This article has been accepted for publication in IEEE Transactions on Information Forensics and Security. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2024.3402355



# Enabling Threshold Functionality for Private Set Intersection Protocols in Cloud Computing

Jingwei Hu, Yongjun Zhao, Benjamin Hong Meng Tan, Khin Mi Mi Aung, Huaxiong Wang

***Abstract*—Multi-party computation (MPC) allows parties to interact with cloud-based data and services while maintaining privacy and confidentiality of their private data. As a special case of MPC, private set intersection (PSI) protocols focus on securely computing the intersection between a server and a client of their private set. Our research extends the threshold functionality for PSI within the realm of cloud computing, where the server possesses a larger set than the client. This paper fills this gap by proposing new private intersection cardinality (PSI-CA) protocol, and more broadly, threshold private set intersection (tPSI) protocol using fully homomorphic encryption (FHE). In tPSI protocol, two parties holding two private sets collaboratively compute the intersection and reveal the result if and only if the size of the intersection exceeds some predefined threshold. In this process, no other information, in particular, elements not in the intersection remain hidden. The problem of PSI-CA and tPSI has many applications in online collaboration, *e.g.*, fingerprint matching, online dating, and ride sharing.**

**At a high level, we use FHE to encrypt a Bloom filter (BF) that encodes the small set and homomorphically check whether the elements in the larger set belongs to the small set, *e.g.*, homomorphic membership test. Counting the number of positive membership directly already yields a PSI-CA protocol with optimal asymptotic communication complexity $\Omega(n) = \Omega(\min(N, n))$, where $N$ (resp. $n$) is the size of the large (resp. small) set. To construct a tPSI protocol, we develop a novel secret token generation protocol: a shared secret token is generated if and only if the intersection size satisfies the threshold condition, by exploiting the programmable bootstrapping technique in FHE. This new secret token generation protocol, when composed with any standard PSI protocol, yields a tPSI with the same asymptotic communication complexity as the chosen plain PSI.**

**Along the way, we develop specific FHE optimizations that might be of independent interest. These optimizations overcome the weakness of low precision in programmable bootstrapping. As a result, tPSI over relatively large sets can be supported.**

***Index Terms*—Private Set Intersection, Threshold PSI, Fully Homomorphic Encryption, TFHE**

## I. Introduction

Private set intersection (PSI) [1]–[4] allows two parties, each holding a private set of data, to compute the intersection of their sets without revealing elements not in the intersection. PSI has a wide range of applications, including:

1) Healthcare: Different hospitals or clinics can compare patient data without revealing any sensitive information about the non-intersecting patients. It can also be used to identify patients who have participated in multiple studies while protecting the privacy of irrelevant patients.
2) E-commerce: Online marketplaces can use PSI to identify common products between buyers and sellers while keeping the rest of their inventories private. This can facilitate transactions without revealing sensitive business information.
3) Finance: Two banks can detect fraud and money laundering by using PSI to identify common customers who have accounts with both institutions without revealing any other customer data.
4) Social media: PSI can be used to identify common interests between users of social media platforms while preserving their privacy. This can help personalize recommendations and improve user engagement.
5) Cybersecurity: PSI can be used to detect data breaches and identify stolen information. For example, an organization can use PSI to compare its internal database with a list of leaked credentials without revealing the contents of either dataset.

While PSI has wide applications, in many scenarios, the focus is not solely on computing the intersection, but further computing specific functions over the intersection. Considering the fingerprint matching application, where one party, the client, possesses a small set of fingerprint features, while the other party, the server, maintains a large set of fingerprint features for multiple users. In this case, the matching algorithm returns positive if the number of matched features exceeds a predetermined threshold (to account for false positive); otherwise, it returns negative. Another example is matchmaking on a dating website, in which users have a set of preferences. The matching protocol should return the shared preferences if the number of common preferences of two users is sufficiently large, allowing the two users to gain more insight of each other. If the number of shared preferences falls below the threshold, the protocol should return an empty set (or nothing) to prevent the disclosure of personal privacy. These types of applications are commonly known as threshold PSI [5]–[7], where each party holds a set of size $n$, and the computation of the set intersection occurs only when the number of overlapping elements exceeds a specified threshold value $t$.

Our primary focus in this work is to explore the feasibility of employing third-generation FHE programmable bootstrapping [8], [9] to construct a PSI-with-computations protocol. It's crucial to note that alternative methods [10]–[17], such as those relying on garbled circuits, GMW protocol, or a combination of FHE and generic-MPC, exist for achieving PSI-with-

Jingwei Hu is with the Digital Trust Centre, Nanyang Technological University, Singapore, Email: davidhu@ntu.edu.sg. Yongjun Zhao is with TikTok Inc., Email: zhaoyongjun.280191@tiktok.com. Khin Mi Mi Aung and Benjamin Hong Meng Tan are with the Institute for Infocomm Research, A*STAR, Singapore, Email: {mi_mi_aung, benjamin_tan}@i2r.a-star.edu.sg. Huaxiong Wang is with the School of Physical and Mathematical Sciences, Nanyang Technological University, Singapore, Email: hxwang@ntu.edu.sg.

computations. In comparison, our approach achieves optimal round complexity, and our protocols maintain conceptual simplicity and modularity. This simplicity streamlines both security analysis and implementations. While we acknowledge that the concrete computational and communication complexity of our solution may not be entirely satisfactory for relatively large set sizes, we consider this work as an initial step towards a practical FHE-based PSI-with-computations protocol. We believe that ongoing advancements in FHE technologies will contribute to overcoming existing limitations and ultimately enhance the feasibility of our proposed approach.

### A. Our Contributions

In this study, we explore the potential of utilizing state-of-the-art fully homomorphic encryption schemes to address the threshold PSI problem. Our contributions are as follows:

1) We propose a private set intersection cardinality protocol designed for two parties in the presence of semi-honest adversaries. The communication complexity of our proposed method achieves the lower bound of $\Omega(\min(N, n))$, where $N$ and $n$ are the sizes of the two private sets respectively.
2) We develop a novel secret token generation protocol building upon our private set intersection cardinality protocol. By composing this new protocol with a standard PSI protocol, we obtain a threshold PSI protocol for two parties in the semi-honest model. This protocol exhibits a communication complexity that scales sublinearly with the size of the larger set, making it well-suited for efficient cloud computing applications.
3) We introduce a range of FHE-related optimizations to reduce the computational and communication overhead and achieve efficient operations on large sets.
4) Our protocols are designed to be conceptually simple and modular to facilitate easy security analysis and fast implementation.

These contributions collectively advance the field of threshold PSI by providing efficient and secure protocols.

## II. Related Work

PSI has been extensively studied in the literature and remains an active research area. Apart from standard PSI, the research community has also investigated PSI variants, such as private intersection cardinality, threshold private set intersection and circuit private set intersection.

**Private Intersection Cardinality:** The private intersection cardinality problem [18]–[22] focuses on the computation of the cardinality of the intersected set, rather than the intersection itself. Freedman *et al.* [1] establishes a lower bound on the communication complexity of private set intersection, which naturally applies to the private intersection cardinality problem as well. The lower bound states that the communication complexity is at least $\Omega(n)$, where $n$ is the size of the set held by each party.

**Threshold Private Set Intersection:** The threshold PSI problem has been investigated in several studies [1], [5]–[7], [23]. Hallgren *et al.* [6] combines the phasing technique with the threshold key encapsulation mechanism to construct a threshold PSI protocol. Ghosh and Nilges [23] proposes a malicious-secure threshold PSI based on oblivious linear function evaluation and robust secret sharing. Zhao and Chow [5] construct semi-honest secure (over and below) threshold PSI protocols using oblivious polynomial evaluation. Ghosh and Simkin [24] reveals that when the intersection size is large (*i.e.*, the size of the intersection is at least $n - 2t \approx n$), a threshold PSI protocol must have a communication complexity of $\Omega(t)$. Bay *et al.* [25] study a new variant of threshold multi-party PSI which outputs items of the server that appear in at least $k(k < N)$ other private sets. Wang *et al.* [26] introduce an enhanced notion of outsourced PSI, called authorized PSI (APSI), which supports flexible authorization and cross-type authorized comparison of datasets.

**Circuit Private Set Intersection:** Differing from the standard Private Set Intersection (PSI), Circuit Private Set Intersection (Circuit-PSI) provides each party with a vector form of shares (secret shares). These shares can collectively reconstruct a complete 0/1 vector, where '1' signifies an element in the intersection, and '0' indicates an element outside the intersection. In the initial work on Circuit-PSI [10], a generic garbled circuit was considered for constructing PSI, with computational and communication complexity of $\Omega(N \log N)$. [11] employed Cuckoo-Hashing and Simple-Hashing for bin-wise comparison, slightly reducing the computational and communication complexity to $\Omega(N \log N / \log \log N)$. [12] introduced the concept of Oblivious Batched Programmable PRF (Batched-OPPRF) and, based on this, first constructed Circuit-PSI with linear communication complexity of $\Omega(N)$. [13] further improves on [12] to construct a circuit-PSI with both linear communication and computation complexity. [14], besides introducing VOLE for constructing OPRF and ultimately standard PSI, explored the transformation of VOLE-based OPRF into Batched-OPPRF to construct Circuit-PSI. [15] discussed the Circuit-PSI problem for asymmetric set sizes for the first time, introducing PIR techniques for resolution. [16] similarly explored Circuit-PSI with asymmetric set sizes, leveraging Fully Homomorphic Encryption (FHE) and MPC techniques to construct a more efficient Circuit-PSI.

## III. Preliminary

### A. TFHE/FHEW

TFHE [8] and FHEW [9] are examples of the third generation of fully homomorphic encryption (FHE) schemes. They offer advantages in terms of bootstrapping performance, which is a crucial operation in FHE schemes. The state-of-the-art advancements in TFHE/FHEW indicate that any discrete function $f$ over a small domain $\mathbb{Z}_t$ can be bootstrapped. This unique capability is often referred to as functional or programmable bootstrapping. It allows for performing computations on encrypted data without decrypting it.

*1) LWE and RLWE:* TFHE ciphertext is essentially an LWE ciphertext. The LWE ciphertext is defined as follows:

$$(\mathbf{a}, b) = (\mathbf{a}, \langle \mathbf{a}, \mathbf{s} \rangle + \frac{q}{t} m + e) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$$

, where $\mathbf{s}$ is a secret key, $m \in \mathbb{Z}_t$ is the message, $e$ is an error term extracted from a discrete Gaussian distribution with small variance $e \sim \mathcal{N}(0, \sigma^2)$. The parameters $n, q, \mathbf{s}$ characterize an LWE ciphertext. We use the notation $LWE_{\mathbf{s}}^{n,q}(m)$ (or $LWE_{\mathbf{s}}(m), LWE(m)$ when the context is clear) to represent an LWE encryption of the message $m$.

