ} are chosen uniformly at random so they are indistinguishable. Moreover, $lk^{(B)}, pk^{(B)}, lk'^{(B)}$ and $pk'^{(B)}$ are the keys picked uniformly at random, so they are indistinguishable. In the real model, each vector $\vec{\boldsymbol{r}}_i^{(B)}$ contains n values blinded with fresh pseudorandom values. In the ideal model, each vector $\vec{\boldsymbol{r}}_i'^{(B)}$ has n fresh random elements of the field. Since the random values and blinded values are indistinguishable, $\vec{\boldsymbol{r}}^{(B)}$ and $\vec{\boldsymbol{r}}'^{(B)}$ are indistinguishable. Note, no client A_σ gets any set (representation) of other clients, therefore even if all $A_\sigma \in \{A_1, ..., A_\xi\}$ collude with each other, they cannot learn anything about client B's set elements. We conclude, the two views are indistinguishable. $\square$

H Feather's Extensions

In this section, we explain how Feather can be extended to (1) further lower the storage cost at the client-side, (2) allow a client to reset its local counter, (3) support the delegation of the authorisation to a semi-honest third party, and (4) further reduce the communication cost.

H.1 Authorizer Storage Space Reduction

There are cases where the clients which authorise the computation have only access to a device with *very limited storage capacity*, e.g. mobile phone or tablet. In the following, we show how we can slightly adjust the PSI protocol, to allow those kinds of authoriser to grant the computation too. The main idea is that client B sends only one bin at a time to each client A_σ when they want to delegate the computation. Then, client A_σ can work on only one bin, and generate the corresponding messages to be sent to client B and the cloud. In step 1(a.)i, client B for each vector $\vec{\boldsymbol{z}}_j^{(B)} \in \vec{\boldsymbol{z}}^{(B)}$, finds index e that is the index of vector $\vec{\boldsymbol{z}}_j^{(B)}$ after $\vec{\boldsymbol{z}}^{(B)}$ is pseudorandomly permuted. Next, in step 1(a.)ii, for each $\vec{\boldsymbol{z}}_j^{(B)}$ it computes $\vec{\boldsymbol{r}}_e^{(B)}$ whose elements are computed as follows:

$$\forall i, 1 \leq i \leq n : r_{e,i}^{(B)} = z_{j,i}^{(B)} + \mathtt{PRF}(tk_e^{(B)}, i)$$

Then, it constructs a vector of h tuples $(\vec{\boldsymbol{r}}_e^{(B)}, j, e)$ (where $1 \leq e, j \leq h$), and randomly permutes the vector. Client B, in step 1(a.)iii sends $lk^{(B)}, pk^{(B)}$ and $\mathsf{ID}^{(B)}$ to client A_σ. Then, in the same step, it sends each tuple: $(\vec{\boldsymbol{r}}_e^{(B)}, j, e)$ in the permuted vector to client A_σ. Therefore, in this process, instead of sending the entire vector, $\vec{\boldsymbol{r}}^{(B)} = [\vec{\boldsymbol{r}}_1^{(B)}, ..., \vec{\boldsymbol{r}}_h^{(B)}]$, it sends a tuple (including one of the vectors in $\vec{\boldsymbol{r}}^{(B)}$) at a time. In step 1(b.)ii, client A_σ finds index g the position where a value at index j (after permutation) would move to, when client A_σ's permutation key is used. Then, it regenerates vector $\vec{\boldsymbol{z}}_j^{(A_\sigma)}$. Next, in step 1(b.)iv, it computes values $v_{g,i}^{(A_\sigma)}$ and $v_{e,i}^{(B)}$ as follows. $\forall i, 1 \leq i \leq n :$

$$v_{g,i}^{(A_\sigma)} = \omega_g^{(A_\sigma)}(x_i) \cdot z_{j,i}^{(A_\sigma)}$$

$$v_{e,i}^{(B)} = \omega_e^{(B)}(x_i) \cdot r_{e,i}^{(B)} = \omega_e^{(B)}(x_i) \cdot (z_{j,i}^{(B)} + \mathtt{PRF}(tk_e^{(B)}, i))$$

Also, client A_σ in the same step, computes vector $\vec{\boldsymbol{q}}_e^{(A_\sigma)}$ whose elements are computed as: $\forall i, 1 \leq i \leq n : q_{e,i} = -(v_{g,i}^{(A_\sigma)} + v_{e,i}^{(B)}) + a_{g,i}^{(A_\sigma)}$. In step 1(b.)v, it sends $\vec{\boldsymbol{q}}_e^{(A_\sigma)}$ to client B. In the same step, client A_σ computes $(l_j^{(A_\sigma)}, l_j^{(B)})$ and sends this tuple (and their ids) to the cloud. Note, client B sends the tuples in a random order to client A_σ, therefore the tuples that the cloud receives, i.e. $(l_j^{(A_\sigma)}, l_j^{(B)})$, are also in a random order. So, the above modification reduces the required storage size at the client-side from $O(hd)$ to a bin's size d, with a complexity: $O(d)$.

