Multi-party Updatable Delegated Private Set Intersection

Aydin Abadi[1(✉)], Changyu Dong[2], Steven J. Murdoch[1], and Sotirios Terzis[3]

[1] University College London, London, UK
{aydin.abadi,s.murdoch}@ucl.ac.uk
[2] Newcastle University, Newcastle upon Tyne, UK
changyu.dong@newcastle.ac.uk
[3] University of Strathclyde, Glasgow, UK
sotirios.terzis@strath.ac.uk

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 [23]. PSI has various applications. For instance, it has been used in COVID-19 contact tracing schemes [21], remote diagnostics [17], and Apple's child safety solution to combat "Child Sexual Abuse Material" (CSAM) [14]. 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 [22].

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, 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

I. Eyal and J. Garay (Eds.): FC 2022, LNCS 13411, pp. 100–119, 2022.
https://doi.org/10.1007/978-3-031-18283-9_6

the cloud, based on its benefits proven in the market [41]. 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 [42]), e-commerce (e.g., consumer behaviour prediction [43]), or e-health (e.g., cancer research on genomic datasets [11]), 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 [9].

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 s 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 [23], 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 [36] 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 [39] 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 [14] 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 [10,16,19,26]. Recently, a new PSI has been proposed that achieves a better balance between communication and computation costs [18]. Also, researchers designed PSIs that let multiple (i.e., more than two) parties efficiently compute the intersection. The multi-party PSIs in [28,37] are secure against passive adversaries while those in [12,25,45] were designed to remain secure against active ones. To date, the protocols in [37] and [25] are the most efficient multi-party PSIs designed to be secure against passive and active adversaries respectively. The computation complexities of [37] and [25] are $O(c\xi^2 + c\xi)$ and $O(c\xi)$ respectively, where ξ is the number of clients. However, Abadi *et al.* [5] 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 [30,33,46] cannot reuse their outsourced encrypted data and require clients to re-encode their data locally for each computation. The most efficient such protocol is [30], 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 [38,40,47] are not secure, as illustrated in [1,6]. In contrast, the PSIs in [1,6,7,44] are secure. Those in [6,7,44] 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 [6,7,44] 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 [1,6,7] 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.

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 [32].

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 [35] can permute a vector of m elements in time $O(m)$. Formal definitions of pseudorandom function and permutation can be found in [32].

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 [13]. In the paper's full version [4], we explain how the hash table parameters are set.

3.3 Horner's Method

Horner's method [20] 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 [15] 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 the paper's full version [4], we explain how the Bloom filter parameters can be set.

3.5 Representing Sets by Polynomials

Freedman *et al.* in [23] 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 [34], 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 [34] 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 [1,6,34] 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 [1,26]. A polynomial in this form can be converted into coefficient form via polynomial interpolation, e.g., via Lagrange interpolation [8].

Usually, PSIs that rely on this representation assume that all x_i are picked from $\mathbb{F}\backslash\mathcal{U}$. 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 [1,7,29], 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 [27,31], 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 the paper's full version [4] 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.

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 Sect. 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 [24].