In the TFHE bootstrapping algorithm, another format of ciphertext, called RLWE ciphertext, is used. An RLWE ciphertext can be viewed as an LWE ciphertext defined over polynomial ring $R_Q = \mathbb{Z}_Q[X]/(X^N + 1)$:

$$(a(X), b(X)) = (a(X), a(X) \cdot z(X) + \frac{Q}{t} m(X) + e(X)) \in R_Q \times R_Q,$$

where $z$ is the secret key, $m$ is the message, $e = \sum_i e_i X^i$ is the error polynomial where each coefficient is extracted from a discrete Gaussian distribution with small variance $e_i \sim \mathcal{N}(0, \sigma^2)$. Likewise, we use $RLWE_z^{N,Q}(m)$ to denote an RLWE encryption of the message $m(X)$.

*2) Key Switch and Mod Switch:* Key switching and modulus switching are two important homomorphic operations in TFHE. We skip the algorithmic detail for these two primitives but outline the functionality abstractly. `KeySwitch` takes as input an LWE encryption of message $m$ under secret key $\mathbf{z}$ and outputs another LWE ciphertext encrypting the same message $m$ but under a different secret key $\mathbf{s}$:

$$LWE_{\mathbf{z}}^{N,Q}(m) \xrightarrow{\texttt{KeySwitch}} LWE_{\mathbf{s}}^{n,Q}(m)$$

`KeySwitch` utilizes another publicly known key called key switching key $\mathbf{KSK}$ to perform this operation and `KeySwitch` introduces extra noise to the input LWE ciphertext. In practice, we may use another form of key switching, called LWE-to-RLWE keyswitch which homomorphically converts an LWE encryption to an RLWE encryption without changing the encrypted message:

$$LWE_{\mathbf{z}}^{N,Q}(m) \xrightarrow{\texttt{LWE-to-RLWE KeySwitch}} RLWE_z^{n,Q}(m)$$

`ModSwitch` takes input as an LWE encryption of message $m$ defined over $\mathbb{Z}_Q^N \times \mathbb{Z}_Q$ and outputs another LWE encryption of the same message defined over $\mathbb{Z}_q^N \times \mathbb{Z}_q$ with $Q > q$:

$$LWE_{\mathbf{s}}^{N,Q}(m) \xrightarrow{\texttt{ModSwitch}} LWE_{\mathbf{s}}^{N,q}(m)$$

A unique feature of `ModSwitch` is that it reduces the noise within the LWE ciphertext. TFHE bootstrapping utilizes this feature to homomorphically evaluate a discrete function $f$ while reducing the noise in the ciphertext.

*3) Sample Extraction:* Sample extraction is another classical technique that homomorphically extracts a term in an encrypted polynomial. More precisely, the input for sample extraction is an RLWE ciphertext $RLWE(m(X))$ which encrypts a polynomial $m(X) = \sum_{i=0}^{N-1} m_i X^i$ and sample extraction extracts the $i$-th term of $m(X)$ as an LWE encryption $LWE(m_i)$ denoted as:

$$LWE_{\mathbf{z}}^{N,Q}(m_i) \leftarrow \texttt{SampleExt}(RLWE_z^{N,Q}(m(X), i)$$

*4) External Product:* In TFHE/FHEW schemes, a new encryption scheme called RGSW [27] is used as the basis for homomorphic multiplication. We use the notation $RGSW_z^{N,Q}(m(X))$ to denote an RGSW encryption of a polynomial $m(X) \in R_Q = \mathbb{Z}_Q[X]/(X^N + 1)$ under the secret key $z$. A specific form of homomorphic multiplication is widely used in TFHE-style fully homomorphic encryptions as follows:

$$RLWE_z^{N,Q}(m_0(X)m_1(X)) \leftarrow RLWE_z^{N,Q}(m_0(X)) \odot RGSW_z^{N,Q}(m_1(X))$$

*5) Half domain bootstrapping from FHEW/TFHE-like FHEs:* Half-domain bootstrapping (HDB) is invented in the classic FHEW/TFHE schemes. It is called 'half-domain' as the function $f : \mathbb{Z}_t \to \mathbb{Z}_t$ it evaluates only covers half of the function domain, *i.e.*, $[0, \cdots, \frac{t}{2} - 1]$.

To construct HDB, the key point is to prepare a rotation polynomial $rotP(X)$ for the given function $f$ such that $Const(rotP \cdot X^{\frac{q}{t}m+e}) = f(m)$ where $Const(P(X))$ denotes extracting the constant term of the polynomial $P(X)$.

A full description of HDB is outlined in Alg. 1. In Step 1, the rotation polynomial $rotP(X)$ is configured based on the bootstrap function $f$. In Step 2, the ciphertext modulus is downscaled from $q$ to $2N$ while preserving the encrypted message $m$. Steps 3-4 orchestrate the key operation blind rotation, yielding $acc_{\text{BR},f} = RLWE_z^{N,Q}(rotP \cdot X^{\widetilde{m}})$ where $\widetilde{m} = b_N - \langle \mathbf{a}N, \mathbf{s} \rangle = \frac{q}{t}m + e$. Step 5 homomorphically extracts the constant term of $rotP \cdot X^{\widetilde{m}}$, resulting in $ct_Q = LWE\mathbf{z}^{N,Q}(f(m))$. Finally, Steps 6-7 produce the output $ct_q = LWE_{\mathbf{s}}^{n,q}(f(m))$. HDB is relatively fast since it performs the most time-consuming operation `BlindRotate`, whose computational complexity is $\mathcal{O}(d_g N^2 \log N)$, only once. Here, $d_g$ is a small constant used in the TFHE system parameter description. For 80-bit security, one HDB function evaluation takes less than 1 second by a typical software implementation on a desktop PC.

**Input:** bootstrapping key brK, LWE ciphertext ct=$LWE_{\mathbf{s}}^{n,q}(m) = (\mathbf{a}, b)$, LWE-to-LWE key-switching key ksK.
**Output:** an LWE encryption of $f(m)$, *i.e.,* either $ct_q = LWE_{\mathbf{s}}^{n,q}(f(m))$ or $ct_Q = LWE_{\mathbf{z}}^{N,Q}(f(m))$.

1 Set $rotP = f(0)\left(\sum_{j=0}^{\frac{N}{t}} X^j + \sum_{j=N-\frac{N}{t}+1}^{N-1} X^j\right) - \sum_{i=1}^{\frac{t}{2}-1} f(i)\left(\sum_{j=N-\frac{N}{t}-\frac{2Ni}{t}+1}^{N+\frac{N}{t}-\frac{2Ni}{t}} X^j\right)$
2 Set $ct_N \leftarrow \texttt{ModSwitch}(ct, q, 2N) = (\mathbf{a}_N, b_N)$
3 Compute $acc_f \leftarrow rotP \cdot X^{b_N}$
4 Run $acc_{\text{BR},f} \leftarrow \texttt{BlindRotate}(brK, acc_f, ct_N)$
5 Run $ct_Q \leftarrow \texttt{SampleExt}(acc_{\text{BR},f}, 0)$
6 Run $ct_{Q,ksK} \leftarrow \texttt{KeySwitch}(ct_Q, ksK)$
7 Run $ct_q \leftarrow \texttt{ModSwitch}(ct_Q, Q, q)$
8 **return** $ct_q$

**Algorithm 1:** Half domain bootstrapping `Bootstrap`(brK, ct, $f(\cdot)$, ksK)

*6) Full domain bootstrapping from FHEW/TFHE-like FHEs:* Another bootstrapping type that improves the domain of the evaluation function $f$ is called full-domain bootstrapping (FDB). FDB essentially provides a way to evaluate a discrete function over the entire domain $\mathbb{Z}_t$ instead of the half domain $\mathbb{Z}_{t/2}$. In other words, FDB improves HDB on the precision of the function it can homomorphically evaluate by 1 bit. There exist several different methods for realizing FDB which include SelecMSB [28]–[30], HalfRange [31] and SPlit [32]. In our PSI implementation, we choose the well-documented and open-sourced codes released by [29] for realizing FDB. FDB is slower than HDB because it performs the most time-consuming task `BlindRotate` for $\ell_{boot}+1 \approx 9$ times. The computational complexity of full-domain bootstrapping is $\mathcal{O}(\ell_{boot} d_g N^2 \log N)$ where $\ell_{boot}$ and $d_g$ are small constants used in the TFHE system parameter description. For 80-bit security, one FDB function evaluation takes about 12 seconds a typical desktop PC reported by [29].

### B. Bloom Filter

Bloom filter [33] is a data structure designed to efficiently determine if an element belongs to a set, but they do not provide a way to retrieve the actual object. It stores membership information in a compact manner.

A Bloom filter is represented as a bit array, denoted as $\mathbf{BF}[\cdot]$.Bloom filter supports two elementary operations: `AppendObject` which adds the target object to the Bloom filter array and `TestMembership` which checks whether a target object has been logged in the Bloom filter.

The false positive rate ($p_{\text{FPR}}$) is the most important performance measure for Bloom filters. It represents the probability that the membership test operation incorrectly asserts that the target object is in the Bloom filter.

In our C++ implementation of the Bloom filter, we use the MD5 hash function. To fulfill the requirement of having a number of $|\mathbf{Hash}|$ distinct hash functions, we pre-generate $|\mathbf{Hash}|$ random numbers and use them to seed a single instance of the MD5 hash algorithm with different randomness. In our implementation, we set the false positive rate to $2^{-20}$ and the number of hash functions to 20, resulting in a Bloom filter size of $|\mathbf{BF}| \approx 28.9 \times |S|$.

## IV. The Framework for Threshold PSI via FHE

In this section, we describe our threshold PSI at an abstract level, which mostly resembles the design blueprint in [5]. The threshold can be one of the three types: Below-threshold (which returns the set intersection only if the intersection cardinality is below some value), Above-threshold (which returns the set intersection only if the intersection cardinality is above some value), and In-between-threshold (which returns the set intersection only if the intersection cardinality falls within some interval). As shown in Fig. 1, the client holds a set $C = \{c_i\}_{i\in[|C|]}$ and the server holds a set $S = \{s_i\}_{i\in[|S|]}$. Our threshold PSI is divided into two subprotocols: the threshold token generation protocol, and the standard PSI protocol. In the threshold token generation protocol, an encryption of a secret token $K'$ and another secret token $K$ are generated from the server side based on the encrypted intersection cardinality, *i.e.*, $Enc(|C\cap S|)$. The client receives and decrypts the encryption of $K'$ and appends $K'$ to every item in his set $C$ such that $C$ is updated to $C' = \{c_i||K'\}_i$. Meanwhile, the server also updates his set from $S$ to $S' = \{s_i||K\}_i$. The client inputs his new set $C'$ and the server inputs his new set $S'$ in the standard PSI protocol, *e.g.*, Diffie-Hellman-based PSI is used in Fig 1. At the end, the client learns nothing more than $C \cap S$ if the threshold criterion is met. Otherwise, the client receives (with an overwhelming probability) an empty set.