H.2 Counter Reset

To rest its counter, i.e., set all counters $c_j^{(I)}$ to zero, the client $I \in \{A_1, ...A_\xi, B\}$ regenerates $\vec{\boldsymbol{z}}^{(I)} = [\vec{\boldsymbol{z}}_1^{(I)}, ..., \vec{\boldsymbol{z}}_{h,}^{(I)}]$ where each vector $\vec{\boldsymbol{z}}_j^{(I)}$ contains the most recent blinding factors used to blind the client's y-coordinates. Also, it picks a fresh master key $k'^{(I)}$ and generates $\vec{\boldsymbol{z}}'^{(I)} = [\vec{\boldsymbol{z}}_1'^{(I)}, ..., \vec{\boldsymbol{z}}_h'^{(I)}]$ such that each vector $\vec{\boldsymbol{z}'}_j^{(I)}$ contains n fresh pseudorandom values $z_{j,i}'^{(I)}$ generated the same way as the ones in step 3, with the difference that here $k'^{(I)}$ is used. After that, it computes vector $\vec{\boldsymbol{z}}''^{(I)} = [\vec{\boldsymbol{z}}_1''^{(I)}, ..., \vec{\boldsymbol{z}}_h''^{(I)}]$ where the elements of each vector $\vec{\boldsymbol{z}}_j''^{(I)}$ are computed as follows. $\forall j, 1 \leq j \leq h$, $\forall i, 1 \leq i : z_{j,i}''^{(I)} = z_{j,i}'^{(I)} - z_{j,i}^{(I)}$. It sends $\vec{\boldsymbol{b}}^{(I)} = \pi(pk^{(I)}, \vec{\boldsymbol{z}}''^{(I)})$

to the cloud and asks it to sum (component-wise) each elements in the vector with the elements in the outsourced dataset. The cloud performs as follows: $\forall g, 1 \leq g \leq h, \forall i, 1 \leq i \leq n : o^{(I)}_{g,i} + b^{(I)}_{g,i} = u^{(I)}_{g,i} + z'^{(I)}_{g,i}$. The client keeps $k'^{(I)}$, discards $k^{(I)}$, and sets its entire counter to zero. Note, although the number of counters is at most h and equals the hash table length, each counter's bit-size is independent of and smaller than the bit-size of each element in the table.

H.3 Authorizer Delegation

In Feather's PSI computation, each authorizer client A_σ can further delegate granting the computation to a semi-honest third party without leaking any information about the set elements. As in step 1.b., when the authorizers grant the computation, they do not need to access their outsourced data. Therefore, as long as the third parties do not collude with the cloud, they cannot learn anything about the outsourced set elements. But, the traditional (multi-party) PSI protocols cannot directly and efficiently offer this feature, as the data has to be encoded by each client locally every time PSI is run.

H.4 Further Communication Cost Reduction

The communication (and computation) cost of Feather, in PSI computation, can be further reduced in the case where a subset of authorizer clients A_σ run *multiple instances* of PSI computation, e.g. β times, with client B at different points in time. In this case, each A_σ can compute and send to the cloud the mapping vector: $\overrightarrow{\hat{\boldsymbol{M}}}_{A_\sigma \rightarrow B}$, only once, and ask the cloud to store it. Then, for any subsequent PSI invocation, the cloud *reuses* that mapping vector. For instance, when there are ξ authorizer clients, this technique results in $2h(\beta - 1)\xi$ total reduction in the communication cost.

I Full Asymptotic Cost Analysis

In this section, we analyse and compare the communication and computation complexity of Feather with other delegated and traditional PSI protocols that support multi-client in the semi-honest model.

I.1 Communication Complexity

In PSI Computation. Below, we analyse the protocols' communication complexity during the PSI computation. To compute PSI in Feather, client B, in step 1(a.)iii, sends $tk^{(B)}$ to the cloud, also it sends to each client A_σ, two values: $pk^{(B)}$ and $lk^{(B)}$, and vector $\overrightarrow{\boldsymbol{r}}^{(B)}$ of h bins. So, in total, client B's communication cost is $\xi(2hd + h + 2) + 1$ or $O(c\xi)$. In step 1(b.)v, each client A_σ sends to the cloud $tk^{(A)}$, vector $\overrightarrow{\hat{\boldsymbol{M}}}_{A_\sigma \rightarrow B}$ containing $2h$ elements, hn blinding factors: $a^{(A_\sigma)}_{g,i}$, and in total $2hn$ y-coordinates, i.e. $\omega^{(A_\sigma)}_{g}(x_i)$ and $\omega^{(B_\sigma)}_{g}(x_i)$. In the same step, it sends

$\vec{\boldsymbol{q}}^{(A_\sigma)}$ containing h bins to client B. So, client A_σ total communication cost is $2h(4d+3)$ or $O(c)$. The cloud communication cost is $h(3d+1)$ or $O(c)$; as in step 2, it sends to client B, vector $\vec{\boldsymbol{t}}$ of h bins and h Bloom filters: $\mathtt{BB}_j^{(I)}$, where $|\mathtt{BB}_j^{(I)}| \approx d$ and $d = 100$. Thus, Feather's total PSI communication is $O(c\xi)$.

The total communication cost of each protocol in [1,5,6,49] is $O(c\xi)$, where the majority of the messages in [5,6,49] are the output of a public key encryption scheme, whereas those in [1] and Feather, are random elements of a finite field, that have much shorter bit-length. Moreover, the traditional PSI protocol in [30] has $O(c\xi)$ communication complexity where all messages are the output of a public key encryption scheme. Also, each scheme in [31,40] has $O(c\xi^2)$ communication cost where most of the messages in these protocols have shorter bit-length, i.e. 128-bit, than in [30]. Nevertheless, the total number of messages exchanged in [40] is less than the one in [31].

