Efficient Unbalanced Private Set Intersection with Continuous Updating

Biwen Chen*, Ze Jiang*, Xiaoguo Li*, Tao Xiang*, Debiao He†
* *College of Computer Science, Chongqing University*, Chongqing, China
macrochen@cqu.edu.cn, jiangze@stu.cqu.edu.cn, xiaoguosk@gmail.com, txiang@cqu.edu.cn
† *School of Cyber Science and Engineering, Wuhan University*, Wuhan, China
hedebiao@163.com
Corresponding Author: Tao Xiang (txiang@cqu.edu.cn)

***Abstract*—Unbalanced Private Set Intersection (PSI) is widely deployed in real-world applications, such as private contact tracing and online advertising, to privately compute the intersection of two parties' datasets with differing sizes. However, existing unbalanced PSI protocols either introduce leakage about the intersection or divide the PSI computation into discrete epochs, limiting their capability for dynamic scenarios. In this paper, we present uu-PSI, a novel *updatable* and *unbalanced* PSI protocol that supports continuous updates to the parties' sets. Our protocol utilizes the Distributed Point Function (DPF) to encode incremental updates as XOR-based shares. This enables efficient dynamic set management without additional computational overhead or disclosing individual set elements. To demonstrate the applicability of our protocol, we integrate uu-PSI into the Apache Kafka platform, enabling both the provider and the client to obtain real-time matching results despite ongoing updates to their datasets. The experimental results show that our uu-PSI protocol reduces computational and communication costs by 24× compared to existing state-of-the-art solutions. Specifically, adding 4,096 elements takes only 31 ms for a server managing one million elements, while adding 64 elements requires just 32 ms for a client handling one thousand elements.**

***Index Terms*—private set intersection, unbalanced data sets, real-time data processing**

I. Introduction

Unbalanced PSI is a cryptographic protocol that securely computes the intersection of two datasets of significantly unequal size, revealing only the common elements. Unbalanced PSI is vital in applications such as private data mining [1], [2], measuring advertising conversion rates [3]–[6], and performing secure computations on medical data [7], [8].

Many studies have explored unbalanced PSI in the past few years. A great number of works use fully homomorphic encryption to reduce communication costs but impose significant computational overhead on the server [9]–[13]. The above works did not consider handling dynamic sets. However, many real-world applications involve scenarios where the datasets are constantly being updated. Examples of such scenarios include private contact discovery [4], [14], biometric authentication [15], matching personal preferences [16], exploring social networks [17], moderating content [18], and tracking contacts for health surveillance [12], [19]. Thus, there is a critical need for PSI protocols that efficiently handle imbalanced and dynamically updating datasets while maintaining low latency and high performance.

979-8-3503-5171-2/24/31.00©2024 IEEE

A. Prior Works

A naive solution of updatable PSI was proposed in earlier works of [23] and [21] based on the offline/online model, which set up data in advance, reducing online phase effort. [23] and [21] update the dataset by adjusting the positions of new elements in the data structure, but this can risk leaking sensitive information. Specifically, as noted by [20], the protocols in [21], [23] may inadvertently reveal the insertion time of the same element in the other party's set, thereby compromising privacy. To maintain security, these protocols need to repeat the preprocessing for any changes to the server's set, which is not suitable for server-side streaming.

A Decisional Diffie-Hellman (DDH) based approach [20] formalizes the updatable PSI's ideal functionality and gives methods to update the dataset. Unlike our work, their construction cannot handle the unbalanced situation. [22] develops a PSI weight cardinality (PSI-WCA) protocol that outputs the sum of the elements in the intersection. The key building block of [22] is the DPF, thus avoiding expensive public-key cryptography. [20], [22] involve dividing the task into smaller intervals (or epochs), which may cause delays in applications where some applications are not tolerant of such delays, such as in disease exposure tracking or personal credential breach detection [24]. Moreover, these approaches can also lead to inefficiencies when there are few or no new updates in a given epoch, as the system still requires running PSI with dummy elements, leading to wasted resources.

Our Motivation. In summary, existing approaches to this problem either have leakage on intersection or divide the PSI execution into epochs. This raises the following question:

"Can we design an unbalanced and updatable PSI scheme without intersection leakage and dividing into epochs?"

B. Our Contribution

In this paper, we affirmatively answer the above question and outline our contributions below.

- We introduce a novel unbalanced and updatable PSI protocol that is secure and efficient without segmenting into

TABLE I: Comparison among unbalanced/updatable PSI protocols

	Cryptography tools	Communication	Client's computation	Servers	Unbalanced	Updatable	Real-Time Update
BMX22 [20]*	DDH	$O(N \log N)$	$O(N \log N)$	1-server	✗	✓	✗
RA21 [21]	DDH	$O(N)$	$O(n)$	1-server	✓	✗	✗
DIL+22 [22]	DPF	$O(n)$	$O(n)$	2-server	✓	✓	✗
WH23 [12]	FHE&DDH	$O(n \log N)$	$O(n)$	1-server	✓	✗	✗
Ours	DPF&DDH	$O(n)$	$O(n)$	2-server	✓	✓	✓

*We take the one-sided version into consideration.

smaller sub-PSI tasks. By combining Oblivious Pseudo-Random Function (OPRF) and DPF, our protocol addresses data leakage, update delays, and excessive communication, ensuring robust privacy for real-time applications.
- Our protocol integrates seamlessly with Apache Kafka [25], leveraging its scalable and flexible architecture to manage large-scale stream submissions, meeting dynamic data processing requirements.
- We formalize the ideal functionality for an updatable PSI protocol and provide formal proof of its security supported by practical experiments. Our results show our protocol outperforms existing models in efficiency, scalability, and security, updating approximately $15\times$ faster and reducing communication overhead by about $24\times$ compared to state-of-the-art protocols [20], [22], [26], [27].

Achieving secure update PSI is challenging with only two parties: the server (with the larger set) and the client (with the smaller set). Existing methods typically involve either the client receiving an encoding of the server's set during the offline phase and extracting the result from this encoding [13], [21], [23] or the client sending an encoding of its set to the server, which returns the result. In the first approach, any updates to the sets require updates to the encoding, allowing the client to compare the original and updated encodings to infer changes in the server's set, thus compromising security. In the second approach, using homomorphic encryption or OPRF imposes significant computational overhead. Homomorphic encryption is costly, and OPRF requires the client to compute over all server elements, resulting in a prohibitive cost. Therefore, we introduce new participants in this work to distribute the security and efficiency burdens, making secure dynamic updates in unbalanced PSI scenarios feasible.

A comprehensive comparison of the existing techniques, along with our proposed improvements that address their limitations, is presented in Table I. [21] and [12] offer unbalanced PSI solutions. But these methods do not support updates. [20] focuses on balanced scenarios, resulting in a high computational cost for the client and lacking support for real-time updates. Given that our protocol reveals results to only one party, we also compare against the one-sided PSI solution from [20]. Although [22] provides an unbalanced and updatable PSI scheme, it is a PSI-WCA protocol and does not accommodate real-time updates. Our proposed solution achieves real-time updates in the unbalanced setting without compromising efficiency.

II. Preliminaries

A. Notations

We begin by introducing the notations that are commonly used in this paper. We use λ and σ to denote the computational and statistical security parameters, respectively. The sequence from 1 to m is represented by $[m] = \{1, \ldots, m\}$. The group, denoted by $\mathbb{G}$, has a prime order q and a generator g. The hash function H, which maps a string of bits to an element of the group, is defined as $H : \{0,1\}^* \to \mathbb{G}$. Let $\mathbf{s}$ denote a vector, where $\mathbf{s}[i]$ refers to its i^{th} element. The symbol $\oplus$ represents modular addition.