More specifically, in the threshold token generation protocol, the client first encodes his set $C$ into a Bloom filter $\mathbf{BF_c}$ and then uses an FHE scheme to encrypt $\mathbf{BF_c}$ as $Enc(\mathbf{BF_c})$. When the server receives $Enc(\mathbf{BF_c})$ sent from the client, he first performs a new primitive called `ePSI-CA` to homomorphically obtain an encryption of the intersection cardinality $|C\cap S|$ as $Enc(|C\cap S|)$. The server then performs another primitive called `SecretTokenGen` to homomorphically generate an encryption of token $K'$ based on a public known threshold criterion, an encryption of $|C \cap S|$ and a self-generated secret token $K$. The relation between $K$ and $K'$ is that $K' = K$ if the intersection cardinality satisfies the threshold criterion; otherwise, $K'$ is completely random. The server uses $K$ to update his set to $S' = \{s_i||K\}_i$, and also sends $Enc(K')$ back to the client. Finally, the client decrypts $Enc(K')$ for updating his set to $C' = \{c_i||K\}_i$. The details of `ePSI-CA` and `SecretTokenGen` will be discussed later.

**Security analysis** We present the following theorem asserting our new threshold PSI protocol is secure against semi-honest adversaries:

**Theorem IV.1.** *Assuming the existence of a CPA-secure fully homomorphic encryption scheme with circuit privacy and semi-honest secure standard PSI protocol; then the protocol in Fig.1 is secure in the presence of semi-honest adversaries.*

## V. Challenges in the FHE-based threshold PSI

The FHE scheme used in this work is TFHE, which is known for its fast bootstrapping capability. TFHE's bootstrapping extends beyond the evaluation of just the identity function and can be applied to any discrete function, making it programmable. However, TFHE has certain drawbacks in terms of computational efficiency, which can be attributed to two main factors.

First, TFHE encrypts messages in LWE ciphertext with a small modulus $q$ denoted as $LWE^{n,q}(\cdot)$. The use of a small modulus limits the capacity for noise growth during homomorphic computations. Even for simple operations like addition, bootstrapping is required after each addition, significantly increasing the computation time per operation.

Second, the precision of programmable bootstrapping in TFHE is limited. TFHE supports the evaluation of discrete functions $f(t) : \mathbb{Z}_t \leftarrow \mathbb{Z}_t$ with small $t$ values ranging from $2^6$ to $2^8$. This means that the precision of the bootstrapped function is only in the range of 6 to 8 bits. However, in the threshold PSI protocol, we often need to perform bootstrapping on intersection cardinalities $|C\cap S|$ that can be relatively large and go beyond the precision supported by TFHE.

This article has been accepted for publication in IEEE Transactions on Information Forensics and Security. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2024.3402355



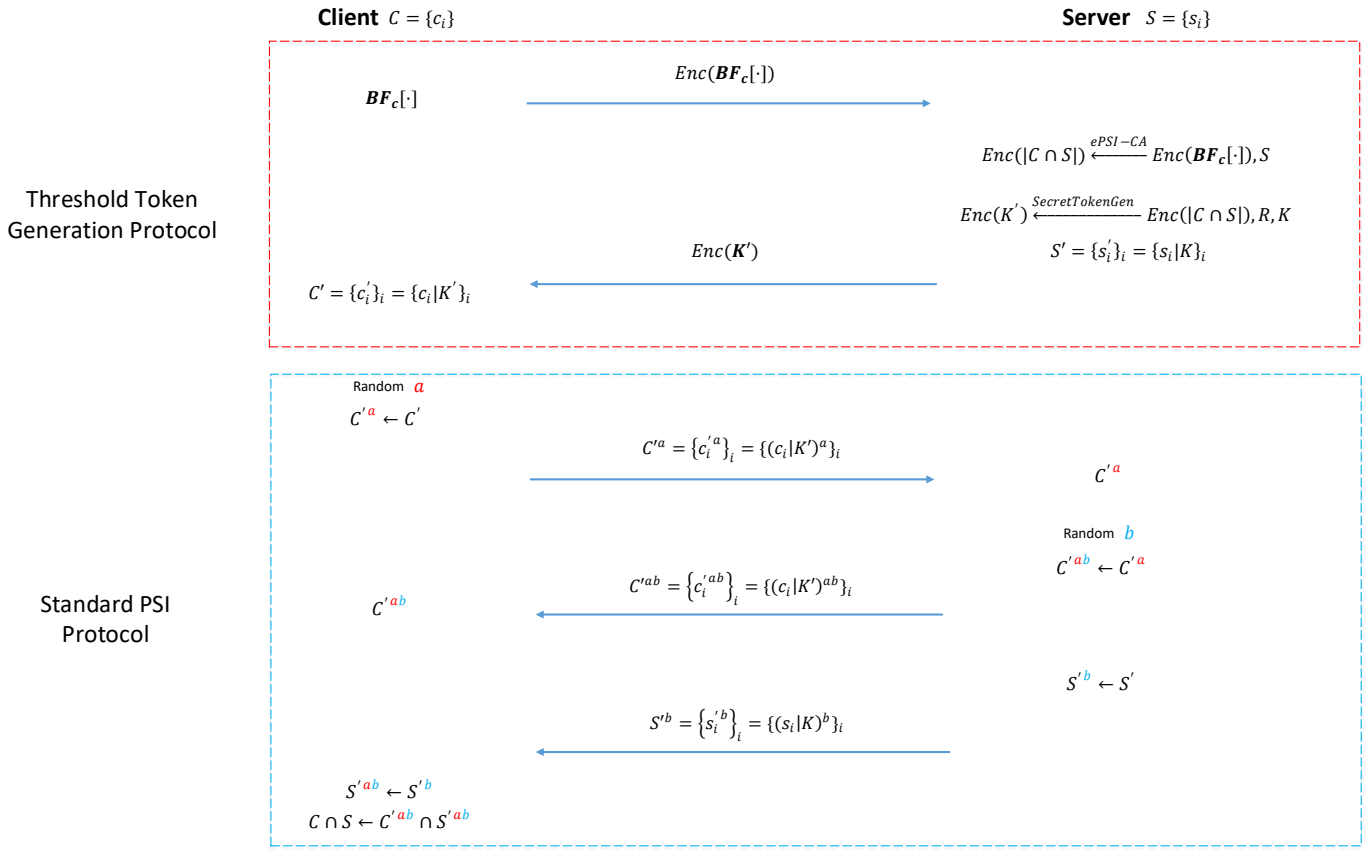


Fig. 1: Design blueprint of our threshold PSI protocol

TABLE I: Notation symbols used in this paper

| Notations | Definition |
|---|---|
| $C$ | client set |
| $c \in C$ | an element in the client set |
| $S$ | server set |
| $s \in S$ | an element in the server set |
| $\|S\|$ | the cardinality of the set $S$ |
| $\mathbf{index} \leftarrow$ compute_indices$(s)$ | compute the Bloom filter indices (returned as an array) related to a set element $s$ |
| $\mathbf{BF}_C[\cdot]$ | Bloom filter array for the set $C$ where $BF_C[idx]$ with $idx \in$ compute_indices(c) for all $c \in C$ is set to 1, otherwise 0. |
| $\delta$ | threshold pulse function, *i.e.*, returns 1 if the input value is above(below) a threshold, otherwise returns 0 |
| $\mathbf{z}, \mathbf{z}_i$ | vector $\mathbf{z}$ and the $i$-th element of the vector $\mathbf{z}$ |
| $LWE_{\mathbf{s}}^{n,q}(m) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$ | an LWE encryption of the message $m \in \mathbb{Z}_q$ w.r.t. secret key $\mathbf{s}$ and lattice dimension $n$ |
| $LWE_{\mathbf{z}}^{N,Q}(m) \in \mathbb{Z}_Q^N \times \mathbb{Z}_Q$ | an LWE encryption of the message $m \in \mathbb{Z}_Q$ w.r.t. secret key $\mathbf{z}$ and lattice dimension $N$ |
| $Err(LWE_{\mathbf{s}}^{n,q}(m))$ | the noise component in the LWE ciphertext $LWE_{\mathbf{s}}^{n,q}(m)$ |
| $x = \|x_{d-1} \cdots x_i \cdots x_0\|_2$ | a $d$-bit integer $x$ with the $i$-th bit represented as $x_i$ |

In this paper, a new approach is proposed to address the limitations of TFHE regarding computational efficiency and precision. The proposed approach involves using bootstrapping in the "BeforeKeySwitch" mode, which enables TFHE to operate directly on LWE ciphertext with a larger modulus denoted as $LWE^{N,Q}(\cdot)$. This new LWE ciphertext format allows for simple homomorphic additions without the need for bootstrapping, resulting in significant improvements in computation efficiency, particularly in the threshold token generation subprotocol.

To handle cases where the intersection set exceeds the precision of TFHE's bootstrapping function, the paper introduces high-precision homomorphic integer addition and comparison operations. These operations enable the protocol to handle larger intersection sets without sacrificing accuracy or incurring excessive computational overhead.

Furthermore, the paper proposes revisions to the threshold PSI protocol and utilizes advanced TFHE homomorphic operations to improve the communication complexity. These modifications aim to optimize the protocol and reduce the amount of communication required between the client and server, leading to more efficient execution.

### A. Let TFHE work at leveled-FHE mode

The first crucial task in the proposed PSI protocol is to implement the primitive `ePSI-CA`. This primitive allows the

server to determine the cardinality of the intersection $C \cap S$ using the encryption of the client's Bloom filter. For simplicity, let's assume that the client's Bloom filter array $\mathbf{BF_c}[\cdot]$ is encrypted bit by bit as $LWE_{\mathbf{s}}^{n,q}(\mathbf{BF_c}[i])$ for all $i$. To compute the intersection cardinality, the server needs to check whether each element $s_i$ in its set $S$ belongs to $\mathbf{BF_c}[\cdot]$. This check is commonly known as the membership test. Formally, the membership test can be described by the following function:

$$\delta = f(\sum_j \mathbf{BF_c}[index_j]) = \begin{cases} 1 & \text{if } \sum_j \mathbf{BF_c}[index_j] \geq |\mathbf{Hash}| \\ 0 & \text{Otherwise} \end{cases}$$

Here, $|\mathbf{Hash}|$ represents the number of hash functions used in the Bloom filter.

Note that the membership test needs to be performed homomorphically. The challenging part is the homomorphic additions of $LWE_{\mathbf{s}}^{n,q}(\mathbf{BF}_c[index_j])$ for all $index_j$. The parameter $q$ is relatively small ($\approx 2^{11}$), which allows only 2-3 times homomorphic additions before decryption failure occurs. This limitation in $q$ significantly affects the performance of $\sum_j LWE_{\mathbf{s}}^{n,q}(BF_c[index_j])$. After a few homomorphic additions, bootstrapping (homomorphic evaluation of the identity function) is required to refresh the ciphertext and reduce the noise component. Specifically, $\Omega(|\mathbf{Hash}|)$ bootstrappings are needed to obtain $LWE_{\mathbf{s}}^{n,q}(\sum_j \mathbf{BF}_c[index_j])$. The naive approach for this is described in Alg. 2.