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 $I \in \{A_1, ...A_\xi, B\}$ 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: $\prod_{l=1}^{d}(x - e_l^{(I)})$, 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: $y_{j,i}^{(I)} = \prod_{l=1}^{d}(x_i - e_l^{(I)})$, 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: $\vec{\mathtt{BB}}^{(I)} = [\mathtt{BB}_1^{(I)}, ..., \mathtt{BB}_h^{(I)}]$.
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: $\vec{\hat{\boldsymbol{o}}}^{(I)} = \pi(pk^{(I)}, \vec{\boldsymbol{o}}^{(I)})$, $\vec{\hat{\boldsymbol{l}}}^{(I)} = \pi(pk^{(I)}, \vec{\boldsymbol{l}}^{(I)})$, 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 $\vec{\mathtt{BB}}^{(I)}$ as: $\vec{\hat{\mathtt{BB}}}^{(I)} = \pi(pk^{(I)}, \vec{\mathtt{BB}}^{(I)})$.
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: $(\vec{\boldsymbol{o}}^{(I)}, \vec{\boldsymbol{l}}^{(I)})$ along with the permuted blinded Bloom filters: $\overrightarrow{\hat{\mathtt{BB}}}^{(I)}$ 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 $(y_{j,i}^{(I)}, x_i)$ 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: $\prod_{m=1}^{d}(x - s_m'^{(I)})$, 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 $\vec{\boldsymbol{u}}_j^{(I)} = [u_{j,1}^{(I)}, ..., u_{j,n}^{(I)}]$. It discards $\mathtt{B}_j^{(I)}$ and builds a fresh one: $\mathtt{B}_j'^{(I)}$ encoding $s^{(I)}$ and valid roots of $\psi_j(x)$.
* otherwise, i.e., if $s^{(I)} \in \mathtt{B}_j^{(I)}$, it sets $\vec{\boldsymbol{u}}_j^{(I)} = \vec{\boldsymbol{y}}_j^{(I)}$ and $\mathtt{B}_j'^{(I)} = \mathtt{B}_j^{(I)}$, where $\vec{\boldsymbol{y}}_j^{(I)}$ and $\mathtt{B}_j^{(I)}$ 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 $\vec{\boldsymbol{u}}_j^{(I)} = \vec{\boldsymbol{y}}_j^{(I)}$ and $\mathtt{B}_j'^{(I)} = \mathtt{B}_j^{(I)}$, where $\vec{\boldsymbol{y}}_j^{(I)}$ and $\mathtt{B}_j^{(I)}$ were computed in step 2.b. It means the element does not exist in the set, so no deletion is needed.
  * otherwise, if $s^{(I)} \in \mathtt{B}_j^{(I)}$, it uses pairs $(y_{j,i}^{(I)}, x_i)$ to interpolate a polynomial: $\psi_j(x)$. It constructs a polynomial: $\prod_{m=1}^{d}(x - s_m'^{(I)})$, 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 $\vec{\boldsymbol{u}}_j^{(I)} = [u_{j,1}^{(I)}, ..., u_{j,n}^{(I)}]$. Also, it discards $\mathtt{B}_j^{(I)}$ and builds a fresh one: $\mathtt{B}_j'^{(I)}$ that encodes valid roots of $\psi_j(x)$ excluding $s^{(I)}$.

5. ***Blind***: Blinds the updated bin: $\vec{\boldsymbol{u}}_j^{(I)}$ and Bloom filter: $\mathtt{B}_j'^{(I)}$ as follows.
   a. generates fresh blinding factors:

$$b_j^{(I)} = \mathtt{PRF}'(\mathtt{PRF}'(bk^{(I)}, j), c_j^{(I)}), \quad \forall i, 1 \le i \le n : z_{j,i}^{(I)} = \mathtt{PRF}(\mathtt{PRF}(k_j^{(I)}, c_j^{(I)}), i)$$

   b. blinds the bin's content and Bloom filter, using the fresh blinding factors.

$$\mathtt{BB}_j^{(I)} = \mathtt{B}_j'^{(I)} \oplus b_j^{(I)} \quad \textit{and} \quad \forall i, 1 \le i \le n : o_{j,i}^{(I)} = u_{j,i}^{(I)} + z_{j,i}^{(I)}$$

6. ***Send update query***: Sends $\vec{\boldsymbol{o}}_j^{(I)} = [o_{j,1}^{(I)}, ..., o_{j,n}^{(I)}], \mathtt{BB}_j^{(I)}$, $l_j^{(I)}$, 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: $\vec{\boldsymbol{z}}^{(B)} = [\vec{\boldsymbol{z}}_1^{(B)}, ..., \vec{\boldsymbol{z}}_h^{(B)}]$ (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., $\forall g, 1 \leq g \leq h : tk_g^{(B)} = \texttt{PRF}(tk^{(B)}, g)$. Then, using each key, it generates fresh pseudorandom values and uses them to blind the vector's elements, as below:

$$\forall g, 1 \leq g \leq h,\ \forall i, 1 \leq i \leq n : r_{g,i}^{(B)} = z_{a,i}^{(B)} + \texttt{PRF}(tk_g^{(B)}, i)$$