In Update. Now, we analyse the protocols' communication complexity during the update. In Feather, for a client to update an element, it sends to the cloud two labels, one in each of steps 1 and 6. Moreover, in step 6, it sends to the cloud a vector of $2d+1$ elements and a Bloom filter: $\mathtt{BB}_j^{(I)}$, where $|\mathtt{BB}_j^{(I)}| \approx d$. So, in total its communication cost is $3(d+1)$ or $O(d)$. The cloud in step 1 sends a vector of $2d+1$ elements and a Bloom filter to the client. Therefore, the cloud's communication cost is $3d+1$ or $O(d)$. Also, the update in each protocol in [1,5,6,49] has $O(c)$ communication complexity.

I.2 Computation Complexity

Note that in our computation analysis, we do not count the pseudorandom and hash functions invocation cost, as they are fast operations and their costs are dominated by the other operations, e.g. modular arithmetic, shuffling, and factorization.

In PSI Computation. Next, we analyse the protocols' computation complexity during the PSI computation. To compute PSI in Feather, client B, in step 1(a.)i, shuffles a vector of h elements that costs $O(c)$. It performs nh and $h(n+n\xi+1)$ modular additions in steps 1(a.)ii and 3 respectively. In step 3, it interpolates h polynomials where each costs $O(d)$. As stated in section 3.2, for a fixed bin's capacity: d, and fixed overflow probability, the hash table length: h, is linear with the set cardinality upper bound: c, i.e. $h = (1+\sigma)\frac{c}{d}$, where $\sigma \geq 1$ and for different values of c we have $d = 100$. In step 3, client B factorizes h polynomials where each costs $O(d^2)$. In total, client B's computation cost is $O(c + 201h\xi + 10302h)$ which is $O(c\xi + c)$. Note, client B performs only *inexpensive* modular additions linear with $c\xi$ while other operations cost is linear with c. Each client A_σ in each step 1(b.)i and 1(b.)ii shuffles a vector of h elements that costs $O(2h)$ in total. In step 1(b.)iv, it performs $2hnd$ multiplications and $2hnd$ additions to evaluate the polynomials. In this step, to generate values $v_{g,i}^{(A_\sigma)}, v_{g,i}^{(B_\sigma)}$, it performs $2hn$ multiplications, it permutes two vectors of h

elements and carries out $2hn$ additions to generate vectors $\vec{\boldsymbol{q}}_e^{(A_\sigma)}$. In total, it performs $O(81208h)$ or $O(c)$ modular operations. The cloud in step 2, performs $3hn(\xi + 1)$ additions and $hn(\xi + 1)$ multiplications to generate values $t_{e,i}$. In total, the cloud's computation cost involves $804h(\xi + 1)$ or $O(c\xi + c)$ modular operations. Hence, Feather's computation complexity of PSI is $O(c\xi + c)$.

In the delegated PSI in [5,6,49], set elements are represented as a single polynomial whose degree is linear to the set cardinality, c. In these protocols, the cost of computing PSI, for each client, is dominated by public key encryption operations, with complexity $O(c)$, and polynomial factorization, for client B, with complexity $O(c^2)$. Therefore, their total computation cost is $O(c\xi+c^2)$. The delegated PSI protocol in [1] uses a hash table and encodes set elements into a set of polynomials of degree d, and this leads to $O(c\xi+c)$ for computing PSI. Now we turn our attention to the traditional PSI protocols [30,31,40]. The computation cost of the protocol in [30] is dominated by threshold additive homomorphic encryption operations. In this protocol, there is a leader party who interacts with other parties to compute a set of values, then all parties interact with each other again to compute the final result. In the protocol, the leader's computation complexity involves $O(c^2)$ exponentiations, while each of the other parties' cost involves $O(c)$ exponentiations. Therefore, the protocol's total computation cost involves $O(c\xi + c^2)$ exponentiations. The authors also provide a variant of the protocol that uses a hash table to reduce the computation cost to $O(c\xi+c\xi \log c)$. The protocols in [31,40] also have a leader client with the same role as above, with a difference that the number of public key operations in these protocols is much lower. The total computation cost of both protocols in [31,40] is $O(c\xi^2+c\xi)$, but [40] has a better performance than [31].

In Update. Now, we analyse the protocols' computation complexity during the update. To update an element in Feather, a client performs $2d + 2$ additions in step 2, and interpolates a polynomial in step 4 that costs $O(d)$. In this step, it extracts a bin set elements that costs $O(d^2)$, and performs nd multiplications and nd additions to evaluate the polynomial. In step 2, the client performs $n+1$ additions to blind the values sent to the cloud. So, the client total computation cost is $O(2n + 2nd + 2 + d + d^2)$ or $O(d^2)$, recall $d = 100$. To update an element in [5,49], a client needs to encode the element as a polynomial, evaluate the polynomial on $2c + 1$ elements and perform $2c + 1$ modular multiplications, to blind the evaluated values. Also, the cloud needs to perform the same number of multiplications to apply the update. Therefore, each protocol's update complexity is $O(c)$. However, in [6] due to the way each outsourced value is blinded, the client has to download the entire dataset, remove the blinding factors and apply the change locally that also costs $O(c)$. In the protocol proposed in [1], if a client naively updates only one bin, then the cloud would be able to learn which elements have been updated with a non-negligible probability. The reason is that the bins are in their original order, and each bin's address is the hash value of an element in the bin. If the client retrieves a bin, then the cloud would be able to figure out what element is being updated with a non-negligible probability when the set universe is not big. So, for the client to securely update its data, it has to locally re-encode the entire outsourced data that costs $O(c)$ too.

