# Multi-PGBF: Efficient Oblivious Key-Value Store and Application to Private Set Intersection

Mingli Wu*, Tsz Hon Yuen†, Man Ho Au*, Siu-Ming Yiu§
*Department of Computing, The Hong Kong Polytechnic University, Hong Kong
Email: mingliwu@connect.hku.hk, man-ho-allen.au@polyu.edu.hk
†Faculty of Information Technology, Monash University, Melbourne, Australia
Email: john.tszhonyuen@monash.edu
‡School of Computing and Data Science, The University of Hong Kong, Hong Kong
Email: smyiu@cs.hku.hk

***Abstract*—An oblivious key-value store is a data structure that can encode and decode $n$ key-value pairs in a table of size $m$ obliviously. After encoding, one cannot distinguish the encoded key-value pairs from other key-value pairs in the input domain. In this paper, we first propose a data structure called Peelable Garbled Bloom Filter (PGBF), which encodes the key-value pairs in a similar way to peeling and unpeeling an *onion*. Specifically, it can divide the key-value pair set (i.e., onion) as multiple subsets (i.e., peels) and order them from the outermost peel to the innermost peel by using a counting Bloom filter. However, using a small expansion rate (i.e., $\eta = m/n$) in PGBF will result in a non-empty core issue with non-negligible probability. To handle this issue, we propose Multi-PGBF by combining multiple PGBFs to do the peelings and unpeelings recursively. In addition, we propose a variant C-Multi-PGBF by clustering a large set into small sets to achieve faster encoding efficiency.**

**Our experiments show that Multi-PGBF and C-Multi-PGBF obtain the best encoding and decoding efficiency. Multi-PGBF improves the encoding time of RR (CCS'22) by $65.1\% \sim 77.6\%$, while C-Multi-PGBF improves the encoding time of the clustered RR variant by $60.2\% \sim 64.7\%$. For decoding, Multi-PGBF is $28.6\% \sim 62.4\%$ faster than RR (CCS'22) and $89.7\% \sim 96.3\%$ faster than RB-OKVS (Usenix'23). When integrated into the state-of-the-art two-party and multi-party private set intersection protocols (Eurocrypt'21, Usenix'24), Multi-PGBF and C-Multi-PGBF lead to faster protocols than those using existing OKVS constructions in most settings.**

## I. Introduction

An oblivious key-value store (OKVS) [1], [2], [3] is a hash data structure that can encode $n$ key-value pairs into a table with size $m \geq n$. To retrieve a key's value, one can simply decode by using this key. After encoding, one cannot distinguish an encoded key-value pair with other key-value pairs from the OKVS table. Because of this obliviousness property and high running efficiency, OKVS is widely used in many applications, such as private set intersection (PSI) [4], [5], [6], [7], [8], [2], [9], [3], [10], [11], [7], [12], [13].

### A. Existing OKVSs

There are three important performance measurements for OKVS: encoding efficiency, decoding efficiency, and expansion rate. In this paper, efficiency refers to the computation efficiency. Ideally, we expect an OKVS has high encoding efficiency, high decoding efficiency, and low expansion rate (i.e., approaching the ideal $\eta = 1$). Depending on the real-world applications, an OKVS can have different preferences for these three measurements. For example, in the traditional client-server model, a server encodes its key-value pairs into an OKVS once; then the server responds to multiple queries from the clients by decoding the querying keys. In this case, the decoding efficiency outweighs the encoding efficiency. However, it is non-trivial for an OKVS to be good in all three measurements simultaneously. For example, the polynomial interpolation can be used as an OKVS with the ideal expansion rate $\eta = 1$ (see section II-A). However, its encoding and decoding costs are as high as $O(n \log^2 n)$ [8].

Dong et al. [1] proposed *Garbled Bloom Filter* (GBF), the first OKVS with linear encoding and decoding efficiency. GBF stores each key's value into $k$ bins selected by $k$ hash functions, in which each bin contains a bitwise-XOR share of the value. The encoding and decoding of GBF are relatively efficient, but its expansion rate is large (i.e., $\eta = \lambda \log_2 e$), where $\lambda$ is the statistical security parameter. Setting $\lambda = 40$, its expansion rate is over 57. Also, it uses as many as $k = \lambda$ hash functions, resulting in low encoding and decoding efficiency (i.e., $O(n)$ with a large constant).

Pinkas et al. [2] proposed another OKVS named PaXoS with a low expansion rate (i.e., $\eta = 2.4$). PaXoS is constructed by a cuckoo graph using two hash functions. The cuckoo graph is a hypergraph derived from a cuckoo hash table. To handle the 2-core in the cuckoo graph, the authors exploit an extra stash-like structure to store the key-value pairs in the 2-core by solving a system of equations with time complexity $O((d+\lambda)^3)$, where $d$ is the 2-core size of the cuckoo graph. Here the 2-core of a (hyper)graph is the maximal subgraph with minimum degree 2. When $n$ is not large (e.g., $n = 2^8$), this non-linear part dominates the encoding time of PaXoS. The following work 3H-GCT [3] increases the number of hash functions from 2 to 3 and further decreases the expansion rate to 1.3.

Raghuraman and Rindal [14] proposed a more efficient OKVS (denoted as RR in this paper) with the same structure as PaXoS/3H-GCT. The difference is that all key-value pairs are encoded by doing matrix-based operations. In the encoding, $n$ keys are encoded as a bit matrix with size $n \times m$, where $m$ is the OKVS table size. For each key $x$, in its corresponding

TABLE I: Table $T$ of GBF. $x_1$'s (resp. $x_2$'s) value $v_1 = \alpha_1 \oplus \alpha_2$ (resp. $v_2 = \beta_1 \oplus \beta_2$) are stored as its shares $\alpha_1$ in $T_1$ and $\alpha_2$ in $T_3$ (resp. $\beta_1$ in $T_6$ and $\beta_2$ in $T_8$).

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| $\alpha_2$ | | $\alpha_1$ | | | $\beta_1$ | | $\beta_2$ | | |

row of the matrix, $w$ bits that are hashed by $w$ hash functions are set as bit 1 and the last $d$ random bits are generated by $h(x)$, where $h(\cdot)$ is another hash function. Then it does matrix triangulation to make the matrix into the row echelon form and back-substitution to encode the key-value pairs. In [14], there is a variant (denoted as C-RR in this paper) by clustering a large key-value set into small clusters, in which the key-value pairs can be encoded more efficiently because small sets are cache-friendly.

Bienstock et al. [15] proposed their RB-OKVS by utilizing a linear Gaussian solver [16]. The key idea in [16] is to represent each key as a sequence of pseudorandom bits with length $\omega$. The sequence starts from a hash location in the bin. Then one can sort the sequences based on the start locations and does the Gaussian elimination to assign the sorted keys' values. Bienstock et al. [15] take their experimental evaluations to find the optimum $\omega$ with very low expansion rates ($1.03 \sim 1.1$).

### *B. Overview of our OKVS*

Before introducing our OKVS, we first show how GBF [1] works with $k = 2$ hash functions ($h_1(\cdot), h_2(\cdot)$) and a table $T$ with size $m = 10$ in Table I. We aim to insert three key-value pairs $(x_1, v_1), (x_2, v_2)$, $(x_3, v_3)$ into this GBF. To put $(x_1, v_1), (x_2, v_2)$ into $T$, we generate some random *XOR shares* $\alpha_1, \alpha_2, \beta_1, \beta_2$ such that $v_1 = \alpha_1 \oplus \alpha_2$ and $v_2 = \beta_1 \oplus \beta_2$. We put each XOR share in a bin in $T$ located by the hash functions $h_1(x_1) = 3$, $h_2(x_1) = 1$, $h_1(x_2) = 6$, $h_2(x_2) = 8$.

Suppose that we want to insert $(x_3, v_3)$ into $T$ with $h_1(x_3) = 3$, $h_2(x_3) = 4$. Since $\alpha_1$ is already assigned to bin 3, we can only calculate $\gamma_3 = v_3 \oplus \alpha_1$ and assign $\gamma_3$ to bin 4. We cannot assign XOR shares if all bins located by the hash functions have already been assigned. To avoid this dilemma, GBF ensures that each key can be mapped to at least one empty bin with probability $1 - 2^{-\lambda}$ by setting a large $k = \lambda$ and a large expansion rate $\eta = \lambda \log_2(e)$, which results in a large performance deterioration.