We present a new method of homomorphic accumulation without refreshing/bootstrapping. The key idea is to leverage two fully homomorphic encryptions (FHEs), one with a smaller modulus $q$ and the other with a larger modulus $Q$ (typically around $2^{63}$), both encrypting the same message $m$. By encrypting the client's bloom filter as $LWE_{\mathbf{z}}^{N,Q}(\mathbf{BF}_c[\cdot])$, it becomes possible to tolerate a significantly higher level of noise during homomorphic additions without the need for bootstrapping.

Once the homomorphic accumulation is completed, a key-switching operation and a modulus-switching operation are performed to obtain $LWE_{\mathbf{s}}^{n,q}(\sum_j \mathbf{BF}_c[index_j])$, which is in the format required for the subsequent bootstrapping step. This new method is outlined in Alg. 3.

**Input:** Encrypted Bloom filter bit-table $\{LWE_{\mathbf{s}}^{n,q}(BF[i])\}_i$ and targeted indices $\{index_j\}_j$
**Output:** $LWE_{\mathbf{s}}^{n,q}(\sum_j BF[index_j])$

```
tmp ← 0
for every index ∈ {index_j}_j do
    tmp+ = LWE_s^{n,q}(BF[index])
    tmp ← Bootstrap(tmp, f = identity)
return tmp
```

**Algorithm 2:** Straightforward strategy for homomorphic accumulation

### *B. Let TFHE bootstrapping work in* `BeforeKeySwitch` *mode*

Switching from a small modulus $q$ to a larger modulus $Q$ enables homomorphic accumulations without the need for bootstrapping. However, it introduces a new challenge. After the server performs the homomorphic membership test and bootstrapping for the function $f(\cdot)$, it obtains $|S|$ $LWE_{\mathbf{s}}^{n,q}$ ciphertexts, each representing either a 1 (indicating the item is in set $S$) or a 0 (indicating the item is not in set $S$). The goal is to accumulate these ciphertexts to obtain the LWE encryption of $|C \cap S|$.

**Input:** Encrypted Bloom filter bit-table $\{LWE_{\mathbf{z}}^{N,Q}(BF[i])\}_i$ and targeted indices $\{index_j\}_j$
**Output:** $LWE_{\mathbf{s}}^{n,q}(\sum_j BF[index_j])$

```
tmp ← 0
for every index ∈ {index_j}_j do
    tmp+ = LWE_z^{N,Q}(BF[index])
Perform tmp ← KeySwitch(tmp)
Perform tmp ← ModSwitch(tmp)
return tmp
```

**Algorithm 3:** Our proposed fast strategy for homomorphic accumulation which improves Alg. 2

As discussed earlier, TFHE bootstrapping accepts $LWE_{\mathbf{s}}^{n,q}(m)$ as input and outputs $LWE_{\mathbf{s}}^{n,q}(f(m))$, but direct addition of $LWE_{\mathbf{s}}^{n,q}(f(m))$ is not feasible due to noise capacity restrictions.

This problem can be resolved by terminating the bootstrapping algorithm earlier. By doing so, the bootstrapping accepts $LWE_{\mathbf{s}}^{n,q}(m)$ ciphertexts over a small modulus $q$ as input and outputs $LWE_{\mathbf{z}}^{N,Q}(f(m))$ over a larger modulus $Q$. In the core of the bootstrapping algorithm, the `BlindRotate` operation manipulates the input in RLWE ciphertexts over $R_Q \times R_Q$ and subsequently uses `KeySwitch` and `ModSwitch` to transform it back to an LWE ciphertext over $\mathbb{Z}_q^n \times \mathbb{Z}$. By skipping the `KeySwitch` and `ModSwitch` steps, the functionality of computing $f(m)$ is preserved, but the result is represented in another LWE ciphertext with a larger modulus $Q$ and a different secret key $\mathbf{z}$.

This modification allows for the accumulation of ciphertexts without the need for bootstrapping, enabling the computation of $|C \cap S|$ in the desired LWE ciphertext format.

Alg. 4 describes the half-domain bootstrapping algorithm in `BeforeKeySwitch` mode. If we do not set mode, then this algorithm is exactly the standard TFHE bootstrapping. However, if we set $mode =$ `BeforeKeySwitch`, then this algorithm outputs $f(m)$ encapsulated in an $LWE_{\mathbf{z}}^{N,Q}$ ciphertext, *i.e.*, $ct_Q$.

**Corollary V.0.1.** *Alg. 4 in* `BeforeKeySwitch` *mode returns* $LWE_{\mathbf{z}}^{N,Q}(f(m))$.

With the new proposal of the `BeforeKeySwitch`-mode bootstrapping, we are ready to construct `ePSI-CA` formally, as shown in Alg. 5.

**Corollary V.0.2.** *Alg. 5 returns an encryption of intersection cardinality, i.e.,* $LWE_{\mathbf{z}}^{N,Q}(|C \cap S|)$.

**Input:** bootstrapping key brK, LWE ciphertext ct=$LWE^{n,q}_{\mathbf{s}}(m) = (\mathbf{a}, b)$, LWE-to-LWE key-switching key ksK.
**Output:** an LWE encryption of $f(m)$, *i.e.,* either $ct_q = LWE^{n,q}_{\mathbf{s}}(f(m))$ or $ct_Q = LWE^{N,Q}_{\mathbf{z}}(f(m))$.

Set $rotP = f(0)\left(\sum_{j=0}^{\frac{N}{t}} X^j + \sum_{j=N-\frac{N}{t}+1}^{N-1} X^j\right) - \sum_{i=1}^{\frac{t}{2}-1} f(i)\left(\sum_{j=N-\frac{N}{t}-\frac{2Ni}{t}+1}^{N+\frac{N}{t}-\frac{2Ni}{t}} X^j\right)$
Set $ct_N \leftarrow \texttt{ModSwitch}(ct, q, 2N) = (\mathbf{a}_N, b_N)$
Compute $acc_f \leftarrow rotP \cdot X^{b_N}$
Run $acc_{\text{BR},f} \leftarrow \texttt{BlindRotate}(brK, acc_f, ct_N)$
Run $ct_Q \leftarrow \texttt{SampleExt}(acc_{\text{BR},f}, 0)$
**if** *mode* == *BeforeKeySwitch* **then**
  **return** $ct_Q$
**else**
  Run $ct_{Q,ksK} \leftarrow \texttt{KeySwitch}(ct_Q, ksK)$
  Run $ct_q \leftarrow \texttt{ModSwitch}(ct_Q, Q, q)$
  **return** $ct_q$

**Algorithm 4:** Half domain bootstrapping `Bootstrap`(brK, ct, $f(\cdot)$, ksK)

**Input:** On the client side: A set $C$, the threshold value $t$; On the server side: A set $S$, encrypted client BF array $LWE^{N,Q}_{\mathbf{z}}(\mathbf{BF_c}[i])$ for all $i$
**Output:** The client side receives an LWE encryption of $|C \cap S|$

The server performs the following loop to obtain $LWE^{N,Q}_{\mathbf{z}}(|C \cap S|)$
**for** *every* $s_i \in S$ **do**
  $\{index_j\}_j \leftarrow compute\_indices(s_i)$
  Perform homomorphic additions to obtain $LWE^{N,Q}_{\mathbf{z}}(\sum_j \mathbf{BF}_c[index_j])$
  Perform Key-switching to obtain $LWE^{n,q}_{\mathbf{s}}(\sum_j \mathbf{BF}_c[index_j])$
  Perform bootstrapping (before-key-switch mode) to obtain $LWE^{N,Q}_{\mathbf{z}}(\delta_i)$
Perform homomorphic additions to obtain $LWE^{N,Q}_{\mathbf{z}}(\sum_i \delta_i)$
**return** $LWE^{N,Q}_{\mathbf{z}}(\sum_i \delta_i)$ *to the client*

**Algorithm 5:** Core primitive `ePSI-CA` via FHE

### C. Putting all pieces together

We formally describe the TFHE-based threshold PSI protocol in Alg. 6, and the correctness of the algorithm is stated in the following corollary.

**Corollary V.0.3.** *Alg. 6 allows the client and the server to compute the set intersection if the intersection cardinality satisfies the prescribed threshold.*

## VI. Case study

In this section, two practical setups for the threshold PSI protocol are discussed: the small-client-small-server case and the small-client-large-server setup.

**Input:** On the client side: A set $C$, the threshold value $t$; On the server side: A set $S$
**Output:** The client side receives $C \cap S$ if $|C \cap S|$ satisfies the threshold requirement

Initialise the FHE with two LWE encryptions, *i.e.* $LWE^{n,q}_{\mathbf{s}}(\cdot)$ and $LWE^{N,Q}_{\mathbf{z}}(\cdot)$
The client creates the Bloom filter of his own set as a bit table $\mathbf{BF_c}[i]$ for $i = 0, \cdots, |\mathbf{BF_c}| - 1$ and then encrypt $\mathbf{BF_c}[i]$ bit by bit as $LWE^{N,Q}_{\mathbf{z}}(\mathbf{BF_c}[i])$, and finally sends $\{LWE^{N,Q}_{\mathbf{z}}(\mathbf{BF_c}[i])\}_i$ to the server.
The server runs the `ePSI-CA` primitive (Alg. 5) to obtain $LWE^{N,Q}_{\mathbf{z}}(|C \cap S|)$
The server prepares a session key $K$ and a random number $R$, and performs a functional bootstrapping for $LWE^{N,Q}_{\mathbf{z}}(|C \cap S|)$ to obtain $LWE^{N,Q}_{\mathbf{z}}(K')$; The server appends $K$ to every element in his set as $S^K = \{s_i|K\}_i$
The client receives and then decrypts $LWE^{N,Q}_{\mathbf{z}}(K')$ to obtain $K'$. The client appends $K'$ to every element in his set $C$ as $C^{K'} = \{c_i|K'\}_i$
Finally, the client and server engages in a normal PSI protocol with set $S^K$ and set $C^{K'}$
**return** $C^{K'} \cap S^K$

**Algorithm 6:** Framework for threshold PSI via FHE

In the small-client-small-server case, both the size of the client's set and the size of the server's set are small. This scenario represents a small-scale deployment of the threshold PSI protocol. It could be applicable in scenarios where both the client and the server have limited computational resources or when the sets involved are naturally small in size. In such cases, the protocol can be executed efficiently without imposing significant computational or communication overhead.

On the other hand, the small-client-large-server setup caters to scenarios where the client's set is small, but the server's set is large. This configuration is more suitable for cloud computing deployments, where the server possesses substantial computational capabilities and resources, while the client is resource-limited. In this case, the powerful server can handle the heavy computational tasks involved in the protocol, while the client can offload most of the computation to the server, leveraging its computational resources. This setup allows for efficient execution of the threshold PSI protocol, taking advantage of the server's capabilities to process large sets efficiently.