J Concrete Cost Evaluation

In this section, we first show how in Feather optimal parameters of a hash table can be determined. Then, we provide a concrete communication and computation evaluation of three protocols: Feather and [1,40]. The reason we only consider [1,40] is that [40] is the fastest traditional multi-client PSI while [1] is the fastest delegated PSI among the protocols studied in section 5. We consider the semi-honest model where all but one clients can collude with each other. The protocol in [1] was mainly designed and implemented for the two-client setting; however, in theory, it is capable of supporting multi-client. Thus, for the sake of fair comparison, we include it in our analysis.

J.1 Choice of Parameters

In Feather, with the right choice of parameters, the cloud can keep the overall costs optimal, while keeping bins' overflow probability negligibly small, e.g. 2^{-40}. In this section, we show how the hash table parameters can be chosen. As before, let c be the upper bound of the set cardinality, d be the bin size, and h be the number of bins. Recall that in our protocol, as we have shown, the overall (computation, communication, and storage) cost depends on the product, hd (i.e. the total number of elements, including set elements and random values stored in the hash table). Furthermore, the computation cost is dominated by factorizing h polynomials of degree $n = 2d + 1$. In order for the cloud to keep the costs optimal, given c, it uses inequality 2, in Section 3.2, to find the right balance between parameters d and h, in the sense that the cost of factorizing a polynomial of degree $n = 2d + 1$ is minimal, while hd is *close* to c.

At a high level, we do the following to find the right parameters. First, we measure the average time, t, taken to factorize a degree $n = 2d + 1$ polynomial for different values of n (see Figure 2). Then, for each c, we compute h for different values of d (see Figure 3). Next, for each d we compute ht, after that for each c we find minimal d whose ht is at the lowest level (see Figure 4). In the experiments, the number of set elements upper bound ranges over $[2^{10}, 2^{20}]$, and the bit length of set elements is 64. First, we determine the running time of polynomial factorization, t. To calculate the running time, we factorize random polynomials of degree $n = 2d + 1$, for different values of n, where $n \in [20, 1000]$, so n can have small and large values. The experiment's [4] result is depicted in Figure 2.

[4] The experiment is based on an implementation written in C++. It were conducted on a macOS laptop, with an Intel $i5$@2.3 GHz CPU, and 16 GB RAM. We use the NTL library for factorizing the polynomials defined over a field, $\mathbb{F}_p$, where p is an 64-bit prime number. Also, to obtain the average time, for each polynomial's degree, n, we run the experiment 100 times using random polynomials whose coefficients are picked uniformly at random from the field.

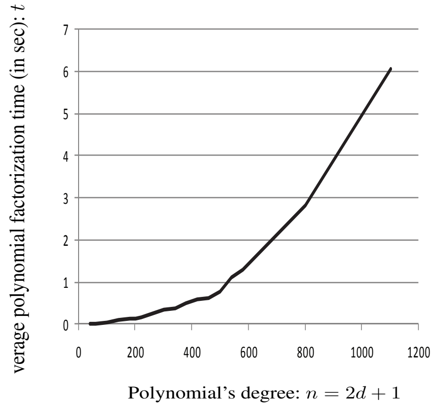


Fig. 2: The average time taken to factorize polynomials of degree n defined over $\mathbb{F}_p$, where p is a 64-bit prime number.

As we can see in the figure when $n > 600$ the time grows rapidly. This means, if we put so many elements in a bin, it would take too long to factorize a bin's polynomial. So we set $n < 600$ (i.e. $d < 300$). Second, for each value of $c \in \{2^{10}, 2^{13}, 2^{15}, 2^{17}, 2^{20}\}$, and $d \in [20, 290]$, we use the inequality to find their corresponding number of bins, h; while keeping the probability (of bin overloading) below 2^{-40}. The result is depicted in Fig 3. Interestingly, as it is evident in the figure, (for all $c \in \{2^{10}, 2^{13}, 2^{15}, 2^{17}, 2^{20}\}$) h grows rapidly when $d < 50$ decreases; however, such growth is much slower when $d > 100$ reduces. Third, given t and h, for each c we calculate the time of factorizing h polynomial of degree $n = 2d+1$ for different d (i.e. we calculate ht). The result is illustrated in Fig 4. It is evident in the figure that, for all c, when $100 \leq d \leq 110$ the computation cost is at the lowest level, so we set $d = 100$. In this case, as can be seen in figure 3, $hd \leq 4c$. For example, as we show in the graph at the bottom left of the figure (i.e. $c = 2^{20} = 1048576$), for $d = 100$, we have $h = 41943$, so $hd = 4194300 \approx 4c$.

In conclusion, in the protocol, the cloud can set $d = 100$ for all values of c. In this setting, hd is at most $4c$ and only with a negligibly small probability, 2^{-40}, a bin may receive more than d elements.

J.2 Concrete Communication Cost Analysis

In PSI Computation. Below, we compare the three schemes' concrete communication costs during the PSI computation. Figure 5 summarises the result. In short, Feather has from 8 to 496 times lower communication cost than [40], while it has from 1.6 to 2.2 times higher communication cost than [1]. The communication cost of [40] is much higher than the other two protocols and grows much