      Let $\vec{\boldsymbol{r}}_g^{(B)} = [r_{g,1}^{(B)}, ..., r_{g,n}^{(B)}]$. Note, $\vec{\boldsymbol{z}}_a^{(B)}$ 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 $lk^{(B)}, pk^{(B)}, \vec{\boldsymbol{r}}^{(B)} = [\vec{\boldsymbol{r}}_1^{(B)}, ..., \vec{\boldsymbol{r}}_h^{(B)}]$, and its id: $ID^{(B)}$, 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 $\overrightarrow{\boldsymbol{M}}_{A_\sigma \rightarrow B}$ whose elements, m_g, are computed as follows.

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

      It permutes the elements of $\overrightarrow{\boldsymbol{M}}_{A_\sigma \rightarrow B}$. This yields mapping vector $\overrightarrow{\hat{\boldsymbol{M}}}_{A_\sigma \rightarrow B}$
      ii. ***Regen. blinding factors:*** regenerates the most recent blinding factors: $\vec{\boldsymbol{z}}^{(A_\sigma)} = [\vec{\boldsymbol{z}}_1^{(A_\sigma)}, ..., \vec{\boldsymbol{z}}_h^{(A_\sigma)}]$ where each $\vec{\boldsymbol{z}}_g^{(A_\sigma)}$ 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: $a_{g,i}^{(A_\sigma)}$ and two random degree-d polynomials: $\omega_g^{(A_\sigma)}, \omega_g^{(B_\sigma)}$ 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 $\omega_g^{(A_\sigma)}$ and $\omega_g^{(B_\sigma)}$, respectively, i.e., $\forall g, 1 \leq g \leq h$ and $\forall i, 1 \leq i \leq n$:

$$v_{g,i}^{(A_\sigma)} = \omega_{g,i}^{(A_\sigma)} \cdot z_{j,i}^{(A_\sigma)} \quad \text{and} \quad v_{g,i}^{(B_\sigma)} = \omega_{g,i}^{(B_\sigma)} \cdot r_{g,i}^{(B_\sigma)} = \omega_{g,i}^{(B_\sigma)} \cdot (z_{a,i}^{(B)} + \mathtt{PRF}(tk_g^{(B)}, i))$$

Then, given permutation keys: $pk^{(A_\sigma)}$ and $pk^{(B_\sigma)}$, for each value $v_{g,i}^{(A_\sigma)}$ it finds its matched value: $v_{e,i}^{(B_\sigma)}$, such that the blinding factors $z_{j,i}^{(A_\sigma)}$ and $z_{j,i}^{(B)}$ of the two values belong to the same bin, $\mathtt{HT}_j$. Specifically, for each $v_{g,i}^{(A_\sigma)} = \omega_{g,i}^{(A_\sigma)} \cdot z_{j,i}^{(A_\sigma)}$ it finds $v_{e,i}^{(B_\sigma)} = \omega_{e,i}^{(B_\sigma)} \cdot (z_{j,i}^{(B)} + \mathtt{PRF}(tk_e^{(B)}, i))$. Next, it combines and blinds the matched values, i.e., $\forall g, 1 \leq g \leq h$ and $\forall i, 1 \leq i \leq n$:

$$q_{e,i}^{(A_\sigma)} = -(v_{g,i}^{(A_\sigma)} + v_{e,i}^{(B_\sigma)}) + a_{g,i}^{(A_\sigma)} = -(\omega_{g,i}^{(A_\sigma)} \cdot z_{j,i}^{(A_\sigma)} + \omega_{e,i}^{(B_\sigma)} \cdot (z_{j,i}^{(B)} + \mathtt{PRF}(tk_e^{(B)}, i))) + a_{g,i}^{(A_\sigma)}$$

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 $q_{e,i}^{(A_\sigma)}$. It sends to the cloud $ID^{(B)}$, $ID^{(A_\sigma)}$, $\vec{\hat{\boldsymbol{M}}}_{A_\sigma \to B}$, the blinding factors: $a_{g,i}^{(A_\sigma)}$, **"Compute"**, and random polynomials' y-coordinates, i.e., all $\omega_{g,i}^{(A_\sigma)}$, $\omega_{g,i}^{(B_\sigma)}$.