### A. When both client's set and server's set are small

*1) Basic idea:* In this scenario, the proposed framework described in Alg. 6 is directly applied. The client sends the encrypted Bloom filter array of their set, denoted as $LWE^{N,Q}_{\mathbf{z}}(\mathbf{BF}c[i])$ for all $i$, to the server. The server then performs the `ePSI-CA` primitive to obtain the encrypted intersection cardinality $LWE^{N,Q}_{\mathbf{z}}(|C \cap S|)$.

Next, the server performs a functional/programmable bootstrapping to obtain $LWE^{N,Q}_{\mathbf{z}}(K')$ and sends it back to the client. Additionally, the server updates their set $S$ to $S^{(K)} = \{s_i||K\}_i$. Upon receiving the encrypted value $LWE^{N,Q}_{\mathbf{z}}(K')$,

the client decrypts it to obtain $K'$. The client then uses $K'$ to update their set $C$ to $C^{(K')} = \{c_i||K'\}_i$.

Finally, the standard PSI protocol, such as the ECDH-based protocol, is performed between the client's new set $C^{(K')}$ and the server's updated set $S^{(K)}$. This protocol allows the client to retrieve the intersection $C \cap S$ or determine that the intersection is an empty set, depending on the value of $|C \cap S|$.

**Optimizing the communication overhead** Note that in the basic framework shown in Alg. 6, the client encrypts his BF array bit by bit as multiple LWE ciphertexts at the beginning of secret token generation subprotocol. We can leverage RLWE packing techniuqe to batch $N$ LWE cipherexts into a single RLWE ciphertexts by encoding $\mathbf{BF}_c[\cdot]$ as $BF_c(X) = \sum_i \mathbf{BF}_c[i]X^i$. After the server receives this single RLWE ciphertext $RLWE(BF_c(X))$, he performs, for every $i \in [N]$, $RLWE(BF_c(X)) \cdot X^{-i}$ and then homomorphically extracts the constant term which returns $LWE(\mathbf{BF}_c[i])$.

**Computational Complexity** The threshold PSI protocol here is bottlenecked by the performance of `ePSI-CA` primitive in the threshold token generation subprotocol, the computational complexity of which is dominated by the number of bootstraps used. Either half-domain bootstrap or full-domain bootstrap is used depending on the set size. Half-domain bootstrap and Full-domain bootstrap costs $\mathcal{O}(d_g \cdot N^2 \cdot \log N)$ and $\mathcal{O}(\ell_{boot} \cdot d_g \cdot N^2 \cdot \log N)$ mod-$Q$ multiplications, respectively. $\ell_{boot}$ and $d_g$ are small constants used in the TFHE system parameter description. On the other hand, the computational complexity of the ECDH-based standard PSI is $\mathcal{O}((|C| + |S|) \log^2 p)$. To conclude, the entire threshold PSI protocol costs $\mathcal{O}(|S| \cdot \ell_{boot} \cdot d_g \cdot N^2 \log N)$.

**Communication Complexity** The client sends the encrypted Bloom filter in the compressed RLWE ciphertexts *i.e.*, $RLWE_z^{N,Q}(BF_c(X))$ and therefore, the size of the encrypted Bloom filter array is $\frac{|\mathbf{BF_c}|}{N} \cdot 2N \cdot \log Q$ bits. The server later sends back an LWE encryption of the token $K'$, *i.e.*, $LWE_{\mathbf{s}}^{n,q}(K')$ whose size is $(n+1)\log q$ bits. To conclude, the total communication complexity is $\mathcal{O}(|C| \log Q + (|C| + |S|) \cdot \log p + n \log q)$ where `ePSI-CA` takes $\mathcal{O}(|C| \log Q + n \log q)$ and `standard-PSI` takes $\mathcal{O}((|C| + |S|) \log p)$ where $32 \leq \log Q \leq 64$, $12 \leq \log q \leq 14$, $12 \leq n \leq 14$, and $\log p = 256$.

### *B. When the client's set is small but the server's set is large*

Overcoming the technical difficulties related to high precision in TFHE for scenarios where the client's set is small but the server's set is large is indeed crucial. In such cases, the set size can exceed the limitations of TFHE's bootstrapping, which can only support 6-8 bits of precision and a maximum set size of 256.

To address this challenge, we need to tackle two key issues:

- *Homomorphic accumulation with higher precision.* One approach is to perform homomorphic accumulation on LWE encryptions with higher precision, such as a series of LWE encryptions of a 20-bit integer. This requires careful design and parameter selection to ensure that the homomorphic operations are efficient and accurate.
- *Programmable bootstrapping with higher precision.* The final step of the protocol, which involves homomorphic secret token generation (Step 8 in Alg. 6), needs to be performed with higher precision, such as 20 bits. This requires optimizing the bootstrapping process to handle larger bit sizes and ensuring the accuracy of the resulting ciphertexts.

Both technical challenges require careful consideration of the TFHE parameters, efficient implementation, and potential optimizations to achieve the desired precision and performance.

*1) Homomorphic accumulation with higher precision:* In our discussion, we assume that half-domain bootstrapping is used since half-domain bootstrapping is orders of magnitude faster than full-domain bootstrapping. Using half-domain bootstrapping also means that it supports up to $B = \log_2 t - 1$ bit-long plaintext. This plaintext is further partitioned into two parts: 1-bit carry and $(B-1)$-bit message. In general, we use $\ell = \lceil \frac{\mathcal{B}}{B-1} \rceil$ LWE ciphertexts for representing an encrypted $\mathcal{B}$-bit integer $x = |x_{\mathcal{B}-1} \cdots x_0|_2$ as $LWE_{\mathbf{z}}^{N,Q}(|x_{B \cdot i+B-1} \cdots x_{B \cdot i}|_2)$ where the most significant bit $x_{Bi+B-1}$ is set as carry and the other bits $x_{Bi+B-2} \cdots x_{Bi}$ are set as message for $i = 0, \cdots, \ell - 1$.

Two specific programmable bootstraps(PBSs) are used in the proposed high-precision (homomorphic) accumulation algorithm. The carry-bit extract function, denoted as $f_{carry}$, allows the extraction of the carry-bit from the original LWE ciphertext $LWE(\text{bit}||\text{message})$. This function specifically retrieves the carry bit and produces a new LWE ciphertext containing only the carry bit. The message-bit extract function, denoted as $f_{message}$, enables the extraction of the message bits from the original LWE ciphertext $LWE(\text{bit}||\text{message})$. This function retrieves the message bits and produces a new LWE ciphertext containing only the message bits.

Now we can formally describe the homomorphic addition for two $\mathcal{B}$-bit integers in Alg 7. It actually works as a homomorphic version of a carry-propagation adder. $c_0$ is set to be a trivial LWE encryption of zero as $(\mathbf{0}, 0)$ in line 2. The for-loop in line 3-7 pairwisely addss the LSB LWE-ciphertexts which encrypts $x_i, y_i, c_i$ for all $i$ from 0 up to $\ell - 1$, and then uses message-bit extract and carry-bit extract which returns a new LWE ciphertext $LWE(z_i)$ encrypting the message bit and a new LWE ciphertext $LWE(c_{i+1})$ encrypting the carry bit. Finally, the addition results are captured in the new LWE ciphertexts $LWE(z_i)$ for all $i \in [\ell]$ where each plaintext $z_i$ represents $B-1$ bits of the addition result.

**Optimization I: MISD-style Bootstrap** An optimization for homomorphic addition involves utilizing the Multiple-Instruction-Single-Data (MISD) approach [34]. This technique allows us to implement both carry-bit extract and message-bit extract using just one half-domain bootstrap operation.

**Optimization II, lazy addition** In our previous discussion, we observed that a bootstrap operation is required every time two encrypted integers are added. Specifically, for adding an $\mathcal{B}$-bit integer, it is clear that $\ell = \lceil \frac{\mathcal{B}}{B-1} \rceil$ multi-value half-domain bootstraps are needed when the carry length is set to 1 bit. To reduce the number of bootstraps, it is natural to extend the carry length, denoted as $B_c$, to multiple bits, such that $B > B_c > 1$. By doing so, it becomes possible to perform $2^{B_c} - 1$ homomorphic additions without requiring a bootstrap

**Input:** Encrypted $\mathcal{B}$-bit Integer $X = \sum_{i=0}^{\ell-1} x_i 2^{(B-1)i}$ and $Y = \sum_{i=0}^{\ell-1} y_i 2^{(B-1)i}$ with $\ell = \lceil \frac{\mathcal{B}}{B-1} \rceil$
**Output:** Encrypted addition result

Integer $X, Y$ is encrypted as $LWE_{\mathbf{z}}^{N,Q}(x_i)$ and $LWE_{\mathbf{z}}^{N,Q}(y_i)$ for $i \in [\ell]$
$LWE_{\mathbf{z}}^{N,Q}(c_0) \leftarrow (\mathbf{0}, 0)$
**for** $i \leftarrow 0$ ***to*** $\ell - 1$ **do**
  $tmp \leftarrow LWE_{\mathbf{z}}^{N,Q}(x_i) + LWE_{\mathbf{z}}^{N,Q}(y_i) + LWE_{\mathbf{z}}^{N,Q}(c_i)$
  $tmp \leftarrow \texttt{KeySwitch}(tmp, ksK)$
  $tmp \leftarrow \texttt{ModSwitch}(tmp, Q, q)$
  $\{LWE_{\mathbf{z}}^{N,Q}(z_i), LWE_{\mathbf{z}}^{N,Q}(c_{i+1})\} \leftarrow \texttt{Bootstrap}(tmp, \{\texttt{msgExt}, \texttt{carryExt}\}, \texttt{BeforeKeySwitch})$
**return** $\{LWE(z_i)\}_i$

**Algorithm 7:** Homomorphic large-integer addition

operation, as the calculation overflow can be tolerated in the carry. This approach, referred to as "lazy addition," allows for a reduction in the number of bootstraps.

**Computational Complexity** First discuss the complexity of the homomorphic addition. Assume the input integer has $\ell$ digits and each digit is $B$ bits long. Alg. 7 consumes $\ell$ `KeySwitch`s and $\ell$ `ModSwitch`s, and $\ell$ multi-value `Bootstrap`s. `KeySwitch` costs $\mathcal{O}(Nd_{ks})$ $\mathbb{F}_Q$ multiplications, `ModSwitch` costs $\mathcal{O}(N)$ 64-bit floating point multiplications, and multi-value `Bootstrap` costs $\mathcal{O}(d_g N^2 \log N)$ $\mathbb{F}_Q$ multiplications. Therefore, the complexity of homomorphic addition is $\mathcal{O}(\ell N^2 \log N)$. Based on this result, it is easily seen that the homomorphic accumulation costs $\mathcal{O}(|S| \cdot \ell \cdot N^2 \log N)$.