B. Cuckoo hash table

Cuckoo hashing [28] uses multiple hash functions—typically two or three—to determine potential locations for storing an item in a hash table. Each hash function provides a different "bin" or slot where the item might be placed. Unlike traditional hashing methods, where a bin can hold multiple items, cuckoo hashing restricts each bin to just one item. We define the relocate and insert operations for a cuckoo hash table T with two hash functions, h_1 and h_2, as follows:
- $\mathsf{Relocate}(x) \to \{0,1\}$. Given an input element x, the algorithm randomly selects an index $h_i(x)$ for $i \in \{1,2\}$ and retrieves the element $s = T[h_i(x)]$. If either of the indices $h_i(s)$ for $i \in \{1,2\}$ in T is vacant, the algorithm assigns $T[h_i(s)] = s$ and returns 1. Otherwise, it recursively calls $\mathsf{Relocate}(s)$. This recursive process is subject to a predefined maximum number of iterations. If the insertion fails to complete within this limit, the algorithm returns 0.
- $\mathsf{Insert}(x) \to \{0,1\}$. For a given input x, if either position $T[h_i(x)]$ is empty, the algorithm sets $T[h_i(x)] = x$ and returns 1. If both positions are occupied, it proceeds by invoking $\mathsf{Relocate}(x)$.

However, if the relocate operation fails, the problematic item is moved to a designated area known as a "stash". According to [29], using two hash functions along with a modestly sized stash usually ensures successful placements with few relocate operations, particularly in larger tables.

C. Oblivious pseudorandom function

OPRF [30] is a cryptographic protocol enabling two parties, such as a client and a server, to collaboratively compute a PRF without revealing their respective inputs to each other. Specifically, the server inputs a value x while the client inputs

a secret key k. The protocol then outputs $f_k(x)$ to the client, where $f : \{0,1\}^\kappa \times \{0,1\}^\ell \to \{0,1\}^\ell$ denotes a PRF.

D. Distributed Point Function

Secret sharing is a cryptographic technique that distributes a secret among multiple parties, such that the secret can only be reconstructed when a sufficient number of shares are combined. A point function $f_{\alpha,\beta}(\cdot)$ is a specific type of function that outputs a designated value β for a particular input α and returns 0 for all other inputs. DPF is a secret-sharing technique specifically designed for point functions. It divides a point function into two additive shares, (f_0, f_1), each represented by a compact key. This structure enables efficient computation and transmission while preserving the privacy of the original function. A DPF consists of the following two algorithms:

- $\mathsf{Gen}(1^\lambda, \alpha, \beta) \to (k_0, k_1)$. Given a security parameter 1^λ and two special input α, β, this algorithm generates a pair of keys (k_0, k_1).
- $\mathsf{Eval}(k, x) \to y$. Given an input x and a key k, this evaluation algorithm computes the output y_0 or y_1 of the share function represented by either key k_0 or k_1, where $y_0 + y_1 = f_{\alpha,\beta}(x)$

A DPF scheme satisfies the following properties: 1) *Correctness:* The reconstructed function $f_{\alpha,\beta}(x) = f_0(x) + f_1(x)$ yields the correct output β for input α and 0 for other inputs, where f_0 and f_1 are represented by the keys output by Gen algorithm. 2) *Secrecy:* The shares f_0 and f_1 do not reveal any information about α or β when considered individually.

III. Problem Statement

In this section, we first present the system model and design goals of this work. Then, we formalize the potential threats in a security model.

A. System Model

Our system, illustrated in Fig. 1, comprises three types of parties: the data provider, clients, and two servers.

- *Data Provider*. The data provider is a contributor responsible for sending out encrypted data streams to two servers.
- *Client*. The client sends encrypted streaming messages to two servers. It aims to determine the intersection between its inputs and the data provider's input.
- *Two servers*. The two servers are core computational components in our system, which securely operate on the data provider's data streams and the client's data streams and then generate the intersection results.

B. Design Goals

We aim to bring effective PSI service to the data provider and the clients while preserving data security for all parties. In particular, we want to achieve the following system goals in uu-PSI:

- **Real-time updating:** The system should support real-time updates, allowing clients to see the updated intersection results immediately after the data provider or clients update their sets. This ensures that the system remains responsive and up-to-date with minimal delay.
- **Data confidentiality:** No party except corresponding clients can learn the intersection result. The update process should not threaten the privacy of the origin and new elements. Moreover, the update process should not leak information more than the PSI ideal functionality allows.
- **Correctness:** The system must ensure that the intersection result is accurate and correctly reflects the intersection of the data provider's and client's sets, both before and after any updates.

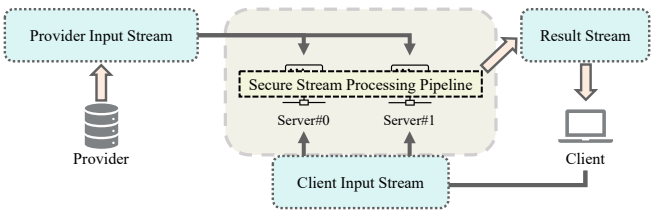


Fig. 1: The workflow of our protocol

C. Security Assumption

We assume all the parties in our system are semi-honest and will correctly follow the protocol but attempt to learn additional information from received messages. Specifically, a server may attempt to infer sensitive information about each data owner's data from the provider's encoded values and clients' queries it processes. The data provider may seek to learn about the client's set when the client interacts with her. The clients attempt to learn the data provider's dataset beyond the intended scope of interaction from the server's or provider's response.

Next, we assume the two servers would not collude with each other. To mitigate this, we can utilize cloud service providers like Microsoft Azure and Amazon AWS to deploy these servers. Given their substantial reputational stakes, these platforms are assumed to operate independently and maintain non-colluding behavior The non-colluding servers can protect data from a central point of attack [31]. This non-colluding servers setup is a widely accepted approach in PSI [22], [32], [33]. Furthermore, the use of non-colluding servers is increasingly moving toward practical deployment, as seen in initiatives by organizations [34]–[36]. This trend underscores the growing recognition of the importance of maintaining independent and secure server operations in privacy-preserving protocols.

D. Problem Formulation and Definitions

In uu-PSI, both the data provider and the client maintain private sets that are frequently updated with new elements. Their objective is to jointly compute the intersection of their sets immediately upon receiving these updates without revealing any additional information beyond the intersection itself.

The ideal functionality of uu-PSI is formalized in Fig. 2, which illustrates our simulation-based security definition. This

Ideal Functionality

Parameters: The set size of n, n', N, N' for set X, X', Y, Y' respectively where $n' \ll n, N' \ll N$ and $n \ll N$
Inputs:
Client inputs set X at time t_0.
Provider inputs set Y at time t_0.
Output at t_0: P_0 receives the set intersection $X \cap Y$
Updates:
Client updates set $X \coloneqq X \cup X'$ at time t_1.
Provider updates set $Y \coloneqq Y \cup Y'$ at time t_1.
Output at t_1: Client receives the set intersection $X \cap Y$

Fig. 2: Ideal functionality $\mathcal{F}_{uu\text{-}PSI}$ for uu-PSI

functionality precisely specifies the information each party is entitled to learn at the end of the protocol, encapsulating the design objectives of our system. It supports real-time updates by providing the client with the intersection results immediately upon any change, thereby ensuring promptness and minimal delay in reflecting updates to the data sets. It preserves data confidentiality by restricting the client's knowledge solely to the intersection, while securely managing updates to prevent leakage of any further information beyond what is permitted by the ideal PSI functionality. Additionally, the functionality guarantees correctness by accurately computing and delivering the intersection results to the client after every update.

Definition 1 (Updatable and Unbalanced PSI)**.** *Let X^* and Y^* be the set $\{X, X'\}$ and $\{Y, Y'\}$. Let $N^* = \{N, N'\}$ and $n^* = \{n, n'\}$. Let $View_0^{\Pi}(X^*, Y^*)$ and $View_1^{\Pi}(X^*, Y^*)$, $View_2^{\Pi}(X^*, Y^*)$ be the view of the data provider, the client and the servers in the protocol Π, respectively. Let $Out^{\Pi}(X^*, Y^*)$ and $O = f(X^*, Y^*)$ be the output of the client in the protocol Π and ideal functionality, respectively.*

We say a protocol Π is semi-honest secure with respect to ideal functionality $\mathcal{F}_{uu\text{-}PSI}$ if there exists PPT simulators Sim_0, Sim_1 and Sim_2 such that for any inputs (X^*, Y^*),

$$\begin{aligned} View_0^{\Pi}(X^*, Y^*) &\overset{c}{\approx} Sim_0(1^{\lambda}, Y^*, O), \\ (View_1^{\Pi}(X^*, Y^*), Out^{\Pi}(X^*, Y^*)) &\overset{c}{\approx} (Sim_1(1^{\lambda}, X^*, N^*), O), \\ View_2^{\Pi}(X^*, Y^*) &\overset{c}{\approx} Sim_2(1^{\lambda}, n^*, N^*, O), \end{aligned}$$

where '$\overset{c}{\approx}$' represents that two distributions are computationally indistinguishable.