**PGBF**. Based on the above observation, we first propose Peelable Garbled Bloom Filter (PGBF). Instead of increasing $k$, we propose an innovative method to order the keys and assign the keys' values sequentially. Following the above example, in case that all $x_3$'s hashed bins are occupied by $x_1$ and $x_2$ (e.g., $h_1(x_3) = 3$, $h_2(x_3) = 6$), we encode $x_3$ before $x_1$ or $x_2$. The encoding will succeed if we do it in any one of the three sequences $[(x_3, v_3), (x_1, v_1), (x_2, v_2)]$, $[(x_1, v_1), (x_3, v_3), (x_2, v_2)]$, and $[(x_2, v_2), (x_3, v_3), (x_1, v_1)]$. From this example, we know that finding a proper sequence of encoding is of vital importance. However, it is non-trivial to find the correct sequence for a large key set. Theoretically, the number of possible sequences is $n!$ (i.e., the permutation number of $n$) for $n$ keys.

To resolve this challenge, we utilize a counting Bloom filter (CBF)[17], in which each bin records the number of times that it is hashed. In the above example, we build a CBF $B$. Then we have $B_1 = B_8 = 1$, $B_3 = B_6 = 2$. By checking $B$, we can easily know that the corresponding bins of $(x_3, v_3)$ are occupied by other key-values (i.e., $B_3 > 1$, $B_6 > 1$) and should be encoded earlier than others; also $(x_1, v_1)$ and $(x_2, v_2)$ can be encoded later because they respectively have unique bins $T_1$ (corresponding to $B_1 = 1$) and $T_8$ (corresponding to $B_8 = 1$).

After building the CBF $B$ for the set, we iteratively find keys with unique bins and sequentially get them out to be a separate sub-set. In each iteration, the corresponding bins of the sub-set in $B$ also need to be deducted. We call this process as "peeling" because it is akin to peeling an onion (i.e., deducting a sub-set from the set). After all the keys are peeled, we assign their values in the reverse order of the peeling sequence to finish the encoding.

**Multi-PGBF**. PGBF can be more efficient than GBF by setting $k$ as small as 3. However, there is a chance of encoding failure by a non-empty core, in which the bin counts of all items are $\geq 2$. It is non-trivial to find a low expansion rate to make the failure probability negligible. For example, even we increase $\eta = 1.3$ to $\eta = 8$ for $n = 2^{20}$, there is still a non-empty core with a non-negligible probability. Therefore, we further propose Multi-PGBF by recursively using multiple PGBFs. We observe that PGBF works as a filter to process a large set and get a smaller set (i.e., the core). Then we use another PGBF to filter the smaller set again. If the smaller set is still unpeelable, then we continue to use more PGBFs to filter until we get an empty core. To assign the values, we simply do the unpeeling reversely. Details are in section IV.

**C-Multi-PGBF**. Motivated by C-RR [14], we further propose C-Multi-PGBF that is a cluster variant of Multi-PGBF. After hashing a large key-value set into small clusters, in which the key-value pairs can be encoded by Multi-PGBFs. Since a small key-value set is more cache-friendly, C-Multi-PGBF can encode more efficiently than Multi-PGBF. More details are shown in section IV-E.

### *C. Our contributions*

Compared with the state-of-the-art OKVSs, Multi-PGBF and C-Multi-PGBF has many strengths. In summary, we make the following contributions.

**Fastest encoding**. Multi-PBGF and C-Multi-PGBF obtain the best encoding efficiency. For example, Multi-PBGF encodes $65.1\% \sim 77.6\%$ and $35.6\% \sim 78.3\%$ faster than RR [14] and RB-OKVS (0.1) [15] (i.e., RB-OKVS with expansion rate $\eta = 1.1$). C-Multi-PGBF is $60.2\% \sim 64.7\%$ as fast as C-RR [14] in encoding.

**Fastest decoding**. Multi-PGBF has the best decoding efficiency for all set sizes. To decode a key to get its value, Multi-PGBF only needs to collect $k = 9$ value shares. In

comparisons, RR [14] and RB-OKVS [15] respectively need to collect $3 + 41/2 \sim 3 + 43/2$ and $169/2 \sim 662/2$ value shares, which is $2.6\times \sim 2.7\times$ and $9.4\times \sim 36.8\times$ as Multi-PGBF. In decoding $n$ keys, Multi-PGBF is respectively $28.6\%\% \sim 62.4\%$ and $89.7\% \sim 96.3\%$ faster than RR [14] and RB-OKVS (0.1) [15]. C-Multi-PGBF is $22.9\% \sim 65.1\%$ faster than C-RR [14]. Fast decoding can be more significant than fast encoding. For example, in the classical client-server model, a server can encode the key-value database just once but be queried (i.e., decoded) a lot of times by clients. Then the advantage of fast decoding can be further amplified.

**Encoding detachment**. In both Multi-PGBF and C-Multi-PGBF, the encoding can be detached as peeling phase and unpeeling phase. The peeling phase is only related to the keys and the unpeeling phase involves the values. Therefore, when the encoding needs to be invoked multiple times for the same keys, the encoding advantages of Multi-PGBF can be further amplified by peeling only once and unpeeling multiple times. For Multi-PGBF, the unpeeling only accounts for $13.1\% \sim 26.6\%$ of the encoding time. Compared with naively encoding $\theta$ times, the detachment improvement can save $73.4\%(1 - 1/\theta) \sim 86.9\%(1 - 1/\theta)$ computation costs. In comparison, both RR [14] and RB-OKVS [15] are matrix-based and the row operations involve both the keys and values. It is unknown how to have the detachment as Multi-PGBF.

**Fast PSI.** To demonstrate the efficiency of Multi-PGBF, we apply Multi-PGBF and C-Multi-PGBF to state-of-the-art PSI protocols and compare the performance with those using other OKVSs. Specifically, in the two-party PSI protocol [10] and multi-party PSI protocols O-Ring/K-Star [13], we respectively replace PaXoS and 3H-GCT with RR/C-RR [14], RB-OKVS [15], and Multi-PGBF/C-Multi-PGBF. As a result, the protocols with Multi-PGBF/C-Multi-PGBF generally run faster than the corresponding protocols with other OKVSs. In addition to the performance, we identify that a two-party PSI protocol [18] (NDSS'26) is insecure. Another multi-party PSI protocol [19] (NDSS'26) has misunderstandings of O-Ring/K-Star [13] in unbalanced set sizes and makes unfair performance comparisons. After making a slight modification to O-Ring and K-Star [13], we enable them to support unbalanced settings with better performance.

## II. Preliminaries

We denote $[n]$ as the set $\{1, 2, \cdots, n\}$. Also, $\{x_i\}_n$ denotes the set $X = \{x_1, x_2, \cdots, x_n\}$. Then we have $Y = \{y_i\}_n$. '$\oplus$' is the bitwise XOR operator. We denote $\bigoplus_{\forall i \in [k]} x_i$ as the XOR sum $x_1 \oplus x_2 \oplus \cdots \oplus x_k$.

### A. Oblivious key-value store (OKVS)

An OKVS is a data structure that can encode and decode key-value pairs obliviously.

**Definition II.1.** (KVS) [3]. A key-value store (KVS) is parameterized by a set of $n$ distinct keys $X$ in a key space $\mathcal{K}$, a set of $n$ values $V$ in a value space $\mathcal{V}$, a set of functions $H$, and a failure probability $\epsilon$; and consists of two algorithms:

- $D \leftarrow Encode_H(\{(x_i, v_i)\}_n)$ or $\perp \leftarrow Encode_H(\{(x_i, v_i)\}_n)$. For $n$ key-value pairs $\{(x_i, v_i)\}_n$ from $\mathcal{K}^n \times \mathcal{V}^n$, store each $v_i$ in $D$. With a probability $\leq \epsilon$, there exists the case that a key's value cannot be stored, the encoding outputs $\perp$; otherwise outputs an object $D$.
- $v \leftarrow Decode_H(D, x)$. Given an object $D$ and a key $x$, return a value $v$. Specifically, if $x \in \{x_i\}_n$ and $x = x_i$, then output $v = v_i$; otherwise, output another value.

In the above definition, the object $D$ can be a table. The parameter $H$ depends on the detailed KVS scheme. In the rest of this paper, we choose to omit $H$ for simplicity. A representative KVS is cuckoo hashing [20], in which one can efficiently store $n$ key-value pairs into a hash table and $H$ is a set of hash functions.

**Definition II.2.** (Oblivious) [3]. A KVS is oblivious (OKVS) if, for all two distinct key sets $\{x_1^1, x_2^1, \cdots, x_n^1\}$ and $\{x_1^2, x_2^2, \cdots, x_n^2\}$ from $\mathcal{K}^n$, if encoding does not fail, then $D^1 \leftarrow Encode(\{(x_i^1, v_i)\}_n)$ and $D^2 \leftarrow Encode(\{(x_i^2, v_i)\}_n)$ are computationally indistinguishable, where $\{v_1, v_2, \cdots, v_n\}$ are drawn uniformly at random from $\mathcal{V}$.