*2) Final PBS (homomorphic secret token generation) with higher precision:* Recall that the purpose of the final Programmable Bootstrap (PBS) is to convert the LWE encryption of $|C \cap S|$ into either an LWE encryption of $K$ or an LWE encryption of $K'$, depending on the value of $|C \cap S|$. In order to address the challenge of the final Programmable Bootstrap (PBS) when dealing with large integers, where the input $|C \cap S|$ may be large, we propose a two-step approach. The first step involves using a homomorphic comparison primitive to obtain an encryption of the comparison indicator. In the second step, this encrypted indicator is then converted to an encrypted $K$ or $K'$ using the half-domain bootstrap directly.

The proposed homomorphic large-integer comparison is described in a recursive manner in Algorithm 8. We utilize the homomorphic sign function (denoted as $sign_B(x)$) introduced in [35] as a fundamental building blockb to compare two relatively large integers, *i.e.*, $\text{compare}_{B^\ell}(X, Y)$ for comparing two integers in the range $[-(B^\ell - 1), B^\ell - 1]$:

$$\text{compare}_{B^\ell}(X, Y) = \begin{cases} 1 & \text{if } X > Y \\ 0 & \text{if } X = Y \\ -1 & \text{if } X < Y \end{cases}$$

In order to convert the encrypted sign function result, $LWE^{N,Q}(sign(X - Y))$, to either $LWE^{N,Q}(K)$ or

**Input:** Encrypted Integer $X = \sum_i x_i B^i$ (smaller than $B^\ell$) and $Y = \sum_i y_i B^i$
**Output:** Encrypted comparison result, *i.e.*, $LWE^{N,Q}(\text{compare}_{B^\ell}(X, Y))$

Integer $X, Y$ is encrypted as $LWE^{N,Q}(x_i)$ and $LWE^{N,Q}(y_i)$
**if** $\ell$==*1* **then**
  **return** $LWE^{N,Q}(sign_B(X - Y))$
**else**
  Collect the first half of the encrypted $X$ and $Y$ as $\mathbf{ct}_{X_0} \leftarrow \{LWE^{N,Q}(x_i)\}_{i=0,\cdots,\frac{\ell}{2}-1}$ and $\mathbf{ct}_{Y_0} \leftarrow \{LWE^{N,Q}(y_i)\}_{i=0,\cdots,\frac{\ell}{2}-1}$ where $\mathbf{ct}_{X_0}$ and $\mathbf{ct}_{Y_0}$ equivalently represents $LWE^{N,Q}(X_0)$ and $LWE^{N,Q}(Y_0)$, respectively
  Collect the second half of the encrypted $X$ and $Y$ as $\mathbf{ct}_{X_1} \leftarrow \{LWE^{N,Q}(x_{i+\frac{\ell}{2}})\}_{i=0,\cdots,\frac{\ell}{2}-1}$ and $\mathbf{ct}_{Y_1} \leftarrow \{LWE^{N,Q}(y_{i+\frac{\ell}{2}})\}_{i=0,\cdots,\frac{\ell}{2}-1}$ where $\mathbf{ct}_{X_1}$ and $\mathbf{ct}_{Y_1}$ equivalently represents $LWE^{N,Q}(X_1)$ and $LWE^{N,Q}(Y_1)$, respectively
  Compute $\mathbf{c}_0 \leftarrow LWE^{N,Q}(\text{compare}_{B^{\frac{\ell}{2}}}(\mathbf{ct}_{X_0}, \mathbf{ct}_{Y_0}))$
  Compute $\mathbf{c}_1 \leftarrow LWE^{N,Q}(\text{compare}_{B^{\frac{\ell}{2}}}(\mathbf{ct}_{X_1}, \mathbf{ct}_{Y_1}))$
  **return** $LWE^{N,Q}(sign_B(\mathbf{c}_0 + 2 \cdot \mathbf{c}_1))$

**Algorithm 8:** Homomorphic large-integer comparison

$LWE^{N,Q}(K')$ depending on the sign of $X - Y$, we compute homomorphically a lookup table (LUT) within the half-domain bootstrap. The LUT allows us to map the encrypted sign function result to the desired encrypted value.

**Computational Complexity** Discuss the complexity of the final PBS. Assume the input integer has $\ell$ digits and each digit is $\log_2 B$ bits long. First, consider the homomorphic comparison primitive. Alg. 8 performs the half domain bootstrap $\ell + \frac{\ell}{2} + \cdots + 1 = 2\ell - 1$ times and thus the complexity of the homomorphic comparison is $\mathcal{O}(\ell N^2 \log N)$. Second, consider the bootstrapping for the function $LUT[\cdot]$ which is performed after the homomorphic comparison. As we have discussed, this homomorphic evaluation of $LUT[\cdot]$ is a normal half-domain bootstrapping and therefore the complexity is $\mathcal{O}(N^2 \log N)$. To conclude, the complexity of the final PBS is $\mathcal{O}(\ell N^2 \log N)$.

*3) Threshold PSI for small-client-large-server case, a communication-optimal solution:* In this subsection, we provide a comprehensive description of our threshold PSI protocol tailored for scenarios where the client's set is small and the server's set is large, optimizing for optimal asymptotic communication complexity $\Omega(|C|)$. In this method, the client computes its own Bloom Filter array $\mathbf{BF_c}$, encrypts it, and sends it to the server. To reduce the communication complexity, an improvement is made by encrypting $N$ bits of $\mathbf{BF_c}$ in one RLWE ciphertext. As a result, the entire $\mathbf{BF_c}$ is enclosed in $\frac{|\mathbf{BF_c}|}{N}$ RLWE ciphertexts.

Upon receiving these RLWE ciphertexts, the server performs homomorphic sample extraction to expand the $\frac{|\mathbf{BF_c}|}{N}$ RLWE ciphertexts into $|\mathbf{BF_c}|$ LWE ciphertexts. Each LWE

ciphertext encrypts a single bit of $\mathbf{BF_c}$. This conversion allows the server to manipulate the encrypted Bloom Filter in a homomorphic manner.

By adopting this approach, the server can perform various operations on the encrypted Bloom Filter, such as homomorphic intersection, cardinality calculation, or any other desired computation related to the PSI protocol. The server can then send the resulting ciphertexts back to the client for decryption and retrieval of the desired PSI results. We revises the original threshold PSI protocol (Alg. 6) by incorporating the high-precision `ePSI-CA` and final PBS to overcome the limitations of low-precision posed by TFHE. Compared with the original threshold PSI, `ePSI-CA` and final PBS are enhanced by their multi-precision versions, respectively.

**Complexity of the communication-optimal solution** The complexity of the proposed solution is dominated by the multi-precision version of `ePSI-CA`, and thus the total complexity is $\mathcal{O}(|S|\cdot\ell\cdot N^2\log N)$. Note that we assume $|S|$ is significantly larger than $|C|$, and thus this method is bottlenecked by this huge computational complexity and thus is impractical in general. It is desirable to find better strategy with computational complexity related to the small $|C|$ rather than the large $|S|$.

*4) Threshold PSI for small-client-large-server case, a performance-balanced solution:* We propose the second approach for the threshold PSI protocol in the small-client-large-server scenario, prioritizing concrete computational efficiency over optimal asymptotic communication complexity, resulting in a sub-optimal communication complexity of $\Omega(|C|\sqrt{|S|})$. In this approach, the client sends the encrypted Bloom Filter array indices of its own set, denoted as $\{LWE_{\mathbf{z}}(index_j)\}_j$, to the server. The server then homomorphically computes $LWE_{\mathbf{z}}(\sum_j \mathbf{BF_s}[index_j])$, followed by $LWE_{\mathbf{z}}(\sum_j \delta_j)$, and finally the encrypted token $LWE_{\mathbf{z}}(K')$.

The challenge here is to compute $LWE_{\mathbf{z}}(\sum_j \mathbf{BF_s}[index_j])$ from $\{LWE_{\mathbf{z}}(index_j)\}_j$, as there are no direct homomorphic operations to support this computation. However, we can overcome this challenge by encoding the indices for every item $c$ in the client's set $C$ as binary vectors. Specifically, each index is represented as a binary vector $\mathbf{index_{bin}}$, where $\mathbf{index_{bin}}[i] = 1$ if $i$ is in the set of indices computed for item $c$, denoted as $\mathbf{index} \leftarrow compute\_indices(c)$, and $\mathbf{index_{bin}}[i] = 0$ otherwise.

Using this binary vector encoding, the server can perform operations on the encrypted Bloom Filter array indices in a homomorphic manner. The computation $LWE_{\mathbf{z}}(\sum_j \mathbf{BF_s}[index_j])$ can be achieved by homomorphically summing the encrypted Bloom Filter array elements corresponding to the non-zero entries in the binary vectors. This allows the server to effectively compute the intersection of the sets and obtain the encrypted token $LWE_{\mathbf{z}}(K')$.

`ePSI-CA` primitive is modified accordingly as shown in Alg. 9. The membership test in the `ePSI-CA` primitive can be simply performed by an inner product between $\mathbf{index_{bin}}$ and $\mathbf{BF_s}$, and then check whether this inner product equals the number of hash function:

$$membership(c \in S) = \begin{cases} True & \text{if } \langle \mathbf{index_{bin}}, \mathbf{BF_s} \rangle = |\mathbf{Hash}| \\ False & Otherwise \end{cases}$$

**Input:** On the client side: A set $C$, the threshold value $t$; On the server side: A set $S$ and $LWE_{\mathbf{z}}^{N,Q}(\mathbf{index_{bin}}[i])$ for all $i$
**Output:** The client side receives an LWE encryption of $|C \cap S|$

Initialise the FHE with two LWE encryptions, *i.e.* $LWE_{\mathbf{s}}^{n,q}(\cdot)$ and $LWE_{\mathbf{z}}^{N,Q}(\cdot)$
The server creates the Bloom filter of his own set as a bit table $\mathbf{BF_s}[i]$ for $i = 0, \cdots, |\mathbf{BF_s}| - 1$
The server performs the following loop to obtain $LWE_{\mathbf{z}}^{N,Q}(|C \cap S|)$
**for** *every* $\mathbf{index_{bin,c_i}}$ *received from the client set* $c_i \in C$ **do**
    Perform homomorphic inner product $\langle LWE_{\mathbf{z}}^{N,Q}(\mathbf{index_{bin}}), \mathbf{BF_s} \rangle$ to obtain $LWE_{\mathbf{z}}^{N,Q}(\langle \mathbf{index_{bin}}, \mathbf{BF_s} \rangle)$
    Perform Key-switching to obtain $LWE_{\mathbf{s}}^{n,q}(\langle \mathbf{index_{bin}}, \mathbf{BF_s} \rangle)$
    Perform bootstrapping (before-key-switch mode) to obtain $LWE_{\mathbf{z}}^{N,Q}(\delta_i)$
Perform homomorphic additions to obtain $LWE_{\mathbf{z}}^{N,Q}(\sum_i \delta_i)$
**return** $LWE_{\mathbf{z}}^{N,Q}(\sum_i \delta_i)$ *to the client*

**Algorithm 9:** core primitive `ePSI-CA` used in the proposed performance-balanced solution

**Communication complexity reduction optimizations** The last unsolved problem is about the communication complexity blow-up in $\mathbf{index_{bin}}$ sent bit by bit from the client to the server. A natural idea for reducing communication complexity is to pack $N$ bits of $\mathbf{index_{bin}}$ into one RLWE ciphertext, and therefore, the total communication overhead for transmitting the encryption of $\mathbf{index_{bin}}$ is as large as $|C|\cdot|\mathbf{Hash}|\cdot\lceil\frac{|\mathbf{BF_s}|}{N}\rceil\cdot 2N\cdot\log_2 Q \approx 2|C|\cdot|\mathbf{Hash}|\cdot|\mathbf{BF_s}|\cdot\log Q$ bits. In our actual experiment, such communication overhead is still large and generally dominates the overall threshold PSI subprotocol.