2. ***Cloud-side Result Computation***: The cloud uses each mapping vector: $\vec{\hat{\boldsymbol{M}}}_{A_\sigma \to B}$ 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., $\forall e, 1 \leq e \leq h$ and $\forall i, 1 \leq i \leq n$:

$$t_{e,i} = (\sum_{\sigma=1}^{\xi} \omega_{e,i}^{(B_\sigma)}) \cdot (o_{e,i}^{(B)} + \mathtt{PRF}(tk_e^{(B)}, i)) - \sum_{\sigma=1}^{\xi} a_{g_\sigma,i}^{(A_\sigma)} + \sum_{\sigma=1}^{\xi} \omega_{g_\sigma,i}^{(A_\sigma)} \cdot o_{g_\sigma,i}^{(A_\sigma)}$$

where $o_{g_\sigma,i}^{(A_\sigma)} \in \vec{\boldsymbol{o}}_{g_\sigma}^{(A_\sigma)} \in \vec{\hat{\boldsymbol{o}}}^{(A_\sigma)}$. It sends to B its blinded Bloom filters: $\vec{\hat{\mathtt{BB}}}^{(B)}$ 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 $\vec{\hat{\mathtt{B}}}^{(B)}$. 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., $\forall e, 1 \leq e \leq h$ and $\forall i, 1 \leq i \leq n$:

$$f_{e,i} = t_{e,i} + \sum_{\sigma=1}^{\xi} q_{e,i}^{(A_\sigma)} = (\sum_{\sigma=1}^{\xi} \omega_{e,i}^{(B_\sigma)}) \cdot (u_{j,i}^{(B)}) + \sum_{\sigma=1}^{\xi} \omega_{g_\sigma,i}^{(A_\sigma)} \cdot u_{j,i}^{(A_\sigma)}$$

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 $\mathtt{B}_e^{(B)} \in \vec{\hat{\mathtt{B}}}^{(B)}$ 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 the paper's full version, 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 Sect. 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 the paper's full version, we provide several remarks on Feather's protocols and explain why naive solutions cannot offer Feather's features. In the full version, 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 [28,37], into account, as they are designed

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]	[6]	[7]	[44]	[37]	[28]
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 cmp. 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	–	–

for the cases where parties maintain locally their set elements and do not (need to) support data update. We present a full analysis in the paper's full version.

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 [1,6,7,44] is $O(c\xi)$, where the majority of the messages in [6,7,44] 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 [28,37] 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 [1,6,7,44] 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 [6,7,44], 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 [28,37]. Each PSI in [28,37] 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 [6,44], 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 [7], 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 [1,37]. The reason we only consider [1,37] is that [37] is the fastest *traditional multi-party* PSI while [1] is the fastest *delegated* PSI among the PSIs studied in Sect. 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 the full version) 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 the paper's full version.

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 [37], while it has 1.6–2.2 times higher cost than the one in [1], for 40-bit elements. The PSI's cost in [37] 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 the paper's full version, 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].

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 [1,37]. All experiments were run on a macOS laptop, with an Intel *i*5@2.3 GHz CPU, and 16 GB RAM. In the paper's full version, we provide full detail about the system's parameters used in the experiment.