IV. Construction

In this section, we first introduce the core of our design (i.e., Distributed Oblivious Pseudo-Random Function, DOPRF), and then present the workflow of our protocol.

A. Distributed OPRF

The foundation of design is a distributed OPRF. Unlike conventional OPRF presented in section II-C, DOPRF can distribute the functionality of the single server to multiple parties. For our design, these parties involve a provider and two servers. This brings the following two benefits:

First, this distribution minimizes unnecessary interactions between the provider and the client. In a single-server setup, the provider and client must frequently interact, which can introduce inefficiencies and potential security vulnerabilities. Dividing the server role can streamline interactions and enhance overall efficiency.

Second, distributing the server role increases resilience against key compromises. In a single-server model, a compromise of the server's key can threaten the entire system's security. With multiple servers, even if one server's key is compromised, the overall system can still maintain a higher level of security. This is crucial when using OPRF to protect low-entropy inputs, where the risk of brute-force attacks is higher.

In summary, transitioning from a single-server OPRF to a distributed server OPRF not only aligns better with real-world application scenarios but also enhances the security and efficiency of the OPRF system.

We now describe the oblivious computation of PRF $f_k(X)$ among the provider, two servers, and the client according to [37]:

1) Provider generates two additive shares k_0, k_1 of k where $k = k_0 + k_1$; sends k_0 to server#0 and k_1 to server #1.
2) On input $x \in \{0,1\}^*$, client chooses $r \xleftarrow{\$} \mathbb{Z}_q$; sends $a = H(x)^r$ to both servers.
3) Server #i verifies that the received a is in group $\mathbb{G}$ and if so it responds with $b_i = a^{k_i}$.
4) Client outputs $f_k(x) = (b_0 \cdot b_1)^{1/r}$.

B. Protocol Description

Our protocol consists of three phases: Initialization, Producer streaming, and Client streaming.

Initialization. This phase involves three main algorithms: Set-up, Online OPRF, and Online DPF.

Set-up. The Set-up Algorithm involves the initialization of the provider and servers. At the start, the provider begins by selecting an OPRF key $k_p \xleftarrow{\$} \mathbb{Z}_q$. The provider then inputs k_p and its set Y into the set-up algorithm. Upon completion, server#0 and server#1 will each obtain the encoded set $\hat{Y}$ and additive shares of k_p. The details of this algorithm are presented in Algorithm 1. The algorithm consists of two main steps: first, the provider splits k_p into two additive shares k_{p0} and k_{p1} such that $k_p = k_{p0} + k_{p1}$ (line 1); second, the provider uses k_p to generate the encoded set $\hat{Y}$ (lines 2-4).

Online OPRF. The Online OPRF Algorithm enables the client to obtain the encodings of its set elements. The client begins by randomly selecting $\alpha \xleftarrow{\$} \mathbb{Z}_q$, and inputs α and X into the Online OPRF Algorithm, which is detailed in Algorithm 2. This algorithm is divided into three main steps: first, the

Algorithm 1 Set-up

Inputs: Provider inputs k_p and Y
Outputs: Server#0 obtains k_{p0} and $\hat{Y}$; server#1 obtains k_{p1} and $\hat{Y}$
1: Provider samples a random k_{p0} from $\mathbb{Z}_q$ and computes $k_{p1} = (k_p - k_{p0}) \mod p$
2: **for** $i = 1$ to $|Y|$ **do**
3: Provider computes $\hat{Y}_i = H(Y_i)^{k_p}$
4: **end for**
5: Provider sends k_{p0} and $\hat{Y}$ to server#0, and k_{p1} and $\hat{Y}$ to server#1

client masks its elements using α, generating $\tilde{X}$, which is then sent to server#0 and server#1; second, server#0 and server#1 encode $\tilde{X}$ using their respective shares k_{p0} and k_{p1}, producing $\tilde{X}^0$ and $\tilde{X}^1$, which are sent back to the client; finally, the client reconstructs $\hat{X}$ using $\tilde{X}^0$ and $\tilde{X}^1$.

Algorithm 2 Online OPRF

Inputs: Client inputs α and X
Output: Client obtains $\hat{X}$
1: **for** $i = 1$ to $|X|$ **do**
2: Client computes $\tilde{X}_i = H(X_i)^{\alpha}$
3: **end for**
4: Client sends $\tilde{X} = \{\tilde{x}_0, ..., \tilde{x}_{|X|}\}$ to server#0 and server#1
5: **for** $i = 1$ to $|X|$ **do**
6: Server#0 computes $\tilde{X}_i^0 = \tilde{X}_i^{k_{p0}}$
7: Server#1 computes $\tilde{X}_i^1 = \tilde{X}_i^{k_{p1}}$
8: **end for**
9: Server#0 sends $\tilde{X}^0$ to client
10: Server#1 sends $\tilde{X}^1$ to client
11: **for** $i = 1$ to $|X|$ **do**
12: Client computes $\hat{X}_i = (\tilde{X}_i^0 \cdot \tilde{X}_i^1)^{-\alpha}$
13: **end for**

Online DPF. The Online DPF Algorithm is used for private matching. The client initializes by preparing a table T and inputs both T and the set X into the Online DPF Algorithm. Upon the algorithm's completion, the client receives the output $\mathbf{r}$. The detailed procedures of this algorithm are presented in Algorithm 3. The algorithm is structured into three key steps: first, the client inserts elements into T using a cuckoo hashing technique; second, the client generates DPF keys K^0 and K^1 using the elements of T as inputs; third, server#0 and server#1 perform DPF evaluations using the encoded set $\hat{Y}$ and the DPF keys, resulting in $\mathbf{r}_0$ and $\mathbf{r}_1$. Finally, the client combines $\mathbf{r}_0$ and $\mathbf{r}_1$ to derive the final result $\mathbf{r}$.

Producer Streaming. The provider can efficiently update its set by adding elements while maintaining privacy and minimizing communication overhead. When provider decides to add new elements to their set, this can be accomplished by first encrypting the elements using the same pseudo-random function as in line 3 of Algorithm 1. The addition of these new elements triggers an update process, wherein both servers and the client recalculate the matching results to incorporate the

Algorithm 3 Online DPF

Inputs: Client inputs $\hat{X}$, T, h_1 and h_2
Output: Client obtains $\mathbf{r}$
1: **for** $i = 1$ to $|\hat{X}|$ **do**
2: Client do $T.\mathsf{Insert}(\hat{X}_i)$
3: **end for**
4: **for** $i = 1, 2 \ldots |T|$ **do**
5: **if** $T[i]$ is not empty **then**
6: Let $d \leftarrow T[i]$
7: **else**
8: Let $d \xleftarrow{\$} \mathbb{Z}_q$
9: **end if**
10: Client computes $K_i^0, K_i^1 \leftarrow \mathsf{DPF.Gen}(1^{\lambda}, d, 1)$
11: **end for**
12: Client sends K^0 to server#0 and K^1 to server#1
13: **for** $i = 1, 2 \ldots |\hat{Y}|$ **do**
14: **for** $j = 1, 2$ **do**
15: Server#0 calculates $p_j = h_j(\hat{y_i})$
16: Server#0 computes $d_j = \mathsf{DPF.Eval}(1^{\lambda}, K_{p_j}^0, \hat{y_i})$
17: Server#0 computes $\mathbf{r}_0[p_j] = r_0[p_j] \oplus d_j$
18: **end for**
19: **end for**
20: Server#0 sends $\mathbf{r}_0$ to client
21: Symmetrically, server#1 computes and sends $\mathbf{r}_1$ to client
22: Client computes result $\mathbf{r} = \mathbf{r}_0 \oplus \mathbf{r}_1$

updated data. As the new elements are processed, server#0 performs DPF evaluations in line 16 of Algorithm 3. To optimize communication, only the altered bits and their positions in $\mathbf{r}_0$ are transmitted. Similarly, server#1 computes and transmits the corresponding essential information.