Inspired by Private Information Retrieval (PIR), folding can be applied to further optimize the `ePSI-CA` primitive in Alg. 9. By transforming the server's Bloom Filter array $\mathbf{BF_s}$ into a $d$-dimensional array and using folding, we can reduce the size of the encrypted $\mathbf{index_{bin}}$ and improve efficiency.

In this approach, the server's Bloom Filter array $\mathbf{BF_s}$ is reshaped into a matrix form $\mathbf{BF_s^{poly}}$ with each entry represented as a polynomial. The polynomial $\mathbf{BF_s^{poly}}[i_x]$ encodes $N$ bits from $\mathbf{BF_s}[i_x N]$ to $\mathbf{BF_s}[i_x N + N - 1]$ as the coefficients of the polynomial, where $i_x$ is the index of the matrix entry.

To extract a specific bit from $\mathbf{BF_s}$ without privacy preservation, we extract the corresponding coordinates $(i_x, i_y)$ in $\mathbf{BF_s^{poly}}$. Here, $i_x$ is represented as a binary vector $\mathbf{i_{x,bin}}$ of size $\mathcal{O}(\sqrt{|S|})$ where $\mathbf{i_{x,bin}}[i_x] = 1$ and 0 elsewhere, and $i_y$ is represented as a monomial $X^{-i_y}$.

The extraction procedure involves an inner product between $\mathbf{i_{x,bin}}$ and $\mathbf{BF_s^{poly}}$ to obtain the $i_x$-th entry of $\mathbf{BF_s^{poly}}$, which is essentially a polynomial denoted as $\mathbf{BF_s^{poly}}[i_x]$. Next, we perform a "left-rotation" by multiplying $X^{-i_y}$ with $\mathbf{BF_s^{poly}}[i_x]$. Finally, we extract the constant term from this rotated polynomial to obtain the desired bit $\mathbf{BF_s}[i]$.

By applying folding and leveraging the structure of the Bloom Filter array, we can reduce the size of the encrypted $\mathbf{index_{bin}}$ and optimize the extraction process, leading to improved efficiency in the threshold PSI protocol for the small-client-large-server case.

The RLWE encryption $RLWE_z^{N,Q}(\cdot)$ has $N$ coefficient slots and therefore we can exploit these slots to reduce request size. If the system parameters are properly set, one RLWE ciphertext suffices to enclose the entire vector $\mathbf{i_{x,bin}}[\cdot]$ if we set $i_{x,bin}(X) = \sum_k \mathbf{i_{x,bin}}[k]X^k$. After the server receives $RLWE_z^{N,Q}(i_{x,bin}(X))$, he computes $LWE_{\mathbf{z}}^{N,Q}(\mathbf{i_{x,bin}}[k]) \leftarrow \texttt{SampleExt}(RLWE_z^{N,Q}(i_{x,bin}(X)), k)$ for $k \in [N]$. Finally, the server uses the LWE-To-RLWE homomorphic key-switching to obtain $RLWE_z^{N,Q}(\mathbf{i_{x,bin}}[k])$. To conclude, a more communication-efficient index expansion procedure goes as follows:

$$RLWE_z^{N,Q}(i_{x,bin}(X)) \xrightarrow{\texttt{RLWEtoLWE}} \{LWE_{\mathbf{z}}^{N,Q}(\mathbf{i_{x,bin}}[k])\}_{i\in[N]} \xrightarrow{\texttt{LWEtoRLWE}} \{RLWE_z^{N,Q}(\mathbf{i_{x,bin}}[k])\}_{i\in[N]}$$

**Communication Complexity** Regarding the communication complexity of the threshold PSI for the small-client-large-server case after the two-dimensional folding trick is applied, Let $|\mathbf{BF_s}|$ ($< N^2$) denote the bit size of the BF array, $|\mathbf{Hash}|$ denote the number of hash functions used in the BF filter, $|C|$ and $|S|$ denote the cardinality of the client set and the server set, respectively. The client needs to encrypt $|C| \cdot |\mathbf{Hash}|$ RLWE ciphertexts and $|C| \cdot |\mathbf{Hash}|$ RGSW ciphertexts which encapsulate the entire set $C$. One RLWE ciphertext is $2N \log_2 Q$ bit long and one RGSW ciphertext is $4d_g N \log_2 Q$ bit long. The total bit size of the encrypted set C is $|C| \cdot |\mathbf{Hash}| \cdot N \cdot \log_2 Q \cdot (2 + 4d_g)$ which is related to the client set $C$ but independent of the server set $S$. Since we assume here the client has a small set C and generally the communication overhead for the `secretTokenGen` subprotocol and more broadly, the `threshold PSI` is also small.

**Computational Complexity** While the communication complexity is significantly reduced, the computational complexity of threshold PSI is heavy where the homomorphic Bloom filter bit extraction procedure introduced in this subsection is the performance bottleneck. Consider the computations for each index $i$, it includes constant-term extraction, LWE-To-RLWE keyswitch, homomorphic inner product and RGSW external product. Among these operations, the LWE-to-RLWE keyswitch bottlenecks the entire Bloom filter bit extraction procedure: each key-switch costs $2N^2 d_{ks}$ $\mathbb{F}_Q$ multiplications and $\frac{|\mathbf{BF_s}|}{N}$ key-switch operations are required meaning that the entire key-switch costs $\mathcal{O}(|\mathbf{BF_s}|Nd_{ks})$ $\mathbb{F}_Q$ multiplications. To summarize, the entire threshold PSI repeats the Bloom filter bit extraction $|C| \cdot |\mathbf{Hash}|$ times and thus costs $\mathcal{O}(|S| \cdot |C| \cdot N \cdot d_{ks} \cdot |\mathbf{Hash}|)$.

## VII. Experimental Results

### A. Threshold PSI Implementations for the Small-Client-Small-Server Case

The proposed methods described in Alg. 6 for the small-client-small-server case were implemented in C++ using the Palisade library. The implementation was done on a system with the following specifications: gcc-9.4.0, Ubuntu 20.04, Intel Xeon Silver 4214R CPU@2.40GHz, and 64GB memory.

The performance of the TFHE-based threshold PSI protocol at an 80-bit security level was evaluated and the results are presented in Table II, which provides information on both communication overhead and computation overhead.

Data (de-)serialization is implemented using the Boost C++ library. This allowed the conversion of ciphertexts, secret keys, bootstrapping keys, and key-switching keys into byte arrays for local storage and transfer over socket packets. The size of the client's set was set to be equal to the size of the server's set, ranging from 10 to 200 elements.

The choice of TFHE parameters is crucial for optimizing the performance of the protocol. In this case, we selected the parameters `HDB:80:7`, `HDB:80:7`, `FDB:80:7`, and `FDB:80:8` for set sizes 10, 50, 100, and 200, respectively. For set sizes 10 and 50, `HDB:80:7` is sufficient because the size of the sets is smaller than $2^{7-1}$. However, for set sizes 100 and 200, we switched to `FDB:80:7` and `FDB:80:8` because the sizes exceed the threshold of $2^7$ and $2^8$ respectively.

It is notable that there is limited existing work on constructing threshold PSI protocols. We are aware of the work by Zhao et al. [5], which focuses on the `SecretTokenGen` subprotocol for $|C| = |S| = 100$. In their work, they propose the use of additive homomorphic encryption scheme Paillier and oblivious polynomial evaluation for constructing `ePSI-CA` and `SecretTokenGen`. In terms of computational overhead, our results are comparable to [5]. We may make improvements by reducing the performance bottleneck through multi-threading acceleration for `ePSI-CA`. Each thread performs an instance of `ePSI-CA` for a subset of $S$, and the resulting ciphertexts are homomorphically summed to obtain the encryption of the intersection cardinality. This approach allows for parallel execution and improves the overall performance of the protocol.

### B. Threshold PSI Implementations for the Small-Client-Large-Server Case

We implemented the performance-balanced solution with all communication overhead reduction techniques enabled. The experimental results shown in Table III are consistent with the analysis of communication and computational complexity.

In terms of communication complexity, the proposed communication overhead reduction techniques, including two-dimensional folding and RLWE packing for homomorphic Bloom Filter bit extraction, effectively reduce the communication overhead. For client set sizes $|C| = 10, 100, 500$, the communication overhead grows linearly as 78.7 MB, 787.1 MB, and 3935.5 MB, respectively. This indicates a reasonable and manageable communication cost even for larger client sets.

However, the computational complexity is heavily bottlenecked by the LWE-to-RLWE key switches in the homomorphic Bloom Filter bit extraction, leading to long execution times for the `SecretTokenGen` subprotocol. In the experiments, the execution time for `SecretTokenGen` is 241.1 seconds, 2619.7 seconds, and 13012.2 seconds for client set sizes $|C| = 10, 100, 500$, respectively.