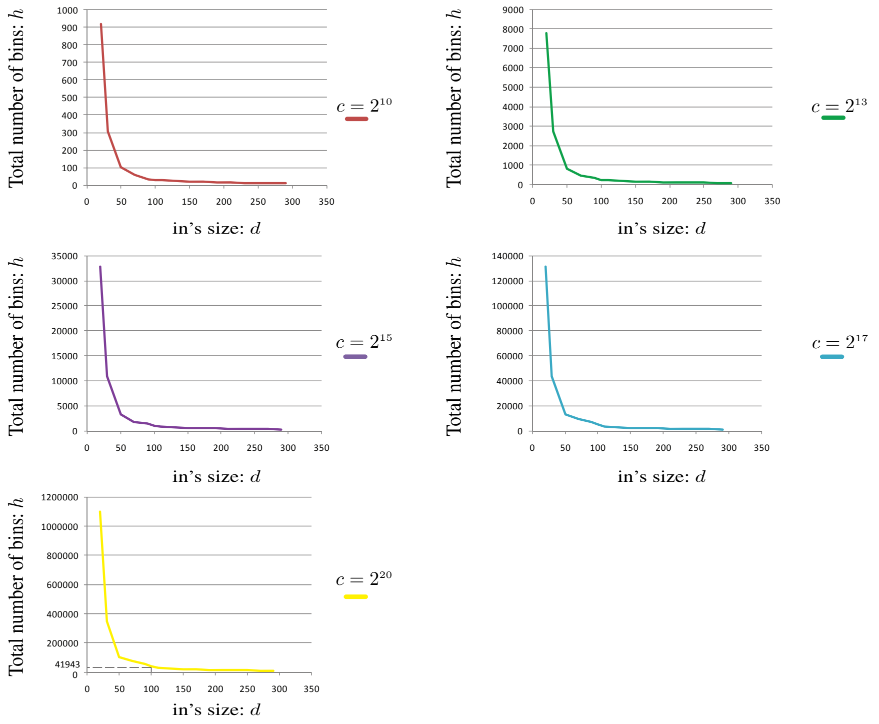


Fig. 3: The relation between the number of bins, h, and the size of each bin, d, for different set size upper bounds, c.

faster when the number of clients increases[5]. The reason is that the number of rounds and messages exchanged between parties in [40] is much higher.

In particular, the total number of bits: b, exchanged in [40] is *quadratic* with the number of clients, i.e., $b = (\xi + 1)(\xi - 1)(l(m_1\beta_1 + m_2\beta_2) + 500(m_1 + m_2))$. For instance, in [40] when an element's size is 40-bit and set cardinality is 2^{16}, the total communication cost in MB ranges in $[407, 505658]$ when the number of clients in the range $[3, 100]$. Nevertheless, in Feather and [1], the total number of bits: b, is *linear* with the number of clients, i.e., in Feather, $b = u((\xi - 1)(10hd + 7h + 2) + h(2d + 1) + 1) + 5771h$ and in [1], $b = u'\xi(h(2d + 1)) + 256(\xi - 1)$. For example, for the same parameters as above, the total communication cost (in MB) of Feather is in the range $[30, 1311]$, while that cost is in the range $[18, 625]$ for [1]. In the above equations, $\xi + 1$ refers to the total number of clients, l is the bit-length of oblivious programmable pseudorandom function's (OPRF) output, such that $l \geq 128$, and m_i, β_i are parameters of Cuckoo hashing, defined in Table 2 in [40], and 500 refers to the amortized communication cost of each OPRF instance in bits. Moreover, u and u' are bit-length of a set's element in

[5] The x-axes, in the figures provided in this paper, are on a logarithmic scale.

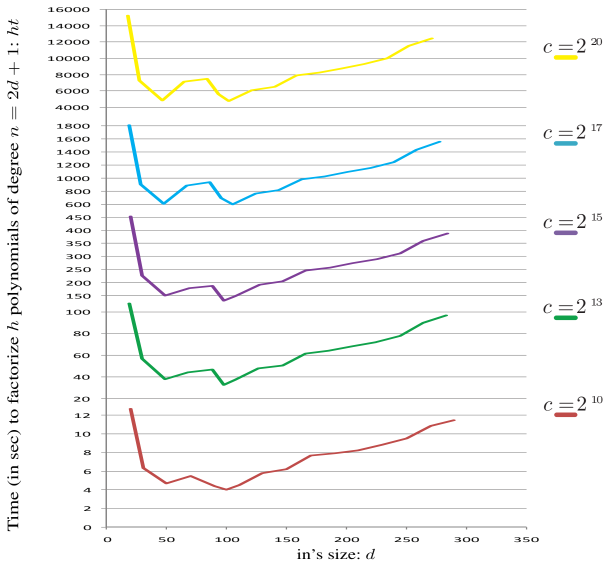


Fig. 4: The average time taken to factorize h polynomials of degree $n = 2d + 1$, for different set size upper bounds, c. The polynomials are defined over the field $\mathbb{F}_p$, where p is an 64-bit prime number.

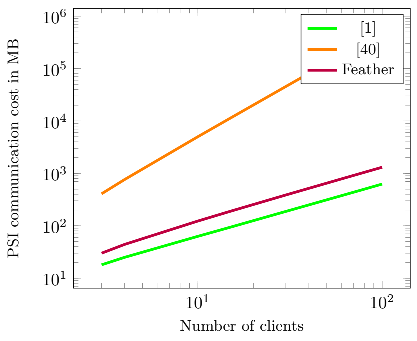


Fig. 5: Communication cost comparison between Feather and [1,40] during PSI computation.

Feather and [1] respectively, such that $u' \geq 2u + 15$. Also, h and $d = 100$ refer to a hash table length and bin's capacity respectively.