For element removal, when provider wishes to delete an element, this can be achieved by notifying the servers to remove the specified element, ensuring that it no longer appears in future computation results. However, for elements that have already appeared in the intersection, provider is unable to identify or remove them, highlighting a critical limitation of the current approach.

Client Streaming. The client can dynamically update its set while preserving data privacy and ensuring synchronization with the provider's updates. When client adds new elements to their set, these elements are first encrypted through the online OPRF phase, which preserves the privacy of both the client's and the provider's data. Following this, client performs the operations outlined in line 2 of Algorithm 3 to determine the appropriate positions for these elements within the cuckoo table. New DPF keys are generated for these elements and communicated to server#0 and server#1, along with information about their placement within the data structure. Upon receiving updates from either the provider or client, the servers undertake computations to evaluate the impact of the new elements on the matching results, which involves locating the updated positions within the cuckoo table and recalculating the results to accurately reflect the current

state of both datasets.

For client-side deletions, if an element is part of the intersection, it is directly removed from the result. A straightforward approach might involve identifying the position of the deleted element within the cuckoo table T and overwriting this position with a random DPF key. However, this method only updates one of the two positions associated with the element in the cuckoo table, potentially exposing sensitive information about the client's data. Developing a more secure and comprehensive method for element removal that addresses these privacy concerns remains an open challenge and is left as future work.

C. Correctness and Efficiency

1) Correctness.: If the parties follow the protocol honestly, at the end of the protocol, client will learn the intersection between his set and provider's set with all but negligible probability. The intuition is that if x is a match of y, say, $x = y$, it will hold that $f(k, x) = f(k, y)$. So $f(k, x)$ will be mapped to either $h_0(f(k, x))$ or $h_1(f(k, x))$. In other words, either $K^i[h_0(f(k, y))]$ or $K^i[h_1(f(k, y))]$ held by one server will be the DPF key generated by $f(k, x)$. Let h_1 be the mapped index of $f(k, x)$, $r_0[h_1]$ and $r_1[h_1]$ will be secret shares computed from $\mathsf{DPF.Eval}(K^i[h_1], f(k, y))$. According to the fact that $f(k, x) = f(k, y)$, $r[h_1] = r_0[h_1] \oplus r_1[h_1]$ will be 1 which represent a match. In contrast, if $x \neq y$, it will hold that $f(k, x) \neq f(k, y)$ except for negligible probability. $r[h_1] = r_0[h_1] \oplus r_1[h_1]$ will be 0 because of the process same as presented above. Note that there may be a negligible probability of false positives due to the PRF collision.

2) Computation and Communication Complexity.: In the initialization phase, the protocol incurs a computational cost of $O(N)$ for the provider, a computational cost of $O(n)$ and communication cost of $O(n)$ for the client, and a computational cost of $O(N)$ for each server. More specifically, the provider's cost is $O(N)$ because it must compute the PRF values for all elements, where N is the total number of elements. The client needs to generate $2n$ DPF keys and send them to both server#0 and server#1, resulting in both computational and communication costs of $O(n)$. Each server performs two $\mathsf{DPF.Eval}$ operations for each PRF value it receives, leading to a computational cost of $O(N)$.

The provider incurs a computational complexity of $O(N')$, where N' is the number of new elements in provider streaming, as it computes the new PRF values for these elements. The client faces both computational and communication complexities of $O(n')$ in client streaming since it must generate and transmit new DPF keys for the added elements. Each server maintains a computational complexity of $O(N')$, as it performs $\mathsf{DPF.Eval}$ operations for both the new PRF values and any related old values mapped to the updated indices.

D. Security

Theorem 1. *Given a random oracle H, a secure OPRF and a DPF scheme, assuming that one-way function exists, and DDH assumption is hard, the protocol Π_{uu-PSI} presented in section IV-B securely realizes the ideal functionality $\mathcal{F}_{uu\text{-}PSI}$.*

Proof. Let $\mathcal{A}$ be a PPT adversary that allows the corruption of one of the servers or the client. We construct PPT simulators Sim_0, Sim_1 and Sim_2 with access to functionality $\mathcal{F}_{uuPSI}$, which simulates the adversary's view. We consider the following three cases: semi-honest provider, semi-honest client, and semi-honest server. We will prove that the joint distribution over the output of $\mathcal{A}$ and the honest party in the real world is indistinguishable from the joint distribution over the outputs of Sim_0, Sim_1 and Sim_2 and the honest party in the ideal world execution.

1) Security against corrupted provider. : Let Sim_0 access to $\mathcal{F}_{uu\text{-}PSI}$ as an honest provider and interact with $\mathcal{A}$ as an honest server. As the provider does not receive any message during the protocol execution, we have, for any input X^*, Y^*,

$$\text{View}_0^{\Pi}(X^*, Y^*) \overset{c}{\approx} \text{Sim}_0(1^\lambda, Y^*, O)$$

2) Security against corrupted client. : Let Sim_1 access to $\mathcal{F}_{uu\text{-}PSI}$ as an honest client and interact with $\mathcal{A}$ as an honest server with the following exception:

- In line 9-10 of Algorithm 2, let $\hat{X}^b \overset{\$}{\leftarrow} \mathbb{G}^n$ and send $\hat{Y}$ to the client on behalf of server#b.
- In line 20-21 of Algorithm 3, let $\mathbf{r} \overset{\$}{\leftarrow} \{0,1\}^{|T|}$ and send $\mathbf{r}$ to the client.

Finally, Sim_1 outputs client's view. Under hybrid argument, we show for any input X^*, Y^*,

$$(\text{View}_1^{\Pi}(X^*, Y^*), \text{Out}^{\Pi}(X^*, Y^*)) \overset{c}{\approx} (\text{Sim}_1(1^\lambda, X^*, N^*), O).$$

Hybrid $\mathcal{H}_0$. Same as real-world execution.

Hybrid $\mathcal{H}_1$**.** This hybrid is identical to $\mathcal{H}_0$, except that the client's output is substituted with $O(X, Y)$, representing the output of the ideal functionality. This substitution is computationally indistinguishable from $\mathcal{H}_0$ due to the protocol's correctness, as demonstrated in Section IV-C.

Hybrid $\mathcal{H}_2$. Same as $\mathcal{H}_1$, but $\hat{Y}$ is replaced with a set of random group elements. From $\mathcal{H}_1$ to $\mathcal{H}_2$, we actually replace elements one by one via a sequence of hybrids $\mathcal{H}_{1,0}, \mathcal{H}_{1,1}, ..., \mathcal{H}_{1,n}$ where $\mathcal{H}_{1,0} = \mathcal{H}_1$ and $\mathcal{H}_{1,n} = \mathcal{H}_2$. We argue that every pair of adjacent hybrids is computationally indistinguishable based on the DDH assumption.

Assume for the purpose of contradiction that there exists a PPT algorithm $\mathcal{D}$ that can distinguish two adjacent hybrids $\mathcal{H}_{1,i}$ and $\mathcal{H}_{1,i+1}$ where $H(\tilde{x})^r k$ is replaced by a random element. We build a PPT distinguisher $\mathcal{B}$ to break the DDH assumption. $\mathcal{B}$ is given a tuple (g_1, g_2, g_3) where $g_1 = g^u, g_2 = g^v$ for random $u, v \in \mathbb{Z}_q$ and g_3 is either g^{uv} or g^z for a random $z \overset{\$}{\leftarrow} \mathbb{Z}_q$. $\mathcal{B}$ generates $\mathcal{D}$'s view as in $\mathcal{H}_{1,i}$ but sets $r \cdot k \coloneqq u$ (although u is unknown) and $H(\tilde{x}) = g_2$.