TABLE II: Timing Performance of the FHE-based Threshold PSI for small sets, 80-bit security level and single-thread.

| Set Size | Communication Overhead (KB) | | | Computation Overhead (seconds) | | |
|---|---|---|---|---|---|---|
| | Total | `SecretTokenGen` | `Standard PSI` | Total | `SecretTokenGen` | `Standard PSI` |
| 10 | 28.1 | 25.3 | 2.8 | 10.217 | 10.169 | 0.048 |
| 50 | 52.9 | 38.8 | 14.1 | 59.681 | 59.590 | 0.091 |
| 100 | 106.9 | 78.8 | 28.1 | 231.227 | 231.107 | 0.120 |
| 100 [5] | n.a. | n.a. | n.a. | n.a. | 225.033 | n.a. |
| 200 | 179.5 | 123.3 | 56.2 | 819.854 | 819.638 | 0.216 |

TABLE III: Timing Performance of the FHE-based Threshold PSI for small sets. The implementation uses high-precision TFHE at 80-bit security level running in multiple threads (maximum 10 threads used).

| Method | $\|C\| : \|S\|$ | Communication Overhead (MB) | | | Computation Overhead (seconds) | | |
|---|---|---|---|---|---|---|---|
| | | Total | `SecretTokenGen` | `Standard PSI` | Total | `SecretTokenGen` | `Standard PSI` |
| ours, threshold PSI | 10:1000 | 78.8 | 78.7 | 0.1 | 241.8 | 241.1 | 0.7 |
| | 100:1000 | 787.2 | 787.1 | 0.1 | 2620.4 | 2619.7 | 0.7 |
| | 500:1000 | 3935.7 | 3935.5 | 0.2 | 13013.0 | 13012.2 | 0.8 |
| | 100:10000 | 788.0 | 787.1 | 0.9 | 23119.5 | 23112.3 | 7.2 |
| [14],circuit-PSI | $2^{20} : 2^{20}$ | 277 | — | — | 103 | — | — |
| [15],circuit-PSI | $2^{8} : 2^{20}$ | 29 | — | — | 3026 | — | — |
| [16],circuit-PSI | $2^{8} : 2^{20}$ | 4.8 | — | — | 12.6 | — | — |

It is worth noting that if the huge communication overhead can be tolerated, a direct implementation of the performance-balanced solution without the homomorphic Bloom Filter bit extraction can significantly reduce the execution time for the `SecretTokenGen` subprotocol. In this case, the execution time is reduced to 16.2 seconds, 105.3 seconds, 345 seconds, and 143.3 seconds for client set sizes $|C| = 10, 50, 100, 200$, respectively. The computational complexity of the `SecretTokenGen` subprotocol is dominated by the `ePSI-CA` primitive, which uses $|C|$ half-domain bootstraps and incurs a complexity of $\mathcal{O}(|C| d_g N^2 \log N)$.

These experimental results highlight the trade-off between communication overhead and computational complexity in the threshold PSI protocol for the small-client-large-server case. Depending on the specific requirements and constraints of the application, the appropriate approach can be chosen to optimize either the communication or computational aspects.

As the first work contemplating the use of Fully Homomorphic Encryption (FHE) to construct threshold Private Set Intersection (PSI), no other available implementations or results have been identified in the open literature. Instead, we conduct a comparative analysis of our threshold PSI protocol with circuit-PSI protocols, as circuit-PSI can be adapted to construct `SecretTokenGen` and subsequently to construct threshold PSI by enhancing generic Multiparty Secure Computation (MPC) protocols. The work by [14] leverages the VOLE primitive and exhibits optimal performance for large and symmetric set sizes. In [15], FHE is employed to enhance communication overhead in asymmetric set scenarios where the client set is small but the server set is large. [16] further optimizes the performance of [15]. Generally, the performance of these state-of-the-art circuit PSI protocols surpasses that of our work. However, it is essential to highlight that the functionality of our threshold PSI protocols is quite different from the circuit protocol. The FHE techniques used in [15], [16] cannot be applied to construct threshold PSI since these works use leveled FHE and are thus restricted by the multiplicative depth they can evaluate. Our primary focus in this study is to explore the feasibility of employing third-generation FHE programmable bootstrapping for constructing a PSI-with-computations protocol. In comparison, our approach achieves optimal round complexity, and our protocols maintain conceptual simplicity and modularity. This simplicity streamlines both security analysis and implementations.

## VIII. Conclusion

In this work, we have embarked on the exploration of constructing private set intersection variants, such as private intersection cardinality and threshold private set intersection, using fully homomorphic encryptions. Specifically, we have utilized TFHE/FHEW-style fully homomorphic encryptions that leverage programmable bootstrapping to incorporate the threshold functionality into the PSI protocol. As a result, we have achieved a linear communication complexity proportional to the size of the sets involved. Our work represents an initial step towards investigating the potential of employing fully homomorphic encryptions for the development of practical threshold PSI protocols.

## Acknowledgments

This research is supported by the National Research Foundation, Singapore and Infocomm Media Development Authority under its Trust Tech Funding Initiative. Khin Mi Mi Aung, Benjamin Hong Meng Tan, Jingwei Hu and Huaxiong Wang were also supported by A*STAR under its RIE2020 Advanced Manufacturing and Engineering (AME) Programmatic Programme (AwardA19E3b0099). Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of National Research Foundation, Singapore and Infocomm Media Development Authority.

## REFERENCES


[1] M. J. Freedman, K. Nissim, and B. Pinkas, "Efficient private matching and set intersection," in *International conference on the theory and applications of cryptographic techniques*. Springer, 2004, pp. 1–19.

[2] C. Meadows, "A more efficient cryptographic matchmaking protocol for use in the absence of a continuously available third party," in *1986 IEEE Symposium on Security and Privacy*. IEEE, 1986, pp. 134–134.

[3] D. N. Hemenway Falk, Brett and R. Ostrovsky, "Private set intersection with linear communication from general assumptions," in *Proceedings of the 18th ACM Workshop on Privacy in the Electronic Society*, 2019, pp. 14–25.

[4] F. Karakoç and A. Küpçü, "Linear complexity private set intersection for secure two-party protocols," in *International Conference on Cryptology and Network Security*, 2020, pp. 409–429.

[5] Y. Zhao and S. S. M. Chow, "Can you find the one for me? Privacy-preserving matchmaking via threshold psi," in *WPES@CCS*. ACM, 2018, pp. 54–65. [Online]. Available: https://doi.org/10.1145/3267323.3268965

[6] P. Hallgren, C. Orlandi, and A. Sabelfeld, "Privatepool: Privacy-preserving ridesharing," in *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. IEEE, 2017, pp. 276–291.

[7] Y. Zhao and S. S. M. Chow, "Are you the one to share? secret transfer with access structure," *Proc. Priv. Enhancing Technol.*, vol. 2017, no. 1, pp. 149–169, 2017. [Online]. Available: https://doi.org/10.1515/popets-2017-0010

[8] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Tfhe: fast fully homomorphic encryption over the torus," *Journal of Cryptology*, vol. 33, no. 1, pp. 34–91, 2020.

[9] L. Ducas and D. Micciancio, "Fhew: bootstrapping homomorphic encryption in less than a second," in *Advances in Cryptology–EUROCRYPT 2015: 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I 34*. Springer, 2015, pp. 617–640.

[10] Y. Huang, D. Evans, and J. Katz, "Private set intersection: Are garbled circuits better than custom protocols?" in *NDSS*, 2012.

[11] B. Pinkas, T. Schneider, G. Segev, and M. Zohner, "Phasing: Private set intersection using permutation-based hashing," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 515–530.

[12] B. Pinkas, T. Schneider, O. Tkachenko, and A. Yanai, "Efficient circuit-based psi with linear communication," in *Advances in Cryptology–EUROCRYPT 2019: 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19–23, 2019, Proceedings, Part III 38*. Springer, 2019, pp. 122–153.

[13] N. Chandran, D. Gupta, and A. Shah, "Circuit-psi with linear complexity via relaxed batch opprf," Cryptology ePrint Archive, Paper 2021/034, 2021, https://eprint.iacr.org/2021/034. [Online]. Available: https://eprint.iacr.org/2021/034

[14] P. Rindal and P. Schoppmann, "Vole-psi: fast oprf and circuit-psi from vector-ole," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2021, pp. 901–930.

[15] T. Lepoint, S. Patel, M. Raykova, K. Seth, and N. Trieu, "Private join and compute from pir with default," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2021, pp. 605–634.

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

[17] F. Karakoç and A. Küpçü, "Enabling two-party secure computation on set intersection," Cryptology ePrint Archive, Paper 2023/609, 2023, https://eprint.iacr.org/2023/609. [Online]. Available: https://eprint.iacr.org/2023/609

[18] L. Kissner and D. Song, "Privacy-preserving set operations," in *Annual International Cryptology Conference*. Springer, 2005, pp. 241–257.

[19] S. Hohenberger and S. A. Weis, "Honest-verifier private disjointness testing without random oracles," in *International Workshop on Privacy Enhancing Technologies*. Springer, 2006, pp. 277–294.

[20] S. K. Debnath and R. Dutta, "Secure and efficient private set intersection cardinality using bloom filter," in *International Conference on Information Security*. Springer, 2015, pp. 209–226.

[21] R. Egert, M. Fischlin, D. Gens, S. Jacob, M. Senker, and J. Tillmanns, "Privately computing set-union and set-intersection cardinality via bloom filters," in *Australasian Conference on Information Security and Privacy*. Springer, 2015, pp. 413–430.

[22] B. Pinkas, T. Schneider, C. Weinert, and U. Wieder, "Efficient circuit-based psi via cuckoo hashing," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2018, pp. 125–157.

[23] S. Ghosh and T. Nilges, "An algebraic approach to maliciously secure private set intersection," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2019, pp. 154–185.

[24] S. Ghosh and M. Simkin, "The communication complexity of threshold private set intersection," in *Annual International Cryptology Conference*. Springer, 2019, pp. 3–29.

[25] A. Bay, Z. Erkin, J.-H. Hoepman, S. Samardjiska, and J. Vos, "Practical multi-party private set intersection protocols," *IEEE Transactions on Information Forensics and Security*, vol. 17, pp. 1–15, 2021.

[26] Y. Wang, Q. Huang, H. Li, M. Xiao, S. Ma, and W. Susilo, "Private set intersection with authorization over outsourced encrypted datasets," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 4050–4062, 2021.

[27] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *Advances in Cryptology–CRYPTO 2013: 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*. Springer, 2013, pp. 75–92.

[28] I. Chillotti, D. Ligier, J.-B. Orfila, and S. Tap, "Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for tfhe," in *Advances in Cryptology–ASIACRYPT 2021: 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6–10, 2021, Proceedings, Part III 27*. Springer, 2021, pp. 670–699.

[29] K. Kluczniak and L. Schild, "Fdfb: Full domain functional bootstrapping towards practical fully homomorphic encryption," *arXiv preprint arXiv:2109.02731*, 2021.

[30] S. Ma, T. Huang, A. Wang, and X. Wang, "Fast and accurate: Efficient full-domain functional bootstrap and digit decomposition for homomorphic computation," *Cryptology ePrint Archive*, 2023.

[31] Z. Liu, D. Micciancio, and Y. Polyakov, "Large-precision homomorphic sign evaluation using fhew/tfhe bootstrapping," in *Advances in Cryptology–ASIACRYPT 2022: 28th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, December 5–9, 2022, Proceedings, Part II*. Springer, 2023, pp. 130–160.

[32] P.-E. Clet, M. Zuber, A. Boudguiga, R. Sirdey, and C. Gouy-Pailler, "Putting up the swiss army knife of homomorphic calculations by means of tfhe functional bootstrapping," *Cryptology ePrint Archive*, 2022.

[33] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970. [Online]. Available: https://doi.org/10.1145/362686.362692

[34] S. Carpov, M. Izabachène, and V. Mollimard, "New techniques for multi-value input homomorphic evaluation and applications," in *Cryptographers' Track at the RSA Conference*. Springer, 2019, pp. 106–126.

[35] F. Bourse, O. Sanders, and J. Traoré, "Improved secure integer comparison via homomorphic encryption," in *Cryptographers' Track at the RSA Conference*. Springer, 2020, pp. 391–416.