The main reason that our protocol has a slightly higher communication cost than [1] is that in our protocol we allow each client A_σ to send to the cloud $2hn$ y-coordinates of random polynomials; instead, similar to [1], we could have let each client send to the cloud only a single key for a pseudorandom function with which all pseudorandom polynomials and their y-coordinates would be re-computed. However, we observed that this would add significant computation cost to the cloud, especially when the number of clients is high, as the cloud would have to evaluate the random polynomials $2\xi hn$ times. Therefore, we tolerate about 2x added communication cost to achieve much higher computation improvement at the cloud side. We provide a detailed concrete communication comparison between the three protocols in Table 2, and detailed communication cost, breakdown by party, in Feather in Table 5.

Table 5: Feather's concrete communication cost (in MB): breakdown by party.

Party	**Setting**	**Field-size**	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
Result Recipient	2-Client	40-bit:	0.05	0.1	0.21	0.42	0.84	1.69	4.52	9.04	18.1	36.2	72.4
		64-bit:	0.06	0.14	0.28	0.57	1.14	2.28	6.1	12.21	24.42	48.84	97.7
		128-bit:	0.11	0.24	0.48	0.96	1.93	3.87	10.31	20.63	41.28	82.57	165.14
	3-Client	40-bit:	0.08	0.16	0.33	0.66	1.34	2.68	7.15	14.31	28.63	57.27	114.56
		64-bit:	0.11	0.24	0.48	0.96	1.93	3.87	10.31	20.63	41.28	82.57	165.14
		128-bit:	0.21	0.43	0.87	1.75	3.51	7.03	18.74	37.49	75	150.01	300.03
	5-Client	40-bit:	0.14	0.28	0.57	1.16	2.32	4.66	12.42	24.85	49.71	99.43	198.86
		64-bit:	0.21	0.43	0.87	1.75	3.51	7.03	18.74	37.49	75	150	300
		128-bit:	0.4	0.82	1.65	3.32	6.67	13.35	35.6	71.21	142.44	284.89	569.81
	10-Client	40-bit:	0.29	0.59	1.19	2.39	4.79	9.6	25.59	51.19	102.4	204.81	409.63
		64-bit:	0.45	0.92	1.85	3.72	7.46	14.93	39.82	79.64	159.3	318.62	637.25
		128-bit:	0.89	1.8	3.61	7.26	14.56	29.16	77.75	155.5	311.04	622.11	1244.26
	100-Client	40-bit:	3	6.11	12.22	24.55	49.2	98.51	262.66	525.33	1050.77	2101.64	4203.38
		64-bit:	4.79	9.75	19.51	39.18	78.51	157.19	419.13	838.26	1676.69	3353.54	6707.25
		128-bit:	9.57	19.46	38.93	78.18	156.68	313.68	836.37	1672.75	3345.82	6691.96	13384.2
Authoriser		40-bit	0.12	0.24	0.49	0.98	1.97	3.96	10.56	21.12	42.25	84.51	169.0
		64-bit:	0.19	0.39	0.78	1.57	3.16	6.33	16.9	33.8	67.6	135.22	270.44
		128-bit:	0.38	0.78	1.57	3.15	6.33	12.67	33.8	67.6	135.21	270.44	540.89
Cloud		40-bit	0.03	0.06	0.12	0.24	0.49	0.98	2.63	5.26	10.53	21.07	42.15
		64-bit:	0.04	0.09	0.19	0.39	0.78	1.58	4.21	8.42	16.85	33.72	67.44
		128-bit:	0.09	0.19	0.39	0.78	1.57	3.16	8.42	16.85	33.71	67.44	134.88

In Update. Now, we evaluate the protocols' concrete communication cost during the update. In [1], in order for a client to securely update its outsourced data, it has to download the entire set, locally update and upload the data. Via this approach, the update's total communication cost (for both the client and cloud), in MB, will be in the range $[0.13, 210]$ when the set size is in the range $[2^{10}, 2^{20}]$ and each element bit-size is 40. But, in Feather a client needs to download and upload only one bin, that makes its total communication cost of update 0.003 MB for all set sizes (for the same element bit size as above). Thus, Feather's communication cost is from 45 to 70254 times lower than the one in [1].

J.3 Concrete Computation Cost Analysis

In this section, we provide an empirical computation evaluation of Feather using a prototype implementation developed in C++. Feather's source code includes about 2000 lines of code (counted by cloc[6]), is available on a GitHub public repository[7]. We compare the concrete computation cost of Feather against the two protocols in [1,40]. Our prototype implementations utilise GMP[8], Cryptopp[9], NTL[10], and Bloom Filter[11] libraries for big-integer arithmetics, cryptographic primitives, polynomial factorization and element membership checks respectively. All experiments were conducted on a macOS laptop, with an Intel $i5$@2.3 GHz CPU, and 16 GB RAM. We do not take advantage of parallelization even though our protocol is highly parallelizable. In the experiments, we use randomly generated 40 or 64 bits integers as set elements, and set size in the range $[2^{10}, 2^{20}]$. The probability of bin overflow in the hash table, for the three protocols, and false-positive in the Bloom filters, for Feather, is 2^{-40}. Similarly, the probability of bin overflow in [1] is set to 2^{-40}. The error probability in the padding scheme, for [1], is also set to 2^{-40} that results in padding size in the range $[52, 63]$ for the above set size range. To achieve optimal performance, as discussed in Section J.1, we set bin capacity to $d = 100$. These choices of hash table parameters lead to a hash table length in the range $[30, 41943]$. Accordingly, in Feather, a Bloom filter's parameters is set as follows, the total number of elements: 100, the false-positive probability: 2^{-40}, the optimal number of hash functions: 40, and the optimal bit-length of Bloom filter: 5771. We also use two different field sizes for Feather: 40 and 64 bits.