From this definition, if the encoded values are random, the two encoding output tables $D^1$ and $D^2$ are indistinguishable. Also, after obtaining $D^1$ and $D^2$, one cannot distinguish the two key sets $\{x_1^1, x_2^1, \cdots, x_n^1\}$ and $\{x_1^2, x_2^2, \cdots, x_n^2\}$. Naively, after getting an OKVS table $D$, one cannot know whether a key is encoded or not, thus maintaining oblivious. Having obliviousness, if a querying key $x$ is not in the encoded key set, $Decode(D, x)$ is a random value. A naive OKVS can be constructed by using polynomial interpolation. By solving the linear equations $P(x_1) = v_1$, $P(x_2) = v_2$, $\cdots$, $P(x_n) = v_n$, one can encode $\{(x_i, v_i)\}_n$ into a univariate polynomial $P(\cdot)$ whose $n + 1$ coefficients $(D_0, D_1, \ldots, D_n)$ are stored in $D$. To decode a key $y$, one can simply calculate $v = \sum_{i=0}^{n} y^i D_i$.

**Definition II.3.** (Linear) [3]. An OKVS is linear (over a field $\mathbb{F}$) if $\mathcal{V} = \mathbb{F}$ ("values" are elements of $\mathbb{F}$), the output of encoding is a vector $D$ in $\{\mathbb{F}^m\}$, and the decoding function is defined as:

$$Decode(D, x) = \langle d(x), D \rangle \stackrel{def}{=} \sum_{i=1}^{m} d(x)_i D_i.$$

for some function $d : \mathcal{K} \mapsto \mathbb{F}^m$ . Hence $Decode(\cdot, x)$ is a linear map from $\mathbb{F}^m$ to $\mathbb{F}$.

For a linear OKVS, one can take the encoding as solving a system of linear equations

$$\begin{bmatrix} d(x_1) \\ d(x_2) \\ \vdots \\ d(x_n) \end{bmatrix} \begin{bmatrix} D_1 \\ D_2 \\ \vdots \\ D_m \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}.$$

A binary OKVS over a field $\mathbb{F}$ is a special case of a linear OKVS, where the $d(x_i)$ ($\forall i \in [n]$) vectors are restricted to $\{0, 1\}^m \subset \mathbb{F}^m$. Then the decoding of a key $x$ can be represented as the XOR sum of $x$'s mapped bins in the table

$D$. It is easy to know that the traditional interpolation is a linear OKVS. The other existing OKVSs (including GBF [1], PaXoS [2], 3H-GCT [3], RB-OKVS [15]) are binary OKVSs. In this paper, our proposed Multi-PGBF is also a binary OKVS (see section IV-D for details).

### B. Bloom filter

A Bloom filter [21] is a probabilistic hash-based data structure that can represent a set of items. In a Bloom filter, there are $k$ independent uniform hash functions $\mathcal{H} = \{h_1(\cdot), h_2(\cdot), \cdots, h_k(\cdot)\}$ and a hash table $T$ of size $m$ in which each bin is a single bit. $T$ is initiated as all '0' and $T_i$ is its $i$-th bin. To profile $n$ items $\{x_i\}_n$ into the array, one just needs to hash each item $x_i$ by all $k$ hash functions and set $T_{h_1(x_i)} = 1$, $T_{h_2(x_i)} = 1$, $\cdots$, $T_{h_k(x_i)} = 1$. To query an item $y$, one can check its $k$ hash bins. If all $T_{h_1(y)}$, $T_{h_2(y)}$, $\cdots$, $T_{h_k(y)}$ are '1's, then take $y \in \{x_i\}_n$; otherwise not. There is no false negative for Bloom filter, i.e., if the query returns not all '1's, then $y \notin \{x_i\}_n$. However, there are false positives, i.e., if $y \notin \{x_i\}_n$, the query may still return all '1's. One drawback of Bloom filter is that it does not support item deletion. In a variant called counting Bloom filter (CBF)[17], each bin of the hash table $T$ can be used as a counter and is enlarged to several bits instead of a single bit. For simplicity, we also call the hash table of a CBF a counting table.

## III. Peelable Garbled Bloom Filter

In this section, we propose a linear OKVS called Peelable Garbled Bloom Filter (PGBF). The construction of PGBF is similar to peeling and unpeeling an *onion*.

### A. Motivation

GBF uses Bloom Filter to assign the XOR shares of each key's value (Algorithm 1 of [1]). The Bloom filter ensures at least one of each item's $k$ bins is uniquely mapped by this item with a predefined failure probability $\epsilon = 2^{-\lambda}$ (i.e., the false positive). Therefore, the final XOR share of each key's value can be assigned to its unique bin. However, GBF has three major weaknesses:

1) Large expansion rate. To ensure the failure probability $\epsilon \leq 2^{-\lambda}$, the hash table size needs to be $m = \lambda n \log_2 e$. The expansion rate is $\eta = \lambda \log_2 e$, which is $> 57\times$ as expensive as the ideal expansion rate 1 by setting $\lambda = 40$. The high expansion rate will result in high communication costs in online interactions between different parties.
2) High encoding costs. Though in linear (i.e. $O(\lambda n)$), the constant $\lambda$ is large, resulting in high encoding costs.
3) High decoding costs. Each item is mapped by as many as $k = \lambda$ hash functions. To decode an item, one has to collect its $k = 40$ shares to calculate the XOR sum. In online interactions, the high decoding costs will increase online computation costs.

### B. High level idea

The above weaknesses of GBF can be overcome by the following observations:

1) The number of hash functions $k$ can be reduced. In Bloom filter, the optimal number of hash functions is $k = \frac{m}{n} \ln 2$. In GBF, $k$ is set to be large to ensure low false positives in Bloom filter. However, introducing more hash functions itself can bring more hash collisions. For example, for $k = 3$, one needs only to hash $3n$ times to the hash table; however, for $k = 40$, one needs to hash $40n$ times to the hash table, which will obviously cause more collisions. In an extreme case, for $k = m$, all items will collide with an overwhelmingly large probability. In comparison, another hash-based structure *Cuckoo hashing* uses only 2 or 3 hash functions [4], [5], [6]. Therefore, it is possible to use less number of hash functions.
2) The size of the hash table $m$ can be smaller. Intuitively, the larger $m$, the fewer collisions for the items. To ensure every item owns its unique bin, $m$ appears to be large. However, we aim to get a small $m$. The collision problem reminds us of the idea of resolving process deadlock in the operating system. Now, considering that we have $n$ processes (corresponding to the $n$ keys), and each process needs to consume $k$ resources (corresponding to the $k$ hash bins). If there is a deadlock (keys collision), there are two general approaches: 1. increase the resources; 2. adjust the execution order of the processes (keys). Since we do not want to adopt the first approach (by increasing $m$), we attempt the second approach. Now, we try to adjust the assigning order of the keys to keep $m$ low.

Based on the above two observations, we devise a new data structure *Peelable Garbled Bloom Filter* (PGBF) that integrates the advantages of both cuckoo hashing that uses a few numbers of hash functions and the GBF's certainty by encoding a key's value as XOR shares.

### C. Our construction

In the above, we have introduced the general idea of PGBF. In this subsection, we will show the details about how to construct PGBF. We first have the following definitions:

- *Anchor*. A key's anchor is the hash function that maps it to a hash table bin uniquely occupied by this key and the unique bin can be utilized to assign the final share of the key's value. Correspondingly, the mapped bin is called the *anchor bin* of this key. Since each key is mapped by $k$ hash functions, a key can have no more than $k$ anchors. It is noted that we only need one anchor to assign the last share of the value.
- *Peelable or unpeelable*. If a key has an anchor, we call it peelable; otherwise, we call it unpeelable. Similarly, if all the keys of a key-value set are peelable, then we call this set is peelable; otherwise, we call this set unpeelable. If a key-value set is $\emptyset$, it is peelable by default.

It is noted that our goal is to find an ordering sequence of the keys to assign their corresponding values' shares. The core idea is to recursively remove some key-value pairs first and delete them from the current set until the set is empty. This process is akin to peeling an *onion* from the outermost peel to the innermost peel. Here, the *onion* is the initial set and the peels are its subsets. After peeling, we need to assign the keys' value shares by unpeeling in a reversed order. We do unpeeling from the innermost peel to the outermost peel because the inner peel is covered by the outer peel, which is intuitive. Specifically, we recursively assign the keys' value shares in the inner peel by assigning their last shares to their corresponding anchor bins.

Generally, there are two main phases for the encoding: 1. the peeling phase that finds the ordering sequence; 2. assigning phase (or unpeeling phase) that assigns the value shares. The details are in Figure 1. In the peeling phase, if a key is hashed to the same bin for an even number of times, then the share in this bin can be *self-cancelled* and this bin cannot be the key's anchor bin. For example, for a key $x_i$, if it has 2 collided hash locations (e.g., $h_1(x_i) = h_2(x_i)$), then the share sum for $D_{h_1(x_i)} \oplus D_{h_2(x_i)} = 0$, no matter what value share we put in this bin. We can easily know that the *self-cancelled* probability for a key is $\geq 1/m$ when $k > 1$, which is not negligible. Therefore, if a key's value shares can be self-cancelled for $a$ times, we can decrease the corresponding counter in the counter table by $2a$. By deleting the *self-cancelled* counters, we can increase the peeling success probability. Then the peeling process can be simplified to find the keys with counting number 1. If the anchor of key is found, then one needs to reduce its counters by 1 in the peeling. It is also noted that the encoding fails when the remaining set cannot be peeled. Here, we call this unpeelable remaining set the *core* of PGBF.