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
[37]	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

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]. 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 the full version, compare these PSIs' runtime in the setup and PSI computation respectively. Briefly, for a small number of clients, the performance of the PSI in [37] 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 [37] 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 [37] when the number of clients is large. We provide a more detailed analysis in the full version. 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 s to run PSI with 1000 clients where each client has 2^{11} elements. Table 9, in the full version, 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 s when the set size increases from 2^{10} to 2^{20}; whereas in Feather, the update runtime remains 0.023 s 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	← 0.023 →										
	64-bit	← 0.035 →										

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 [25] 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 was 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 Trans. Dependable Secure Comput. **16**(4), 608–624 (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., Dong, C., Murdoch, S.J., Terzis, S.: Multi-party updatable delegated private set intersection-full version. In: FC (2022)
5. Abadi, A., Murdoch, S.J., Zacharias, T.: Polynomial representation is tricky: maliciously secure private set intersection revisited. In: Bertino, E., Shulman, H., Waidner, M. (eds.) ESORICS 2021. LNCS, vol. 12973, pp. 721–742. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-88428-4_35
6. Abadi, A., Terzis, S., Dong, C.: O-PSI: delegated private set intersection on outsourced datasets. In: Federrath, H., Gollmann, D. (eds.) SEC 2015. IAICT, vol. 455, pp. 3–17. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-18467-8_1
7. Abadi, A., Terzis, S., Dong, C.: VD-PSI: verifiable delegated private set intersection on outsourced private datasets. In: Grossklags, J., Preneel, B. (eds.) FC 2016. LNCS, vol. 9603, pp. 149–168. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54970-4_9
8. Aho, A.V., Hopcroft, J.E.: The Design and Analysis of Computer Algorithms. Pearson Education India (1974)
9. Apple Inc.: Security threat model review of Apple's child safety features (2021). https://www.apple.com/child-safety/pdf/Security_Threat_Model_Review_of_Apple_Child_Safety_Features.pdf
10. Badrinarayanan, S., Miao, P., Raghuraman, S., Rindal, P.: Multi-party threshold private set intersection with sublinear communication. In: Garay, J.A. (ed.) PKC 2021. LNCS, vol. 12711, pp. 349–379. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-75248-4_13
11. 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)
12. Ben-Efraim, A., Nissenbaum, O., Omri, E., Paskin-Cherniavsky, A.: PSImple: practical multiparty maliciously-secure private set intersection. IACR Cryptology ePrint Archive (2021)
13. Berenbrink, P., Czumaj, A., Steger, A., Vöcking, B.: Balanced allocations: the heavily loaded case. In: STOC (2000)

14. Bhowmick, A., Boneh, D., Myers, S., Talwar, K., Tarbe, K.: The Apple PSI system (2021). https://www.apple.com/child-safety/pdf/Apple_PSI_System_Security_Protocol_and_Analysis.pdf
15. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7), 422–426 (1970)
16. Branco, P., Döttling, N., Pu, S.: Multiparty cardinality testing for threshold private set intersection. IACR Cryptology ePrint Archive (2020)
17. Brickell, J., Porter, D.E., Shmatikov, V., Witchel, E.: Privacy-preserving remote diagnostics. In: CCS (2007)
18. Chase, M., Miao, P.: Private set intersection in the internet setting from lightweight oblivious PRF. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020. LNCS, vol. 12172, pp. 34–63. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-56877-1_2
19. Chen, H., Laine, K., Rindal, P.: Fast private set intersection from homomorphic encryption. In: ACM CCS (2017)
20. Dorn, W.S.: Generalizations of Horner's rule for polynomial evaluation. IBM J. Res. Dev. **6**(2), 239–245 (1962)
21. Duong, T., Phan, D.H., Trieu, N.: Catalic: delegated PSI cardinality with applications to contact tracing. In: Moriai, S., Wang, H. (eds.) ASIACRYPT 2020. LNCS, vol. 12493, pp. 870–899. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64840-4_29
22. 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
23. Freedman, M.J., Nissim, K., Pinkas, B.: Efficient private matching and set intersection. In: Cachin, C., Camenisch, J.L. (eds.) EUROCRYPT 2004. LNCS, vol. 3027, pp. 1–19. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24676-3_1
24. von zur Gathen, J., Panario, D.: Factoring polynomials over finite fields: a survey. J. Symb. Comput. **31**(1–2), 3–17 (2001)
25. Ghosh, S., Nilges, T.: An algebraic approach to maliciously secure private set intersection. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019. LNCS, vol. 11478, pp. 154–185. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17659-4_6
26. Ghosh, S., Simkin, M.: The communication complexity of threshold private set intersection. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019. LNCS, vol. 11693, pp. 3–29. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-26951-7_1
27. Hahn, F., Kerschbaum, F.: Searchable encryption with secure and efficient updates. In: ACM CCS (2014)
28. Inbar, R., Omri, E., Pinkas, B.: Efficient scalable multiparty private set-intersection via garbled bloom filters. In: Catalano, D., De Prisco, R. (eds.) SCN 2018. LNCS, vol. 11035, pp. 235–252. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98113-0_13
29. Kamara, S., Mohassel, P., Raykova, M.: Outsourcing multi-party computation. ePrint (2011)
30. Kamara, S., Mohassel, P., Raykova, M., Sadeghian, S.: Scaling private set intersection to billion-element sets. In: Christin, N., Safavi-Naini, R. (eds.) FC 2014. LNCS, vol. 8437, pp. 195–215. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45472-5_13