In particular, whenever $H(\cdot)$ is computed. $\mathcal{B}$ samples a random $s \in \mathbb{Z}_q$ and sets the output to be g^s. When server need to compute $H(x)^{rk}$, since $\mathcal{B}$ knows k as well as $s \in \mathbb{Z}_q$ such that $H(y) = g^s$, it can compute $H(y)^{rk}$ as g_1^{sk}; for $\tilde{y}$, $\mathcal{B}$ replaces $H(\tilde{y})$ with g_3^k.

If $g_3 = g^{uv}$, then $\mathcal{B}$ generates client's view as in $\mathcal{H}_{1,i}$; otherwise g_3^k is a random group element, hence $\mathcal{B}$ generates client's view as in $\mathcal{H}_{1,i+1}$. Since $\mathcal{D}$ can distinguish these two hybrids, $\mathcal{B}$ can break the DDH assumption. Contradiction.

Hybrid $\mathcal{H}_3$. Same as $\mathcal{H}_2$ but in lines 20-21 of Algorithm 3, server sends random $\mathbf{r} \in \{0,1\}^{|T|}$. $\mathbf{r}$ is computationally indistinguishable from the original $\mathbf{r}_b$ if the DPF scheme is secure. Hence, these two hybrids are computationally indistinguishable. We claim that the client's view in this hybrid is exactly Sim_1's output. This concludes the proof.

3) Security against corrupted server. : Let Sim_2 access to $\mathcal{F}_{\textit{uu-PSI}}$ as an honest server and interact with $\mathcal{A}$ as an honest client and an honest data provider with the following exception:

- In line 5 of Algorithm 1, send random element $r \xleftarrow{\$} \mathbb{Z}_q$ to $\mathcal{A}$ on behalf of provider.
- In line 4 of Algorithm 2, let $R \xleftarrow{\$} \mathbb{G}^n$ and send R to $\mathcal{A}$.
- In line 12 of Algorithm 3, let $K = \{k_i^b \leftarrow \mathsf{DPF.Gen}(\alpha_i)\}$ for all $i \in [|T|]$ to $\mathcal{A}$ where $\alpha_i \xleftarrow{\$} \mathbb{G}$.

Finally, Sim_2 output server's view. We are going to show that the simulated execution is indistinguishable from the real-world execution via a hybrid argument. That is, for any input (X^*, Y^*), we have,

$$\text{View}_2^{\Pi}(X^*, Y^*) \overset{c}{\approx} \text{Sim}_2(1^\lambda, n^*, N^*, O).$$

Hybrid $\mathcal{H}_0$. Same as real-world execution.

Hybrid $\mathcal{H}_1$. Same as $\mathcal{H}_0$, but H is replaced with a random function. This is computationally indistinguishable from $\mathcal{H}_0$ because H is modeled as a random oracle.

Hybrid $\mathcal{H}_2$. Same as $\mathcal{H}_1$ but replace the OPRF key share in line 1 of Algorothm 1 with randomly sampled element r. This is computationally indistinguishable from $\mathcal{H}_1$ because the origin key share is also randomly sampled.

Hybrid $\mathcal{H}_3$. Same as $\mathcal{H}_2$ but replace the $\tilde{X}_i = H(X_i)^r$ with $\tilde{X}_i = R_i$ for all $i \in [|X|]$ where $R \xleftarrow{\$} \mathbb{G}^n$. From $\mathcal{H}_2$ to $\mathcal{H}_3$, we actually replace elements one by one via a sequence of hybrids $\mathcal{H}_{2,0}$,$\mathcal{H}_{2,1}$,...,$\mathcal{H}_{2,n}$, where $\mathcal{H}_{2,0} = \mathcal{H}_2$ and $\mathcal{H}_{2,n} = \mathcal{H}_3$. We argue that every pair of adjacent hybrids is computationally indistinguishable based on the DDH assumption.

Assume for the purpose of contradiction that there exists a PPT algorithm $\mathcal{D}$ that can distinguish two adjacent hybrids $\mathcal{H}_{2,i}$, $\mathcal{H}_{2,i+1}$ where $H(\tilde{x})^r$ is replaced by a random group element. We build a PPT distinguisher $\mathcal{B}$ to break the DDH assumption. $\mathcal{B}$ is given a tuple (g_1, g_2, g_3) where $g_1 = g^u, g_2 = g^v$ for random $u, v \in \mathbb{Z}_q$ and g_3 is either g^{uv} or g^z for a random $z \xleftarrow{\$} \mathbb{Z}_q$. The purpose of $\mathcal{B}$ is to distinguish this tuple. $\mathcal{B}$ generates $\mathcal{A}$'s view as in $\mathcal{H}_{2,i}$ but sets $r := u$ (although u is unknown) and $H(\tilde{x}) = g_2$.

In particular, whenever $H(\cdot)$ is computed, $\mathcal{B}$ samples a random $s \in \mathbb{Z}_q$ and sets the output to be g^s. When client need to compute $H(x)^r$, since $\mathcal{B}$ knows $s \in \mathbb{Z}_q$ such that $H(x) = g^s$, it can compute $H(x)^r$ as g_1^s. For $\tilde{x}$, $\mathcal{B}$ replaces $H(\tilde{x})^u$ with g_3. If $g_3 = g^{uv}$, then $\mathcal{B}$ generates $\mathcal{A}$'s view as in $\mathcal{H}_{2,i}$; otherwise $\mathcal{B}$ generates $\mathcal{A}$'s view as in $\mathcal{H}_{2,i+1}$. Since $\mathcal{D}$ can distinguish these two hybrids, $\mathcal{B}$ can break the DDH assumption. Contradiction.

Hybrid $\mathcal{H}_4$. Same as $\mathcal{H}_3$ but replace the K^b with random DPF keys ($\mathsf{DPF.Gen}(1^\lambda, K_{p_j}^b, \alpha)$ where $\alpha \xleftarrow{\$} \mathbb{G}$) for all $i \in [|T|]$. From $\mathcal{H}_3$ to $\mathcal{H}_4$, we in fact replace the elements one by one via a sequence of $\mathcal{H}_{3,0}$,$\mathcal{H}_{3,1}$,...,$\mathcal{H}_{3,m}$ where $\mathcal{H}_{3,0} = \mathcal{H}_3$ and $\mathcal{H}_{3,n} = \mathcal{H}_4$. We argue that every pair of adjacent hybrids is computationally indistinguishable if the DPF scheme satisfies the secrecy requirement.

Assume for the purpose of contradiction that there exists a PPT distinguisher $\mathcal{D}$ that can distinguish two adjacent hybrids $\mathcal{H}_{3,i}$ and $\mathcal{H}_{3,i+1}$ where k_i^b is replaced with a DPF key generated from random value. We construct a PPT distinguisher $\mathcal{B}$ to break the DPF secrecy requirement.

$\mathcal{B}$ inputs a pair of random group elements $\alpha, \beta \in \mathbb{G}$, and is given a DPF key k. k is either generated from α or β. $\mathcal{B}$ set the ith element x_i and $\alpha \xleftarrow{\$} \mathbb{G}$ and get a output k. $\mathcal{B}$ generates the server's view in step 5 of initialization but sets the ith DPF key with input k. If k is generated from x_i, then $\mathcal{B}$ generates server's view as in $\mathcal{H}_{3,i}$; otherwise k is generated from random element, hence $\mathcal{B}$ generates server's view as in $\mathcal{H}_{3,i+1}$. Since $\mathcal{D}$ can distinguish these two hybrids, $\mathcal{B}$ can break the secrecy of DPF. Contradiction. We claim that the server's view in this hybrid is exactly Sim_2's output. This concludes the proof. □

V. Implementation and Benchmark

We implement uu-PSI in Java and report its performance in this section.

A. Implementation details

We set the computational security parameter to $\lambda = 64$. We use Java for most implementations, including BouncyCastle [38] library for group computation and SHA-256 for the hash function. We also use native codes in C++ for DPF via the Java native interface. In particular, we adopt the DPF implementation from [39]. We process elements with size of 128 bits.

We compare our protocol with previous works, including:

- KKRT16 [26]: well-known fast PSI from batched OPRF.
- RR22 [27]: computation-optimized and communication-optimized, works best in the setting of LAN networks.
- BMX22 [20]: computation-optimized updatable PSI, and works best in the network with moderate bandwidth (e.g., 5–50Mbps).
- DIL+22 [22]: updatable PSI with non-colluding servers.