After showing the encoding algorithm, it is easy to know the decoding algorithm. To decode a key $x$ to retrieve its value, one can simply compute

$$Decode(x) = \bigoplus_{\forall j \in [k']} D_{h_j(x)}.$$

### D. Peelability

If there is no core for a PGBF onion, this onion is peelable. Now we study the peelability, which is the probability that an onion can be successfully peeled. Different from GBF which directly uses the Bloom filter's false positive probability to determine its failure probability, PGBF is a recursive structure. The anchors of the items have to be recursively detected peel by peel. Therefore, it is non-trivial to give a precise peelability. In most applications, to ensure an overwhelmingly large peelability (i.e., $1 - 2^{-\lambda}$), the failure probability $2^{-\lambda}$ is often considered alternatively. We theoretically prove that the failure probability of PGBF is no less than that of cuckoo hashing with the same $k$, $m$, and $n$. We also prove that the failure probability of PGBF is no greater than the one for peeling a cuckoo graph (i.e., a random hypergraph). Due to the space limit, we put the proof in Appendix A.

Utilizing the peelability upper and lower bounds can help us to find appropriate parameters for PGBF. For example, Kolesnikov et al. [22] set the expansion rate as 1.2 when using $k = 3$ hash functions in cuckoo hashing with a small stash size. By setting $k = 3$ in PGBF, we can directly find the optimum parameters from $\eta \geq 1.2$. In another example, 3H-GCT [3] that is built based on cuckoo graph has a expansion rate 1.3 by setting $k = 3$ with a small 2-core size bound. Then we can search the expansion rate for PGBF $\eta \leq 1.3$.

### E. Concrete instantiations for PGBF

*1) How to choose $k$?:* The number of hash functions $k$ is important to the encoding and decoding efficiency of PGBF. Given $n$ and $m$, a larger $k$ naively implies larger encoding and decoding costs. More importantly, it affects the peelability of PGBF. In previous works [23], [6], the authors both found that $k = 3$ performs better than $k = 2$ in terms of failure probability for cuckoo hashing without a stash by a given $n$ and $m$. When increasing to $k = 4$, the improvement is marginal compared with the increasing costs. However, in PGBF, we observe that the expansion rate for $k = 3$ is lower than $k = 4$. Also, setting an even $k$ itself has a self-cancellation risk, no matter what value we set.

In PGBF, there may remain a core set for the onion after the peeling process. The size of this core shows the PGBF's ability to peel the original key set. The smaller the core size, the stronger the peeling ability of the PGBF. We empirically evaluate the final filter rate as the ratio between the core size and original set size for $k = 2, 3, 4$ by fixing $n = 2^{20}$ in Fig. 2. For other set sizes, they also show similar trends. The ideal core size is 0 and the ideal final filter rate is 0. From the subfigures, we have two observations:

1) To approach the ideal final filter rate 0, the expansion rate $\eta$ works in different ranges (i.e., $\eta \geq 2.0$ for $k = 2$, $\eta \geq 1.23$ for $k = 3$, $\eta \geq 1.3$ for $k = 4$) and granularities (i.e., 0.1 for $k = 2$, 0.01 for $k = 3, 4$);
2) Both final filter rates show a logarithmic decreasing trend with the increase of $\eta$. For $k = 2$, this decreasing trend goes smoothly. However, for $k = 3$ (resp. $k = 4$), it first shows a almost linear trend when $\eta \in [1.16, 1.22]$ (resp. $\eta \in [1.20, 1.29]$); then decreases sharply from $\eta = 1.22$ to $\eta = 1.23$ (resp. $\eta = 1.29$ to $\eta = 1.30$).

From the first observation, we select $k = 3$ because it has the smallest expansion rate to achieve the same peelability compared with $k = 2$ and $k = 4$.

*2) How to choose $m$?:* We aim to find the smallest $m$ to encode $n$ key-value pairs with the failure probability $\leq 2^{-\lambda}$. Given $n$ and the expansion rate $\eta$, the hash table size can be deducted by $m = \eta n$. Therefore, we probe the smallest $\eta$ to achieve the goal alternatively. From the second observation, we find that there is a critical expansion rate for a set size $n$. For example, $\eta = 1.23$ for $n = 2^{20}$. For other set sizes, we also observe that $\eta = 1.24$ for $n = 2^{16}$ and $\eta = 1.3$ for $n = 2^{12}$. However, we have an interesting finding: even we increase the expansion rate a lot, there are still failure cases. For example, when we increase $\eta = 1.3$ to $\eta = 8$ for $n = 2^{12}$,

**Inputs**: A key-value set $\{(x_i, v_i)\}_n$.
**Parameters:**
A hash table $D$ with size $m$. A corresponding counting table $B$ with size $m$. Hash functions $\mathcal{H} = \{h_i(\cdot)\}_{k'}$, where $h_i(\cdot) : \{0,1\}^\sigma \to [m]$.
**Algorithm:**
leftmargin=0.4cm

1) **Initiation.** Randomly sample a table $D$. Initialize all bins of $B$ as 0. For each key $x_i$, get its bin locations $h_1(x_i)$, $h_2(x_i)$, $\cdots$, $h_{k'}(x_i)$ and increase the $k'$ corresponding counters in the counting table $B$ by 1. Prepare an empty stack $S$.
2) **Peeling the keys**. Denote the key set $\{x_i\}_n$ as $K_1$. For each $x_i \in K_1$, if there exists an anchor $j \in [k']$ that $B_{h_j(x_i)} = 1$, push $(x_i, v_i, j)$ into $S$, peel $x_i$ from $K_1$, and decrease all $B_{h_1(x_i)}, B_{h_2(x_i)}, \cdots, B_{h_{k'}(x_i)}$ by 1. Recursively peeling $K_1$ until all keys in $K_1$ are not peelable (i.e., a core $K_2'$). For each key in $K_2'$, if its hashed bins are self-cancelled, delete the bin counters and peel $K_2'$ to get a core $K_2$. If $|K_2| \neq 0$, the encoding fails; otherwise, the encoding continues.
3) **Assigning value shares.** The assigning process also goes recursively but in a reverse order. Specifically, for $(x, v, j^*) = S_n, S_{n-1}, \cdots, S_1$, assigning the final share of $x$'s value $v$ into its anchor bin as

$$D_{h_{j^*}(x)} = \Big(\bigoplus_{\forall j \in [k'] \setminus \{j^*\}} D_{h_j(x)}\Big) \oplus v.$$

Fig. 1: The encoding algorithm of PGBF by peeling the keys and assigning (i.e., unpeeling) the values.

there are still failure cases (i.e., a non-empty core) in our trials. Utilizing a larger expansion rate may help, however, the table becomes incompact, which is not expected.

## IV. Multi-PGBF

In this section, to handle the incompact issue of PGBF, we propose *Multi-PGBF* by recursively utilizing multiple PGBFs. *Multi-PGBF* is not only compact, but also flexible and can ensure negligible failure probability.

### A. Our construction

In PGBF, we put our focus on the peeling process and calculate the peelability. Now, we emphasize the basic function of PGBF: it is a filter. As a filter, it accepts an input key-value set (i.e., an onion) and filters out a part of key-value pairs (i.e., the peels) to get a remaining set (i.e., the core). *We propose to use another filter to accept the remaining set as the input set and filter it again to get the second remaining set.* This can be done recursively until the final remaining set is empty. This recursive filtering process is exactly the idea of peeling. The difference is that we need to peel several onions instead of one as a single PGBF, which also means that we need to use more hash functions. Since a single PGBF is a linear OKVS, Multi-PGBF is also a linear OKVS. We illustrate the recursive peeling strategy in Fig. 3. In this figure, we first peel a large onion to get its core. Then we take this core as a smaller onion and peel it to get a smaller core. The peeling is recursively done in $u$ rounds until the final onion has an empty core. In general, the peeling is implemented both onion by onion and peel by peel. Also, the assigning or unpeeling goes in a reverse direction. We first assign the final shares of keys from the innermost peel to the outermost peel for the $u$-th onion. Then we assign the keys' shares for the $(u-1)$-th onion in the same way. Finally, we finish the assigning by unpeeling the outermost peel of the first onion. The encoding algorithm for Multi-PGBF is shown in Fig. 4.