In PSI Computation. Now, we compare the three schemes' runtime during the PSI computation. First, we compare the runtime of Feather and the PSI in [1]; to do that we run the latter protocol on the same machine specified above. As the protocol in [1] has been designed and implemented for the two-client setting, we compare the performance of Feather and the scheme in [1] in that setting. Figure 6 summarises the result.

As Figure 6 indicates, Feather's performance is better than the PSI in [1]. In particular, Feather is 2-2.5 times faster than the PSI in [1]. The main reason is that Feather avoids using the padding technique, so it can operate on a smaller field size; wheres, the protocol in [1] uses the padding that negatively affects all operations, particularly polynomial factorization, the dominant operation in both protocols.

Tables 6 and 7 compare the two protocols' performance at setup and PSI computation respectively. As Table 6 illustrates, client-side setup runtime in Feather is up to 1.4 times lower than that in [1]. This Feather's improvement

[6] https://github.com/AlDanial/cloc
[7] https://github.com/anonymous-02020/Feather/tree/master/Implementation
[8] https://gmplib.org
[9] https://www.cryptopp.com
[10] https://www.shoup.net/ntl
[11] http://www.partow.net/programming/bloomfilter/index.html

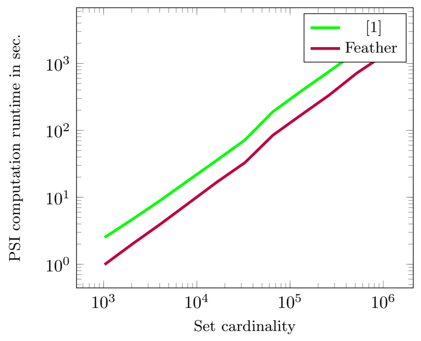


Fig. 6: PSI computation runtime comparison between Feather and [1].

Table 6: Client-side setup runtime comparison between Feather and [1] (in sec.).

Phase	Protocols	Elem. size	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
Client-side Setup	Feather	40-bit	0.11	0.22	0.45	0.9	1.8	3.53	10.17	20.42	40.06	81.35	163.01
		64-bit	0.11	0.23	0.45	0.9	1.81	3.54	10.24	20.44	40.74	81.6	163.13
	[1]	40-bit	0.16	0.33	0.65	1.31	2.6	5.18	13.87	27.49	55.56	113.48	233.79
		64-bit	0.16	0.33	0.65	1.32	2.61	5.24	14.27	27.65	55.95	119.1	234.78

Table 7: PSI computation runtime comparison between Feather and [1] (in sec.).

Phases	Protocols	Elem. size	Set's Cardinality										
			2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
Client-side Computation Delegation	Feather	40-bit	0.08	0.21	0.38	0.8	1.63	3.41	9	16.8	34.1	73.2	145.57
		64-bit	0.09	0.21	0.42	0.83	1.66	3.42	9.01	18.06	35.18	73.62	147.09
	[1]	40-bit	0.35	0.71	1.44	2.87	5.78	11.46	30.82	60.26	121.83	244.44	488.14
		64-bit	0.36	0.72	1.45	2.88	5.98	11.49	31.47	60.88	122.22	256.01	490.04
Cloud-side Computation	Feather	40-bit	0.01	0.02	0.04	0.1	0.2	0.43	1.08	2.15	4.42	9.04	17.78
		64-bit	0.01	0.02	0.05	0.1	0.21	0.43	1.1	2.17	4.8	9.1	17.9
	[1]	40-bit	0.34	0.68	1.36	2.74	5.49	10.95	29.24	58.67	116.43	233.36	469.79
		64-bit	0.34	0.69	1.36	2.75	5.59	10.99	30.12	58.7	116.64	244.27	470.5
Client-side Result Retrieval	Feather	40-bit	0.9	1.79	3.59	7.31	15.02	29.1	74.68	150.58	297.07	640.63	1208.99
		64-bit	1.23	2.54	5	9.5	20.23	39.63	102.5	200.26	411.27	806.49	1636.61
	[1]	40-bit	1.82	3.33	6.28	12.47	24.55	49.43	130.5	265.67	519.31	1041.26	2081.19
		64-bit	2.39	4.01	7.66	14.92	29.63	58.72	162.53	313.42	620.69	1293.42	2479.55
Total	Feather	40-bit	0.99	2.02	4.01	8.21	16.85	32.94	84.76	169.53	335.59	722.87	1372.34
		64-bit	1.33	2.77	5.47	10.43	22.1	43.48	112.61	220.49	451.25	889.21	1801.6
	[1]	40-bit	2.51	4.72	9.08	18.08	35.82	71.84	190.56	384.6	757.57	1519.06	3039.12
		64-bit	3.09	5.42	10.47	20.55	41.2	81.2	224.12	433	859.55	1793.7	3440.09

stems from two main factors, (a) using an efficient error detecting mechanism that allows parties to work on a smaller field, and (b) utilising a more efficient polynomial evaluation method, i.e., Horner's method. Moreover, as Table 7 shows, the cloud-side performance, in Feather is 26-34 times faster than [1] and the gap will increase if more clients participate. The reason is that Feather allows each client to send y-coordinates of random polynomials to the cloud. Therefore, unlike [1], the cloud does not need to re-evaluate the polynomials and this yields saving computation cost at the cloud-side (with the penalty of doubling the overall communication cost). Otherwise, this computation cost would be a bottleneck at the cloud, particularly when the number of clients is high.