We run the experiments between two machines where machine 1 has Intel Core I9-12900KF 3.10GHz and 128 GB RAM, and machine 2 has Intel Xeon Platinum 8375C 2.90GHz and 192GB RAM. The two machines communicate with local area network (LAN) with 10 Gbps of bandwidth and a 0.02 ms Round-Trip Time (RTT). We simulate wide area network (WAN) using Linus tc command where the RTT is set to be 80ms. And we test our experiments on various network bandwidths, including 200Mbps, 50Mbps, and 5Mbps.

We run an independent Kafka cluster on each server as its interface for handling data submission and retrieval requests,

as in Fig. 3. Specifically, Kafka will be used: 1) as a data pipeline for buffering data streams submitted by the data provider; 2) as a DPF key pipeline to push the generated DPF keys to each server (i.e., for completing the two-server stream DPF evaluation process); and 3) as a gateway for securely delivering result shares to the client. Each party (including the data provider and the client) will interact with the two Kafka clusters respectively via inherent Kafka APIs, and we will enable broker replications to boost the performance.

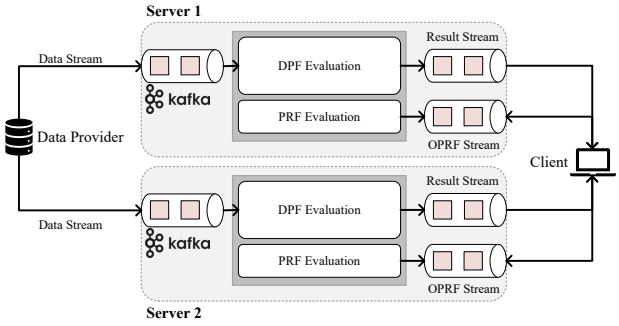


Fig. 3: Our PSI's integration of Kafka

Setting. To demonstrate the updatable property, we consider the following scenario: client and provider have initial input sets of sizes N and n, respectively. An update is defined as the process where the provider and client each input new sets of sizes N' and n', and then compute the resulting intersection. In KKRT16 [26] and RR22 [27], to handle updates, the parties perform a fresh PSI on their updated sets to compute the new intersection.

B. Initialization

We structure the initialization phase into two parts: offline encoding and online computation. The offline encoding, described in lines 2-4 of Algorithm 1, is so named because it is performed only once, and its outputs can be reused in subsequent operations. The online computation is defined by the procedures in Algorithms 2 and 3.

Offline Encoding. We have analyzed the communication and computational costs associated with the offline encoding phase. The results are presented as follows:

Computational cost. Similarly, the computational cost of offline encoding increases with the provider's input size N. We assessed this cost across a range of set sizes from 2^{16} to 2^{28}, with the results presented in Fig. 4. Note that this phase can be parallelized, and we have implemented parallelization. The results, also shown in the figure, reveal a $4.7\times$ improvement when increasing the number of threads from 1 to 6 and a $7\times$ improvement from 1 to 24 threads. The local encoding time is proportional to the size of the server's set.

Communicational cost. Offline encoding scales linearly with the provider's input size N. We construct a list of 128-bit values to represent the provider's set, and the outputs of the PRF are subsequently hashed into 64-bit strings. The communication overhead is depicted in Fig. 4.

After encoding, the data provider streams each element to servers via Kafka cluster, which is used as a buffer for coping with large-scale data stream submissions. Table II shows that increasing both the partition and broker numbers in Kafka can help further boost the performance.

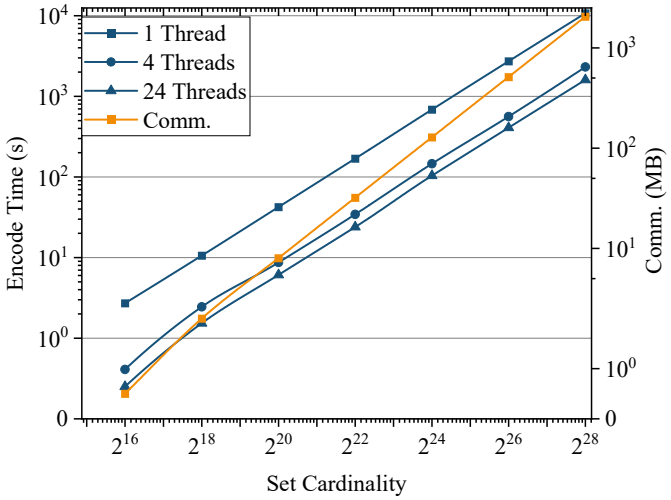


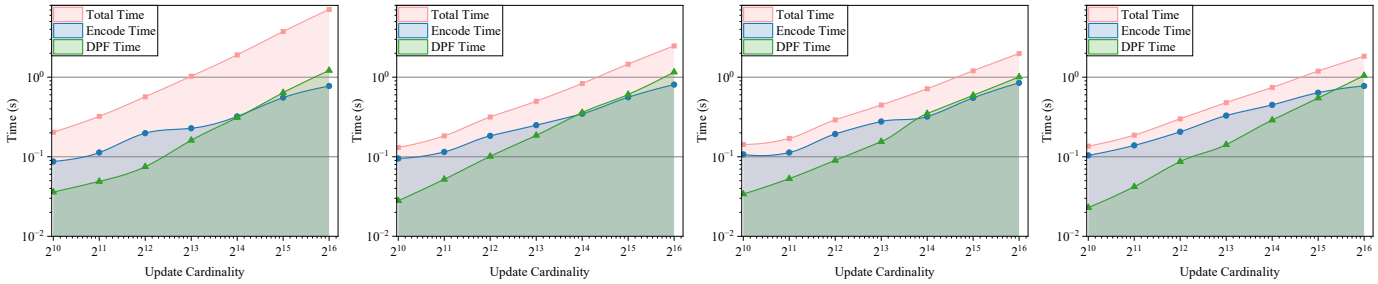
Fig. 4: Performance of offline encoding

TABLE II: Throughput (elements/second) vs. Kafka replications and broker partitions

Broker Partitions	Replication Factor 1	3
1	1.73×10^6	1.70×10^6
3	1.92×10^6	2.17×10^6
5	2.07×10^6	2.20×10^6
10	2.30×10^6	2.34×10^6

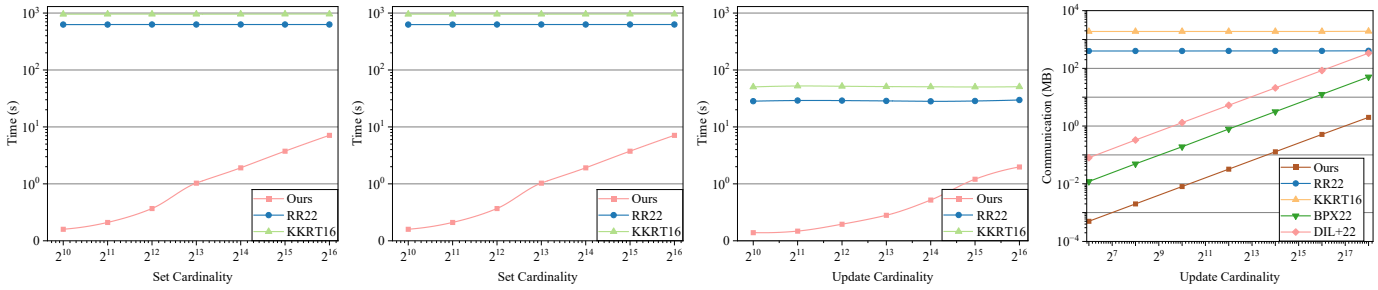
Online Computation The Cuckoo hashing parameters are selected based on the guidelines provided in [26]. The online computation involves performing $3 \cdot n$ elliptic curve multiplications, approximately $2 \cdot n$ DPF generation operations, and roughly $3 \cdot N$ DPF evaluation operations.