As for the *decoding* algorithm, we only need to collect the $k = k_1 + k_2 + \cdots + k_u$ XOR shares for a key in the sub-hash table $D_1, D_2, \cdots, D_u$ and compute the XOR sum:

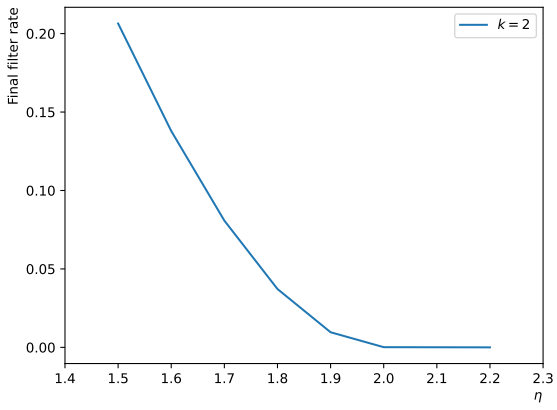
$$D(x) = \bigoplus_{\substack{\forall i \in [u] \\ \forall j \in [k_i]}} D_{i, h_{i,j}(x)}, \tag{1}$$

where $k_i$ and $D_i$ are respectively the numbers of hash functions and hash tables of the $i$-th PGBF. To decode $n$ keys, the computation complexity is $O(kn)$.

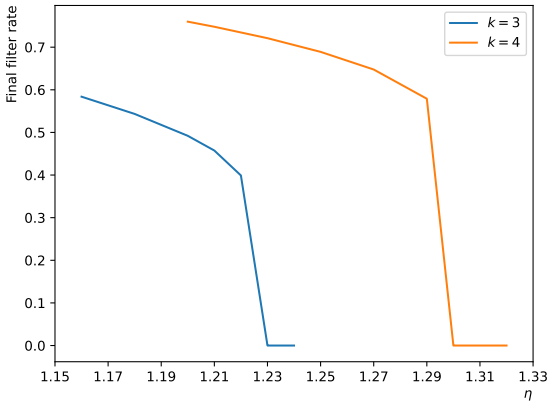
### B. Peelability

The structure of Multi-PGBF is flexible. Ideally, Multi-PGBF can ensure $100\%$ peelability by dynamically adjusting the number of PGBF depending on the input set. Therefore, we have no core concerns. In comparison, a single PGBF cannot achieve that by adjusting the number of hash functions $k$ or the hash table size $m$ during the inserting process. For a single PGBF, $k$ and $m$ have to be set to ensure a low failure probability in advance.

For Multi-PGBF, in practice, we still need to set appropriate parameters to ensure an overwhelmingly low failure probability to maintain obliviousness. It is noted that adding more PGBFs than a predefined one may leak the set information. For the concrete parameters, we need to set the number of PGBFs $u$, the hash table size $m_i$, and the number of hash functions $k_i$ for each PGBF. On the one hand, the number of hash functions for the first onion (i.e., $k_1$) and its hash table size $m_1$ have to be appropriately set before peeling the input set. If $k_1$ or $m_1$ is too large, then extra resources will be wasted. If $k_1$ or $m_1$ is




(a) $k = 2$




(b) $k = 3, 4$

Fig. 2: The final filter rates (i.e., $\frac{|K_2|}{n}$ ) for different expansion rate $\eta$ when $k = 2, 3, 4$. $K_2$ is the core size.

too small, then a large portion of the input set needs to be re-peeled in the following onions, which will also waste extra resources. Furthermore, in the secure computation between multiple parties, the number of the PGBFs, the size of the hash tables, or the number of hash functions may reveal some input information to the other parties.

Since the peeling accepts random inputs, it is possible for any one of the $u$ PGBF in Multi-PGBF to finish peeling in advance. In the case that the $i$-th PGBF finishes the peeling and $i < u$, the peeling can stop in the $i$-th PGBF and Multi-PGBF can assign the value shares from the $i$-th onion to the first onion. Denote the failure probability of the $i$th PGBF is $\epsilon_i$. The failure probability of Multi-PGBF is

$$\epsilon = \epsilon_1 \epsilon_2 \cdots \epsilon_u. \tag{2}$$

Denote $g(m, n)$ as the failure probability that inserting $n$ keys into a table with size $m$. Then for the failure probability of the first PGBF, we have

$$\epsilon_1 = g(m_1, n) = \sum_{j_1=2}^{\bar{\alpha}_1} f(m_1, n, j_1). \tag{3}$$

Here $f(m, n, i)$ is the probability that the core size is $i$ after inserting $n$ keys in a table with size $m$. Correspondingly, we denote $f^{-1}(m, n, \epsilon)$ as core size that the probability is $\epsilon$ after inserting $n$ keys into a table with size $m$. $\bar{\alpha}_i$ is the core size upper bound of the $i$th PGBF (i.e., the probability that the core size $\alpha_i \geq \bar{\alpha}_i$ is negligible). For the failure probability until the second PGBF, we have

$$\epsilon_1 \epsilon_2 = \sum_{j_1=2}^{\bar{\alpha}_1} f(m_1, n, j_1) * g(m_2, j_1). \tag{4}$$

Here $g(m_2, j_1) = \sum_{j_2=2}^{f^{-1}(m_2, j_1, 2^{-\lambda})} f(m_1, j_1, j_2)$. For the failure probability until the third PGBF (i.e., the PGBF's failure probability), we have

$$\epsilon_1 \epsilon_2 \epsilon_3 = \sum_{j_1=2}^{\bar{\alpha}_1} \sum_{j_2=2}^{f^{-1}(m_2, j_1, 2^{-\lambda})} f(m_1, n, j_1) f(m_2, j_1, j_2) g(m_3, j_2). \tag{5}$$

Here $g(m_3, j_2) = \sum_{j_3=2}^{f^{-1}(m_3, j_2, 2^{-\lambda})} f(m_3, j_2, j_3)$.

Now we can generalize the failure probability of Multi-PGBF as

$$\epsilon = \sum_{j_1=2}^{\bar{\alpha}_1} \sum_{j_2=2}^{f^{-1}(m_2, j_1, 2^{-\lambda})} \cdots \sum_{j_{u-1}=2}^{f^{-1}(m_{u-1}, j_{u-2}, 2^{-\lambda})} f(m_1, n, j_1) f(m_2, j_1, j_2) \cdots f(m_{u-1}, j_{u-2}, j_{u-1}) g(m_u, j_{u-1}). \tag{6}$$

Here $g(m_u, j_{u-1}) = \sum_{j_u=2}^{f^{-1}(m_u, j_{u-1}, 2^{-\lambda})} f(m_u, j_{u-1}, j_u)$.

### C. Concrete instantiation

From the failure probability in equation (6), to instantiate Multi-PGBF, we need to know $f(m, n, j_1)$ and $f(m_{i+1}, j_i, j_{i+1})$ $(j_i \geq f^{-1}(m_{i+1}, j_i, 2^{-\lambda}) \geq j_{i+1})$. For very small sets $n = 2$ and $n = 3$, we have calculated $f(m, 2, 2)$, $f(m, 3, 2)$ and $f(m, 3, 3)$. However, it is non-trivial to compute the probability when $n$ is large. To enable the computation, we utilize experimental trials to get $f(m, n, \alpha)$ as previous works (e.g., [24], [6], [15]). In Fig. 5, we show $\gamma = -\log_2(\epsilon)$ for different set size $n$ and core size $\alpha$ (after $2^{25} \sim 2^{32}$ trials). In this figure, the dashed lines show the experimental results. Given $n$ and $m$ (after the critical expansion rate $\eta = m/n$), it is easy to find that $\gamma$ logarithmically increases with the increase of $\alpha$. Therefore, for each $n$, we utilize $\gamma = a_1 \log_2(0.5\alpha + 1) + a_2$ to fit and show the fitting function with the corresponding solid line. The identified functions and related parameters are shown in Table II. Then we can take

$$f(m, n, \alpha) = 2^{-(a_1 \log_2(0.5\alpha + 1) + a_2)} \tag{7}$$

for a PGBF.

The concrete parameter settings for Multi-PGBF is shown in Table III. In this table, we utilize $u = 3$ PGBFs because it

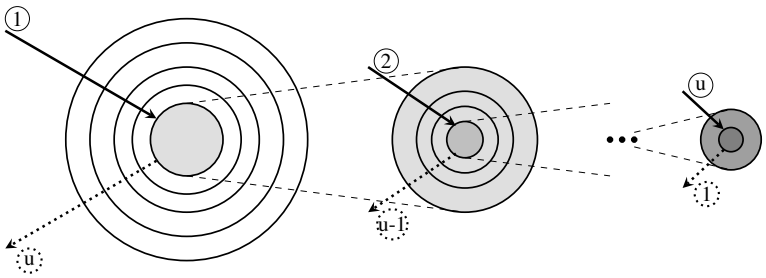



Fig. 3: The peeling and unpeeling process of Multi-PGBF. The solid arrow indicates that the peeling goes from the left onion to the right onion and from the outer peel to the inner peel. The dotted arrow shows that the unpeeling goes reversely.