31. Kamara, S., Papamanthou, C.: Parallel and dynamic searchable symmetric encryption. In: Sadeghi, A.-R. (ed.) FC 2013. LNCS, vol. 7859, pp. 258–274. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39884-1_22
32. Katz, J., Lindell, Y.: Introduction to Modern Cryptography. CRC Press (2007)
33. Kerschbaum, F.: Outsourced private set intersection using homomorphic encryption. In: ASIACCS (2012)
34. Kissner, L., Song, D.: Privacy-preserving set operations. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 241–257. Springer, Heidelberg (2005). https://doi.org/10.1007/11535218_15
35. Knuth, D.E.: The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd edn. Addison-Wesley (1981)
36. Kolesnikov, V., Kumaresan, R., Rosulek, M., Trieu, N.: Efficient batched oblivious PRF with applications to private set intersection. In: CCS (2016)
37. Kolesnikov, V., Matania, N., Pinkas, B., Rosulek, M., Trieu, N.: Practical multiparty private set intersection from symmetric-key techniques. In: CCS (2017)
38. Liu, F., Ng, W.K., Zhang, W., Giang, D.H., Han, S.: Encrypted set intersection protocol for outsourced datasets. In: IC2E (2014)
39. Pinkas, B., Rosulek, M., Trieu, N., Yanai, A.: PSI from PaXoS: fast, malicious private set intersection. In: Canteaut, A., Ishai, Y. (eds.) EUROCRYPT 2020. LNCS, vol. 12106, pp. 739–767. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45724-2_25
40. Qiu, S., Liu, J., Shi, Y., Li, M., Wang, W.: Identity-based private matching over outsourced encrypted datasets. IEEE Trans. Cloud Comput. **6**(3), 747–759 (2018)
41. Silva, J.: Banking on the cloud: results from the 2020 cloudpath survey (2020). https://www.idc.com/getdoc.jsp?containerId=US45822120
42. Tsai, C.F., Hsiao, Y.C.: Combining multiple feature selection methods for stock prediction: union, intersection, and multi-intersection approaches. Decis. Support Syst. **50**(1), 258–269 (2010)
43. Citrin, A.V., Sprott, D.E., Silverman, S.N., Stem Jr., D.E.: Adoption of internet shopping: the role of consumer innovativeness. Ind. Manag. Data Syst. **100**(7), 294–300 (2000)
44. Yang, X., Luo, X., Wang, X.A., Zhang, S.: Improved outsourced private set intersection protocol based on polynomial interpolation. Concurr. Comput. **30**(1), e4329 (2018)
45. Zhang, E., Liu, F., Lai, Q., Jin, G., Li, Y.: Efficient multi-party private set intersection against malicious adversaries. In: CCSW (2019)
46. Zhao, Y., Chow, S.S.M.: Can you find the one for me? Privacy-preserving matchmaking via threshold PSI. IACR Cryptology ePrint Archive (2018)
47. Zheng, Q., Xu, S.: Verifiable delegated set intersection operations on outsourced encrypted data. In: IC2E (2015)