Computational cost. Table III presents the online computation times for various set sizes. The evaluations are conducted with provider set sizes $N \in \{2^{16}, 2^{18}, 2^{20}\}$ and client set sizes $n \in \{2^{10}, 2^{12}, 2^{14}\}$. For a provider set size of 2^{16}, the online computation time is under 0.4 seconds. For a provider set size of 2^{20}, the time is under 2 seconds, and for a provider set size of 2^{24}, it is under 22 seconds. The online running time exhibits approximately linear growth with the provider set size. It is noteworthy that the client's computational workload is minimal, comprising only DPF key generation and group multiplication, which scales linearly with the client set size. The client's computation cost represents less than 10% of the total online computation time (e.g., 0.07 seconds out of 1.7 seconds for $N = 2^{20}$ and $n = 2^{10}$). We identify



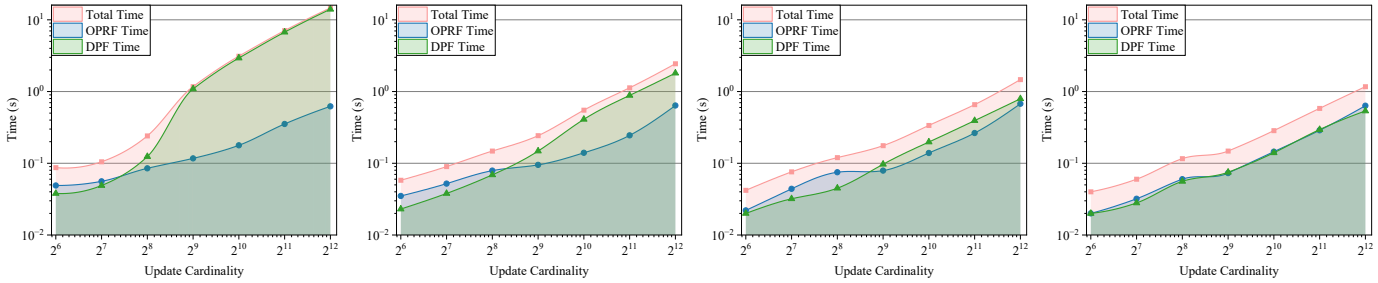
(a) Bandwidth=5Mbps (b) Bandwidth=50Mbps (c) Bandwidth=200Mbps (d) LAN

Fig. 5: Data provider update time under different bandwidth



(a) Bandwidth=5Mbps (b) Bandwidth=50Mbps (c) Bandwidth=200Mbps (d) Communication

Fig. 6: Comparison of update time and communication



(a) Bandwidth=5Mbps (b) Bandwidth=50Mbps (c) Bandwidth=200Mbps (d) LAN

Fig. 7: Comparison of update time and communication

that the predominant bottleneck in the online computation is the DPF evaluations performed by the server, accounting for approximately 90% of the total online time.

Communicational cost. The online communication cost scales linearly with the client set size. A key advantage of our protocol is that the client is not required to store the encoded larger set, which is particularly beneficial for devices with constrained storage capacity. During the online phase, each encoded elliptic curve point requires 17 bytes, while each DPF key occupies 1041 bytes for a 64-bit input length. Importantly, the communication cost in the online phase is independent of the server set size. Table III illustrates the communication overhead for various client set sizes.

C. Efficiency of Streaming

In this section, we evaluate the performance of both provider-side and client-side streaming, as detailed in Sections

TABLE III: The online running time and communication cost of PSI protocol

Cardinality		Online Time (s)	Communication (MB)
N	n		
2^{16}	2^{10}	0.19	2.05
	2^{12}	0.23	8.20
	2^{14}	0.44	32.80
2^{20}	2^{10}	1.70	2.05
	2^{12}	1.74	8.20
	2^{14}	1.95	32.80
2^{24}	2^{10}	21.95	2.05
	2^{12}	21.99	8.20
	2^{14}	22.20	32.80

IV-B and IV-B. We consider a streaming scenario where the parties have initially executed a PSI protocol with a set size of 2^{24}, and now they wish to update their original sets by adding new elements to compute the intersection of the updated sets.

We compared the provider-side streaming time of our protocol against the methods in KKRT16 [26] and RR22 [27]. For KKRT16 [26], we used the open-source implementation from [40], and for RR22 [27], we used the implementation from [41]. And we compare the communication cost against KKRT16 [26], RR22 [27], BMX22 [20], and DIL+22 [22].

Provider-side Streaming. The computational cost for provider-side streaming scales linearly with the number of newly added elements. Fig. 5 illustrates the provider-side update performance across various bandwidths, with update sizes ranging from 2^{10} to 2^{16}. At a bandwidth of 5 Mbps, communication times are relatively high due to bandwidth constraints. However, as the bandwidth increases, the computational time for provider-side streaming improves by a factor of $3.9\times$.

The results from Fig. 6 indicate that, under varying network bandwidth conditions, our protocol exhibits a significant computational advantage, exceeding $15\times$ compared to KKRT16 [26] and RR22 [27]. This advantage arises because our protocol's computational complexity for updates depends solely on the number of new elements, while theirs depends on the size of the entire set. However, as the number of updated elements increases, the runtime of our protocol may surpass theirs, as the advantage of uu-PSI diminishes with a larger number of updates. Moreover, it can be observed from these figures that our protocol is less affected by bandwidth variations compared to their protocols.

The communication cost for provider-side streaming increases linearly with the number of newly added elements, ranging from 2^6 to 2^{18}. In contrast, the communication costs in [KKRT16] and [RR22] are dependent on the size of the entire set, which highlights the significant improvement our protocol offers. For comparisons with BMX22 [20] and DIL+22 [22], we computed the communication costs based on the theoretical data provided in their respective papers. Fig. 6d presents a comparison of the communication costs for uu-PSI against KKRT16 [26], RR22 [27], BMX22 [20], and DIL+22 [22]. The results demonstrate that our communication cost surpasses theirs by a factor ranging from $24\times$ to $961\times$.

Client-side Streaming. The computational cost of client-side streaming depends on both the server set size and the client update set size. We define the ratio between the client set and server set as $\delta = N/n$. Fig. 7 illustrates the client update time under various bandwidths with $\delta = 2^6$. The figure reveals that, under lower bandwidth conditions, the bottleneck for client updates is the communication time required to transfer DPF keys. Comparing Fig. 6 and Fig. 7, we observe that client-side streaming achieves a $16\times$ improvement in time compared to the protocols in KKRT16 [26] and RR22 [27].

The communication cost of client-side streaming increases linearly with the size of the newly added client set, as shown in Table IV. The primary cost for client-side streaming arises from the transmission of DPF keys. Compared with DIL+22 [22], our approach demonstrates a $1.3\times$ improvement, although the communication cost of the solution proposed in BMX22 [20] is lower than ours.

TABLE IV: Client streaming's communication cost

Cardinality	2^6	2^7	2^8	2^9	2^{10}	2^{11}
Comm.	0.065	0.129	0.258	0.517	1.027	2.066

VI. Conclusion

Our proposed uu-PSI protocol represents a significant advancement in private set intersection by directly addressing the challenges of handling unbalanced data sets and accommodating frequent updates. By integrating our protocol with Apache Kafka, we provide a robust and scalable framework for managing high-volume data streams, ensuring that our solution is well-suited to the needs of modern data-intensive applications. Our experimental results demonstrate the effectiveness of the protocol in reducing both computational and communicational costs while maintaining strict privacy and security standards. In future work, we plan to further optimize the protocol's efficiency and explore its applicability to other domains that require secure and efficient data interaction.

Acknowledgement

The work was supported by the National Natural Science Foundation of China under Grants 62102050, 62325209, the Natural Science Foundation of Chongqing under Grants CSTB2022NSCQMSX0582, and the Fundamental Research Funds for the Central Universities under Grants. 2042023KF0203, 2042024kf1013.

References

[1] Y. Li, Z. L. Jiang, L. Yao, X. Wang, S.-M. Yiu, and Z. Huang, "Outsourced privacy-preserving c4. 5 decision tree algorithm over horizontally and vertically partitioned dataset among multiple parties," *Cluster Computing*, pp. 1581–1593, 2019.

[2] K. Nomura, Y. Shiraishi, M. Mohri, and M. Morii, "Secure association rule mining on vertically partitioned data using private-set intersection," *IEEE Access*, pp. 144 458–144 467, 2020.