**Inputs**: A key-value set $\{(x_i, v_i)\}_n$.
**Parameters:**

The number of PGBFs $u$. A hash table $D = \{D_1, D_2, \cdots, D_u\}$, where $D_i$ is a sub hash table with size $m_i$. A counting table $B = \{B_1, B_2, \cdots, B_u\}$, where $B_i$ is a sub counting table with size $m_i$, $B_{i,j}$ is the $j$th counter of $B_i$. Hash functions $\mathcal{H} = \{\mathcal{H}_1, \mathcal{H}_2, \cdots, \mathcal{H}_u\}$, where $\mathcal{H}_i = \{h_{i,1}, h_{i,2}, \cdots, h_{i,k_i}\}$ and $h_{i,j}$ is the $j$th hash function of the $i$th sub hash table.

**Algorithm:**

1) **Initiation.** Randomly sample a table $D$. Initialize all bins of $B$ as 0. For each item $x_i$, increasing its $k = k_1 + k_2 + \cdots + k_u$ corresponding counters $B_{j,z}$ ($\forall j \in [u], z \in [k_i]$) by 1. Prepare an empty stack $S$.
2) **Peeling key sets.** Denote the key set $\{x_i\}_n$ as $K_1$. From $i = 1$ to $i = u$,
   - recursively peeling $K_i$ to get a peeling stack $S_i'$ by using the counting table $B_i$ until there is a core $K_{i+1}$ (details are in Figure 1). If $K_{i+1}$ is empty, the peeling stops. For each triple $(x, v, j^*)$ in $S_i'$, inserting the PGBF index $i$ into it to be a triple $(x, v, i, j^*)$. Then appending $S_i'$ to $S$.

   If $|K_{u+1}| \neq 0$, the encoding fails; otherwise, the encoding continues.
3) **Assigning value shares.** The assigning process also goes recursively but in a reverse order. Specifically, for $(x, v, i, j^*) = S_n, S_{n-1}, \cdots, S_1$, we first get $x$'s XOR sum except the anchor bin as

$$\overline{D_{i,h_{i,j^*}(x)}} = \Big(\bigoplus_{\substack{\forall z \in [u]\setminus\{i\} \\ \forall j \in [k_i]}} D_{z,h_{z,j}(x)}\Big) \oplus \Big(\bigoplus_{\forall j \in [k_i]\setminus\{j^*\}} D_{i,h_{i,j}(x)}\Big),$$

   where $h_{i,j^*}(x)$ is its anchor location and $D_{i,h_{i,j^*}(x)}$ is the anchor bin. Then we assign its anchor bin as

$$D_{i,h_{i,j^*}(x)} = v \oplus \overline{D_{i,h_{i,j^*}(x)}}.$$

Fig. 4: The encoding algorithm for Multi-PGBF by recursively utilizing multiple PGBFs.

TABLE II: The fitting functions for different set size $n$ in PGBF. $\bar{\alpha}$ is the core size upper bound for $n$ and $\eta$.

| $n$ | $\eta$ | $\bar{\alpha}$ | $\gamma$ |
|---|---|---|---|
| $2^8$ | 1.69 | 47 | $8.0824\log_2(0.5\alpha+1)+0.21561$ |
| $2^9$ | 1.69 | 47 | $9.11627\log_2(0.5\alpha+1)-0.27941$ |
| $2^{10}$ | 1.39 | 27 | $10.75392\log_2(0.5\alpha+1)-0.79063$ |
| $2^{11}$ | 1.33 | 23 | $11.43862\log_2(0.5\alpha+1)-0.26131$ |
| $2^{12}$ | 1.3 | 18 | $12.72858\log_2(0.5\alpha+1)-0.22027$ |
| $2^{14}$ | 1.26 | 13 | $14.81106\log_2(0.5\alpha+1)-0.19490$ |
| $2^{16}$ | 1.24 | 9 | $17.75376\log_2(0.5\alpha+1)-0.72666$ |

is sufficient to amplify the failure probability. For the large set size $n = 2^{20}$, we set the core size of the first PGBF and the expansion rates of the second and third PGBFs as the ones of $n = 2^{16}$. The intuition is that if the following two PGBFs can handle a core with size 9 when $n = 2^{16}$, they can also handle it when $n = 2^{20}$. Due to the page limit, we leave the details about the instantiations in Appendix B.

### D. Further Analysis

After the above concrete instantiation, we know Multi-PGBF can achieve negligible failure probabilities in encoding (see Table II). Now, we further analyze the obliviousness and linearity of Multi-PGBF.

*Obliviousness*. In Multi-PGBF, the value of each key is encoded as random XOR shares in the hash table $D$. For the non-hashed bins in $D$, they are simply filled with random values. Therefore, it is easy to know that $D$ is uniform from the value distribution $\mathcal{V}^m$ if the encoded values are uniform. Due to the page limit, we put the formal proof of the obliviousness in Theorem C.1 in Appendix C.

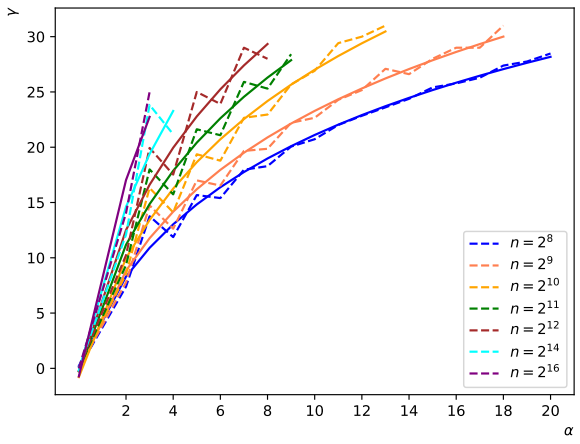



Fig. 5: Curve fitting for different set sizes $n$. The dashed line shows the experimental results and its corresponding solid line in the same color shows the fitting curve.

TABLE III: The parameter settings to achieve negligible failure probability for Multi-PGBF $\infty$. The core size upper bound for a PGBF $\propto$ is denoted as $\bar{\alpha}_i$. $\eta = \eta_1 + (\eta_2\bar{\alpha}_1 + \eta_3\bar{\alpha}_2)/n$ is the expansion rate of Multi-PGBF. $2^{-\gamma} = 2^{-\gamma_3}$ is the failure probability of Multi-PGBF and $\gamma_i = -\log 2(\epsilon_1 \cdots \epsilon_i)$.

| $n$ | $\propto$ | 1 | 2 | 3 | $\infty$ |
|---|---|---|---|---|---|
| $2^{12}$ | $\eta_i$ | 1.3 | 3.39 | 3.39 | $\eta = 1.33$ |
| | $\bar{\alpha}_i$ | 18 | 18 | 18 | |
| | $\gamma_i$ | 10.22 | 26.28 | 40.21 | $\gamma = 40.21$ |
| $2^{16}$ | $\eta_i$ | 1.24 | 4 | 4 | $\eta = 1.24$ |
| | $\bar{\alpha}_i$ | 9 | 9 | 9 | |
| | $\gamma_i$ | 14.04 | 28.65 | 40.33 | $\gamma = 40.33$ |
| $\geq 2^{20}$ | $\eta_i$ | 1.23 | 4 | 4 | $\eta = 1.23$ |
| | $\bar{\alpha}_i$ | 9 | 9 | 9 | |
| | $\gamma_i$ | $> 14.04$ | $> 28.65$ | $> 40.33$ | $\gamma > 40.33$ |

*Linearity*. To retrieve a query key's value, one only needs to compute the sum of $k$ XOR shares in the table $D$. Then it is easy to know Multi-PGBF is linear. Formally, we prove the linear property in Theorem IV.1.

**Theorem IV.1.** *Multi-PGBF in Fig. 4 is a linear OKVS.*

*Proof.* From the decoding equation 1, to decode a key $x$, one can easily find that $Decode(D, x) = \langle d(x), D\rangle$ and $d(x)$ is a bit vector with size $m$ in which $d(x)_{h_{i,j}(x)} = 1$ ($\forall i \in [u], j \in [k_i]$) and the other bits are 0s. Therefore, Multi-PGBF is a linear OKVS. Specifically, it is also easy to find that Multi-PGBF is also a binary OKVS. □

### *E. A cluster variant: C-Multi-PGBF*

For decoding the value of a key $x$, one can simply locate its cluster $C_{h(x)}$ and decode as a normal Multi-PGBF by using the table $D_{h(x)}$.