Now, we compare the runtime of Feather and [40]. For the latter protocol's runtime, we use the values reported in [40]. For Feather, we measure the total runtime of three phases: client-side PSI delegation (that includes both clients), cloud-side result computation and client-side result retrieval. We highlight that in Feather all parties use a single thread and core; whereas, in [40] the leader client deals with all other clients' queries in parallel using multiple cores which leads to lower runtime at the leader-side. Moreover, the experiment in [40] took advantage of 16 times bigger RAM than ours. Similar to the other two protocols, we assume each client upon receiving a message, starts performing computations on it. As Figure 7 shows, the performance of [40] is better than Feather especially for a small number of clients. The main reason is that Feather utilises polynomial factorization to extract the set elements from outsourced data. This operation makes Feather's performance slower than [40] which does not need to use it, as the clients keep their data locally. Hence, this is the added cost of our protocol to achieve data outsourcing where the clients can delete their data after outsourcing it. To provide concrete values here, for instance, when set cardinality is 2^{16} and element bit-size is 40, [40] is from 40 to 4 times faster than Feather when the number of clients is 3 and 15 respectively. However, the performance of the protocol in [40] becomes *significantly* worse when the number of clients increases. The main reason is that the protocol's total computation cost is quadratic with the number of clients.

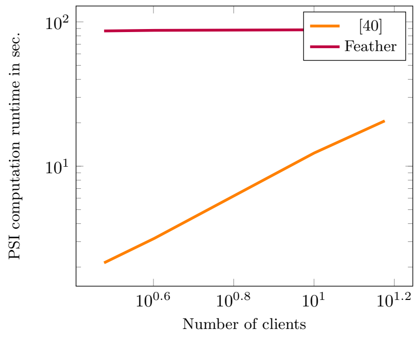


Fig. 7: PSI computation runtime comparison between Feather and [40].

Nonetheless, in Feather, the runtime *gradually* grows when the number of clients increases, as its total computation cost is linear with the number of clients; see Table 8 for a detailed runtime comparison between the two protocols. We

Table 8: PSI computation runtime comparison between Feather and [40] (in sec.).

Protocols	Elem. size	Set's Cardinality			Number of Clients			
		2^{12}	2^{16}	2^{20}	3	4	10	15
[40]	40, 64-bit	✓			0.3	0.34	1.01	1.85
			✓		2.14	3.16	12.33	20.61
				✓	41.64	52.25	182.8	304
Feather	40-bit	✓			4.29	4.32	4.45	4.64
			✓		86.56	87.38	88.14	88.78
				✓	1383.79	1414.39	1639.94	1945.24
	64-bit	✓			5.32	5.43	5.49	6.3
			✓		111.08	113.51	114.31	116.11
				✓	1817.5	1835.51	2143.33	2443.61

highlight that, [40] has not reported the runtime for more than 15 clients; but, it is not hard to see that for a (relatively) large number of clients, e.g., 100-150, our protocol will outperform [40]. We have also conducted experiments for the case where a very large number of clients participate in our protocol, i.e., up to 16000 clients. To provide a concrete value here, our protocol only takes 4.7 seconds to run PSI on 1000 clients each of which has 2^{11} elements. See Table 9 for more detail.

Table 9: Feather's PSI computation runtime for different number of clients (in sec.). Elements size: 40-bit.

Protocol	Set's Cardinality	Number of Clients									
		10	20	100	1000	5000	6000	7000	10000	15000	16000
Feather	2^{10}	0.99	1	1.13	2.28	21.09	37.41	51.45	119.94	286.08	328.85
	2^{11}	2.06	2.04	2.27	4.75	107.65	158.2	214.5	406.41	891.04	909.46

In Update. Now, we compare the runtime of Feather and the PSI in [1] during the update. Note, the protocol and implementation in [1] do not provide any mechanism for updating sets, but we have developed a prototype implementation of this protocol that allows clients to securely update their outsourced data by downloading and unblinding its entire outsourced data; updating a bin; re-encoding the elements of it; re-blinding the entire set; and uploading it to the cloud. The source code is available in [3]. For the update operation in Feather, we measure the total runtime of four phases at the client-side: (a) computing an update query, (b) unblinding and decoding a bin, (c) applying the update, and (d) encoding and re-blinding the bin. For the update in [1], we measure the total time when a client starts unblinding its set until it finishes re-blinding it.

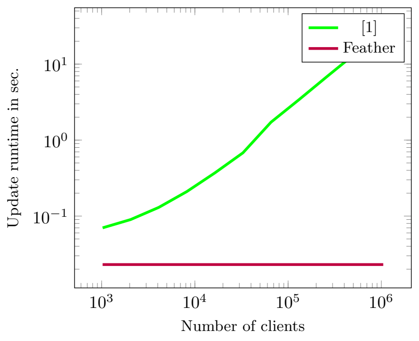


Fig. 8: Update runtime comparison between Feather and [1].

Figure 8 depicts the update runtime comparison between the two protocols. As evident, the update runtime for Feather is much lower than that of in [1], especially for a larger set size. Recall, we have provided a detailed update runtime comparison between the two protocols in Table 3. As the table shows, the update runtime of the PSI in [1], for 40-bit elements, grows from 0.07 to 27 seconds when the set size increases from 2^{10} to 2^{20}; whereas for Feather, the update runtime remains 0.023 seconds for all set sizes. Hence, the update in Feather is from 3 to 1182 times faster than the one in [1], for 40-bit elements. The reason for this huge difference in the performance is that in [1] the client needs to unblind and re-blind all the bins in the hash table, but in Feather such operations are carried out only on one bin.