[3] M. Ion, B. Kreuter, E. Nergiz, S. Patel, S. Saxena, K. Seth, D. Shanahan, and M. Yung, "Private intersection-sum protocol with applications to attributing aggregate ad conversions," *Cryptology ePrint Archive*, 2017.

[4] B. Pinkas, T. Schneider, and M. Zohner, "Scalable private set intersection based on ot extension," *ACM Transactions on Privacy and Security (TOPS)*, pp. 1–35, 2018.

[5] M. Ion, B. Kreuter, A. E. Nergiz, S. Patel, S. Saxena, K. Seth, M. Raykova, D. Shanahan, and M. Yung, "On deploying secure computing: Private intersection-sum-with-cardinality," in *In proceeding of IEEE European Symposium on Security and Privacy (EuroSP)*, 2020, pp. 370–389.

[6] S. Lv, J. Ye, S. Yin, X. Cheng, C. Feng, X. Liu, R. Li, Z. Li, Z. Liu, and L. Zhou, "Unbalanced private set intersection cardinality protocol with low communication cost," *Future Generation Computer Systems (FGCS)*, pp. 1054–1061, 2020.

[7] A. Miyaji, K. Nakasho, and S. Nishida, "Privacy-preserving integration of medical data: a practical multiparty private set intersection," *Journal of medical systems (JMSY)*, pp. 1–10, 2017.

[8] O. Ruan, Z. Wang, J. Mi, and M. Zhang, "New approach to set representation and practical private set-intersection protocols," *IEEE Access*, vol. 7, pp. 64 897–64 906, 2019.

[9] H. Chen, K. Laine, and P. Rindal, "Fast private set intersection from homomorphic encryption," in *In proceeding of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017, pp. 1243–1255.

[10] H. Chen, Z. Huang, K. Laine, and P. Rindal, "Labeled psi from fully homomorphic encryption with malicious security," in *In proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018, pp. 1223–1237.

[11] K. Cong, R. C. Moreno, M. B. da Gama, W. Dai, I. Iliashenko, K. Laine, and M. Rosenberg, "Labeled psi from homomorphic encryption with reduced computation and communication," in *In proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021, pp. 1135–1150.

[12] M. Wu and T. H. Yuen, "Efficient unbalanced private set intersection cardinality and user-friendly privacy-preserving contact tracing," in *USENIX Security Symposium (USENIX Security)*, 2023, pp. 283–300.

[13] Y. Son and J. Jeong, "Psi with computation or circuit-psi for unbalanced sets from homomorphic encryption," in *In proceeding of ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, 2023, pp. 342–356.

[14] D. Kales, C. Rechberger, T. Schneider, M. Senker, and C. Weinert, "Mobile private contact discovery at scale," in *In proceeding of USENIX Security Symposium (USENIX Security)*, 2019, pp. 1447–1464.

[15] S. Ghosh and M. Simkin, "The communication complexity of threshold private set intersection," in *In proceeding of Annual International Cryptology Conference (CRYPTO)*, 2019, pp. 3–29.

[16] Z. Gheid, Y. Challal, and L. Chen, "Private and efficient set intersection protocol for rfid-based food adequacy check," in *In proceeding of IEEE Wireless Communications and Networking Conference (WCNC)*, 2018, pp. 1–6.

[17] X. A. Wang, F. Xhafa, X. Luo, S. Zhang, and Y. Ding, "A privacy-preserving fuzzy interest matching protocol for friends finding in social networks," *Soft Computing*, pp. 2517–2526, 2018.

[18] A. Bhowmick, D. Boneh, S. Myers, K. Talwar, and K. Tarbe, "The apple psi system," *Apple, Inc., Tech. Rep*, 2021.

[19] X. Yang, Y. Zhao, S. Zhou, and L. Wang, "A lightweight delegated private set intersection cardinality protocol," *Computer Standards & Interfaces (CSAI)*, p. 103760, 2024.

[20] S. Badrinarayanan, P. Miao, and T. Xie, "Updatable private set intersection," *Proceedings on Privacy Enhancing Technologies (PETS)*, vol. 2022, 2022.

[21] A. C. Davi Resende and D. de Freitas Aranha, "Faster unbalanced private set intersection in the semi-honest setting," *Journal of Cryptographic Engineering*, pp. 21–38, 2021.

[22] S. Dittmer, Y. Ishai, S. Lu, R. Ostrovsky, M. Elsabagh, N. Kiourtis, B. Schulte, and A. Stavrou, "Streaming and unbalanced psi from function secret sharing," in *In proceeding of International Conference on Security and Cryptography for Networks (SCN)*, 2022, pp. 564–587.

[23] Á. Kiss, J. Liu, T. Schneider, N. Asokan, and B. Pinkas, "Private set intersection for unequal set sizes with mobile applications," *Proceedings on Privacy Enhancing Technologies (PETS)*, 2017.

[24] L. Li, B. Pal, J. Ali, N. Sullivan, R. Chatterjee, and T. Ristenpart, "Protocols for checking compromised credentials," in *In proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019, pp. 1387–1403.

[25] "Apache kafka," https://kafka.apache.org/, 2024.

[26] V. Kolesnikov, R. Kumaresan, M. Rosulek, and N. Trieu, "Efficient batched oblivious prf with applications to private set intersection," in *In proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016, p. 818–829.

[27] S. Raghuraman and P. Rindal, "Blazing fast psi from improved okvs and subfield vole," in *In proceeding of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022, pp. 2505–2517.

[28] R. Pagh and F. F. Rodler, "Cuckoo hashing," in *In proceeding of European Symposium on Algorithms (ESA)*, 2001, pp. 121–133.

[29] A. Kirsch, M. Mitzenmacher, and U. Wieder, "More robust hashing: Cuckoo hashing with a stash," *SIAM Journal on Computing (SICOMP)*, pp. 1543–1561, 2010.

[30] M. J. Freedman, Y. Ishai, B. Pinkas, and O. Reingold, "Keyword search and oblivious pseudorandom functions," in *In proceeding of Theory of Cryptography Conference (TCC)*, 2005, pp. 303–324.

[31] D. Kaviani, S. Tan, P. G. Kannan, and R. A. Popa, "Flock: A framework for deploying on-demand distributed trust," *Cryptology ePrint Archive*, 2024.

[32] D. Demmler, P. Rindal, M. Rosulek, and N. Trieu, "Pir-psi: scaling private contact discovery," *In proceeding of Privacy Enhancing Technologies (PETS)*, 2018.

[33] Y. Chen, A. Wu, Y. Yang, X. Xin, and C. Song, "Efficient verifiable cloud-assisted psi cardinality for privacy-preserving contact tracing," *IEEE Transactions on Cloud Computing (TCC)*, 2024.

[34] J. Aas and T. Geoghegan, "Introducing isrg prio services for privacy-respecting metrics," https://blog.mozilla.org/security/2019/06/06/next-steps-in-privacy-preserving-telemetry-with-prio, Nov. 2020, accessed: 2020-11.

[35] S. Englehardt, "Next steps in privacy-preserving telemetry with prio," https://www.abetterinternet.org/post/introducing-prio-services/, June 2019, accessed: 2019-06.

[36] Google, "Addendum to the analytics in exposure notifications express: Faq," https://github.com/google/exposure-notifications-android/blob/master/doc/enexpress-analytics-faq-addendum.md, Oct. 2021, accessed: 2021-10.

[37] S. Casacuberta, J. Hesse, and A. Lehmann, "Sok: oblivious pseudorandom functions," in *In proceeding of IEEE European Symposium on Security and Privacy (EuroSP)*, 2022, pp. 625–646.

[38] The Legion of the Bouncy Castle Inc., "Bouncy castle cryptography library," https://www.bouncycastle.org/, 2024, accessed: 2024-08-30.

[39] D. Kales and team, "dpf-cpp: Distributed point function (dpf) in c++," https://github.com/dkales/dpf-cpp, 2024, accessed: 2024-08-30.

[40] Cryptography research at Oregon State University, "libpsi," https://github.com/osu-crypto/libPSI.

[41] Visa-Research, "volepsi," https://github.com/Visa-Research/volepsi/.