In Multi-PGBF, a key's value is stored as $k_1+k_2+k_3$ shares in the OKVS table $D$. However, from Table II, the subtable $D_1$ is much larger than $D_2$ and $D_3$. When $n$ is large (e.g., $n \geq 2^{20}$), the first $k_1$ shares of each key's value are distant (i.e., the distances between $h_{1,1}(x)$, $h_{1,2}(x)$, and $h_{1,k_1}(x)$ are large) and not cache-friendly, which makes the encoding less efficient than that when $n$ is small. In the OKVS RR [14], the authors propose a cluster variant C-RR that has fast encoding efficiency when the set size is large. The idea is to hash a large number of key-value pairs into small clusters. Then in each small cluster, the encoding can be cache-friendly and efficient. In this paper, we also take this clustering strategy and propose the cluster variant C-Multi-PGBF, which is shown in Figure 6. The cost of the cluster variant is that the expansion rate is increased such that the cluster size is sufficiently large to hold the hashed key-value pairs. In the instantiation, one needs to set an initiated cluster size $c$. Then one can get the number of clusters $n/c$ and compute the real cluster size $c'$ such that all the $n$ key-value pairs can be hashed into the clusters with an overwhelmingly large probability (i.e., $1-2^{-\lambda}$). In this paper, we set $c = 2^{16}$.

**Inputs**: A key-value set $\{(x_i, v_i)\}_n$.
**Parameters:**
A cluster size $c$. A hash function $h$.
**Algorithm:**
1) **Initiation.** Prepare $n/c$ empty clusters $C_1, C_2, \cdots, C_{n/c}$. Compute the maximum cluster size $c'$ for all clusters. For each cluster $C_i$, prepare a Multi-PGBF table $D_i$ for inserting $c'$ key-value pairs. For each $(x_i, v_i)$, hash it to a cluster $C_{h(x_i)}$.
2) **Multi-PGBF encoding.** For each cluster $C_i$, encode all its key-value pairs in $D_i$.

Fig. 6: The encoding algorithm of C-Multi-PGBF by clustering all key-value pairs into different clusters, each of which are encoded as Multi-PGBF.

## V. OKVS COMPARISONS

In this section, we compare Multi-PGBF and C-Multi-PGBF with the state-of-the-art OKVSs (i.e., PaXoS [8], 3H-GCT [3], RR/C-RR [14], and RB-OKVS [25]) in the encoding time, re-encoding time, and decoding time. In some applications (e.g., the multi-party PSI protocols [13]), it needs to encode with new values but with the same keys. Naively encoding can also work but brings much higher costs. Therefore, we include the re-encoding time in comparisons. For interpolation, it is too slow in both encoding and decoding. Therefore, we only present its performance. For GBF, it can be a special case of our PGBF in which there is only one peel. Multi-PGBF and C-Multi-PGBF are both much more efficient than GBF. For other OKVSs (except for GBF), we utilize their source codes and the concrete performance is shown in Table IV. The benchmark machine has Intel(R) Core(TM) i9-13900KF 24-Core Processor and 62G RAM. All experiments are run using a single thread.

TABLE IV: Expansion rate $\eta$, encoding time (ms), re-encoding time (ms), and decoding time (ms) comparisons for different OKVSs with different set sizes (i.e., $n \in \{2^{12}, 2^{16}, 2^{20}, 2^{24}\}$) in a single thread. For re-encoding, one encode with the same existing keys but with different values. Both the keys and values are with bit length 128. The best results are in bold. Cells with '-' indicate that it is too slow to get the data.

| **OKVS** | $\eta$ | $2^{12}$ | $2^{16}$ | $2^{20}$ | $2^{24}$ |
|---|---|---|---|---|---|
| **Encoding** | | | | | |
| Interpolation [8] | **1.000** | 60.013 | 1267.665 | 25688.689 | - |
| GBF [1] | 57.708 | 2.551 | 74.010 | 1409.667 | 33711.182 |
| PaXoS [2] | $2.551 \sim 2.405$ | 3.115 | 53.703 | 1634.334 | 31705.937 |
| 3H-GCT [3] | $1.389 \sim 1.301$ | 3.080 | 56.835 | 2123.723 | 43583.735 |
| RR [14] | $1.330 \sim 1.224$ | 0.820 | 13.173 | 414.426 | 11243.405 |
| C-RR [14] | $1.364 \sim 1.330$ | 0.802 | 11.169 | 178.047 | 2860.872 |
| RB-OKVS (0.03) [15] | $1.045 \sim 1.030$ | 5.318 | 46.395 | 795.647 | 13593.346 |
| RB-OKVS (0.1) [15] | $1.115 \sim 1.110$ | 1.109 | 16.396 | 294.544 | 4721.947 |
| Multi-PGBF | $1.331 \sim 1.230$ | **0.286** | **3.570** | 92.895 | 3041.623 |
| C-Multi-PGBF | $1.372 \sim 1.242$ | 0.319 | 4.031 | **62.928** | **1069.340** |
| **Re-Encoding** | | | | | |
| PaXoS [2] | $2.551 \sim 2.405$ | 2.431 | 41.067 | 1126.625 | 20835.724 |
| 3H-GCT [3] | $1.389 \sim 1.301$ | 1.889 | 31.903 | 885.223 | 17074.783 |
| Multi-PGBF | $1.331 \sim 1.230$ | **0.038** | 0.676 | 24.670 | 566.396 |
| C-Multi-PGBF | $1.372 \sim 1.242$ | 0.038 | **0.670** | **12.219** | **246.790** |
| **Decoding** | | | | | |
| Interpolation [8] | **1.000** | 32.135 | 748.226 | 16381.116 | - |
| GBF [1] | 57.708 | 1.014 | 43.336 | 884.266 | 20519.768 |
| PaXoS [2] | $2.551 \sim 2.405$ | 1.497 | 24.830 | 500.565 | 8783.944 |
| 3H-GCT [3] | $1.389 \sim 1.301$ | 1.267 | 21.227 | 447.763 | 8708.937 |
| RR [14] | $1.330 \sim 1.224$ | 0.125 | 1.444 | 28.451 | 609.198 |
| C-RR [14] | $1.364 \sim 1.330$ | 0.133 | 1.547 | 32.644 | 721.503 |
| RB-OKVS (0.03) [15] | $1.045 \sim 1.030$ | 2.288 | 37.750 | 666.886 | 11305.949 |
| RB-OKVS (0.1) [15] | $1.115 \sim 1.110$ | 0.894 | 14.631 | 264.482 | 4240.687 |
| Multi-PGBF | $1.331 \sim 1.230$ | 0.053 | 0.543 | **13.575** | **435.156** |
| C-Multi-PGBF | $1.372 \sim 1.242$ | **0.049** | **0.539** | 20.130 | 556.286 |

### *A. Comparisons with PaXoS/3H-GCT*

Both PaXoS [2] and 3H-GCT [3] also take the strategy of peeling and unpeeling. However, there are many differences between Multi-PGBF and PaXoS/3H-GCT. First, PaXoS/3H-GCT are constructed based on the cuckoo graph, while Multi-PGBF is based on CBF. Second, in the peeling, PaXoS/3H-GCT takes depth-first search (DFS) strategy, while Multi-PGBF takes a breadth-first search (BFS) strategy. Third, PaXoS/3H-GCT exploits the normal Gaussian elimination to solve the core. However, Multi-PGBF recursively utilizes PGBF. Fourth, Multi-PGBF takes fine-tuned expansion rates for different set sizes rather than fixed ones in PaXoS/3H-GCT, which are shown in Table IV. In our findings, setting fixed ones for all set sizes (i.e., $\eta = 2.4$ for PaXoS and $\eta = 1.3$ for 3H-GCT) may cause non-negligible failure probability (e.g., $n \leq 2^{11}$). In performance, Multi-PGBF is much more efficient. Specifically, in encoding, Multi-PGBF is respectively $10.42\times \sim 17.59\times$ and $10.76\times \sim 22.86\times$ as fast as PaXoS and 3H-GCT; in decoding, Multi-PGBF is respectively $20.19\times \sim 45.73\times$ and $20.01\times \sim 39.09\times$ as fast as them. For re-encoding, PaXos/3H-GCT and our Multi-PGBF/C-Mulit-PGBF only need to do the unpeeling to assigning the new values. Therefore, it can be cheaper than naively encoding again. In re-encoding, our Multi-PGBF is $36.79\times \sim 64.78\times$ and $30.15\times \sim 50.33\times$ as fast as PaXoS/3H-GCT. Due to the space limit, we leave more details in Appendix D.

### *B. Comparison with RR [14]*

RR [14] is encoded based on doing matrix-related operations (e.g., triangulation) to solve the system of linear equations, which involves both the keys and values. In Multi-PGBF, the encoding can be detached into the peeling and unpeeling phases. In the peeling, only the keys are needed. In the unpeeling, the values are used to finish the encoding. Therefore, with the same keys, if the encoding needs to be invoked for $\theta > 1$ times, one can only do the peeling once and the unpeeling for re-encoding $\theta$ times, thus saving encoding costs. In Multi-PGBF, the unpeeling time only accounts for $13.10\% \sim 26.56\%$ of the encoding time. Therefore, compared with naively encoding Multi-PGBF for $\theta$ times, this can save $73.44\%(1-1/\theta) \sim 86.90\%(1-1/\theta)$ computation costs. For example, in multi-party PSI [13], a party may encode for $\theta = 2$ times. For the encoding, one can save $36.72\% \sim 83.45\%$ encoding costs. Compared with Multi-PGBF, it is unknown how to have the detachment in RR [14]. Also, Multi-PGBF can easily be a KVS by directly inserting values into the anchor bins. It is unknown how to turn RR [14] to be a KVS. To decode a key's value, in average, one needs to collect $w + 41/2 \sim w + 43/2$ value shares in RR, where $w$ is the number of hash functions. For $w = 3$, the average number of shares is $23.5 \sim 24.5$, which is $2.61\times \sim 2.72\times$ as large as Multi-PGBF.

In performance, Multi-PGBF is more efficient than RR [14] in both encoding and decoding with close expansion rates (i.e.,

$< 0.5\%$). From Table IV, Multi-PGBF is respectively $2.86\times \sim 4.46\times$ and $1.40\times \sim 2.66\times$ as fast as RR in encoding and decoding. For the cluster variant C-RR, our clustered C-Multi-PGBF is respectively $2.51\times \sim 2.83\times$ and $1.30\times \sim 2.87\times$ as fast as it in encoding and decoding.

### C. Comparison with RB-OKVS [15]

As RR [14], RB-OKVS [15] is also based on doing matrix operations (e.g., row additions). Multi-PGBF also gains the encoding detachment advantage over RB-OKVS [15]. In RB-OKVS [15], the authors utilize the linear Gaussian solver [16] and introduce it to OKVS by finding proper parameters to achieve a negligible failure probability. To be fair, it is supposed to cite [16] in Table IV. For simplicity, in OKVS comparisons, we still cite RB-OKVS [15] in the table. In [16], to solve the linear system $M\mathbf{s} = \mathbf{v}$ in [16], in each row of the matrix $M$, one needs to generate a random band with width $\omega$ (i.e., consecutive $\omega$ random bits) from a random start column for each key. Since bits are consecutive, the Gaussian solver is cache-friendly and efficient in computation. However, $\omega$ needs to be set very large (i.e., $169 \sim 662$) from Figure 11 of [15]. To decode to get a key's value, on average, one needs to compute the sum of $169/2 \sim 662/2$ shares in the solved RB-OKVS table, which is much larger than $k = 9$ in Multi-PGBF. Because of this, RB-OKVS can have many disadvantages. For example, both RB-OKVS and Multi-PGBF can be a KVS by directly storing the value in its corresponding key's unique bin (i.e., the pivot bin in [16] and anchor bin in Multi-PGBF). Then after applying Multi-PGBF into the state-of-the-art PSI protocol KKRT16 [22], the sender only needs to return 9 OPRF values to the receiver; while the sender only needs to return $169 \sim 662$ OPRF values by using RB-OKVS (i.e., $18.8\times \sim 73.6\times$ as expensive).

In another example, a client wants to secretly query a key's value from a server's key-value database. The server can encode its key-value database into an OKVS table. Then the client can make the query by using private information retrieval (PIR) [26]. In PIR, a client can query an entry of the server's database by inputting the entry's index without disclosing this index to the server, thus keeping privacy. By utilizing RB-OKVS, the client needs to make the query for $169 \sim 662$ indices, which is also $18.8\times \sim 73.6\times$ as expensive as that by using Multi-PGBF. A concrete example of collecting fewer value shares to gain performance advantage in PSI is shown in circuit PSI [27]. In circuit PSI [27], the client needs to secretly query the value shares from the server by using PIR. By using RB-OKVS (0.1), the band width is $\omega = 206$ when the server set size is $n_y = 2^{20}$. Then the client with size $n_x$ needs to query for $\omega n_x$ value shares. By using Multi-PGBF, as RR [14] (that is used in [27]), it only needs to query $k_1 n_x = 3n_x$ shares, which is $68.67\times$ as small as that by using RB-OKVS (0.1). It is noted that the subtables $D_2$ and $D_3$ of Multi-PGBF are both small and can be directly sent to the client without breaking the security. This is the same as sending the dense table (i.e., the subtable in RR to solve the 2-core) from the server to the client in [27]. Then there is no need to do PIR to query the last $k_2 + k_3$ value shares.

In RB-OKVS, the authors set different expansion rates. Setting 0.03 and 0.1 indicate that the expansion rates are respectively $1.03$ and $1.1$. We test the minimum 0.03 and the maximum 0.1 from their Figure 11. For encoding, from Table IV, Multi-PGBF is respectively $4.47\times \sim 18.57\times$ and $1.55\times \sim 4.59\times$ as fast as RB-OKVS (0.03) and RB-OKVS (0.1). If it needs to re-encode once, the advantage of Multi-PGBF becomes more apparent. For decoding, Multi-PGBF is respectively $25.98\times \sim 69.52\times$ and $9.75\times \sim 26.94\times$ as fast as RB-OKVS (0.03) and RB-OKVS (0.1).

## VI. Applications: Private set intersection

In this section, we apply our Multi-PGBF and C-Multi-PGBF into PSI, in which the parties securely get the intersection without revealing other information. Specifically, we first replace PaXoS [2] with Multi-PGBF and C-Multi-PGBF in the state-of-the-art two-party PSI protocol [10]. Then we replace 3H-GCT [3] with Multi-PGBF and C-Multi-PGBF in a multi-party PSI protocol *K-Star* [13].

### A. Two-party PSI

**Parameters:** The set size of the receiver and the sender are respectively $n_x$ and $n_y$.
**Inputs:** The receiver's set $X$. The sender's set $Y$.
**Outputs:** The receiver gets $X \cap Y$. The sender gets nothing.

Fig. 7: The ideal functionality of two-party PSI.

The ideal functionality of two-party PSI is shown in Figure 7. In two-party PSI, a party who inputs $X$ gets the intersection $X \cap Y$, while the other party who inputs $Y$ learns nothing.

*1) RS21 [10] with different OKVSs:* In RS21 [10], the authors combined PaXoS with a primitive vector oblivious linear evaluations (VOLE) [25] to realize two-party PSI. In their work, they expected a more efficient OKVS such that the performance of their protocol can be improved. Our Multi-PGBF and C-Multi-PGBF can exactly replace PaXoS to gain better performance.

We compared the semi-honest PSI protocol in [10] with different OKVSs in Table V. From this Table, the protocol with Multi-PGBF and C-Multi-PGBF win in the most of settings. Generally, by using Multi-PGBF and C-Multi-PGBF can run the fastest when the bandwidth is high (e.g., $\geq 100$ Mbps). Specifically, in LAN, the protocol with C-RR [14] runs faster than the protocol with other OKVSs (except for ours). In comparison with it, the protocol with Multi-PGBF and C-Multi-PGBF can run $9.53\%$ and $2.69\%$ faster when $n = 2^{12}$. The reason is that the set size is small, and our OKVS encoding and decoding advantages are in microseconds (more details are shown Table IV). Then the other costs in the PSI protocol dominate. When $n \geq 2^{12}$, the protocol with Multi-PGBF and C-Multi-PGBF can run $0.94\% \sim 11.96\%$

TABLE V: Performance comparisons between the semi-honest PSI protocol [10] with different OKVSs. 'RB' is short for RB-OKVS. For LAN, the bandwidth is 1 Gbps and the round trip latency is 0.06 ms. For WAN, the bandwidth is respectively 100 Mbps, 10 Mbps, and 1 Mbps, with 80 ms latency. The best results are in bold.

| $n$ | Protocol | Communication | | | Total running time (s) | | | |
|---|---|---|---|---|---|---|---|---|
| | | $R \to S$ | $S \to R$ | Total | LAN | 100 Mbps | 10 Mbps | 1 Mbps |
| $2^{12}$ | RR [14] | 0.350 | 0.077 | 0.427 | 0.037 | 1.152 | 1.413 | 4.578 |
| | C-RR [14] | 0.350 | 0.077 | 0.427 | 0.037 | 1.151 | 1.410 | 4.576 |
| | RB(0.03) [15] | 0.333 | 0.077 | **0.410** | 0.045 | 1.163 | 1.412 | **4.445** |
| | RB(0.1) [15] | 0.337 | 0.077 | 0.414 | 0.039 | 1.154 | **1.404** | 4.470 |
| | Multi-PGBF | 0.350 | 0.077 | 0.427 | **0.034** | **1.148** | 1.408 | 4.574 |
| | C-Multi-PGBF | 0.350 | 0.077 | 0.427 | 0.036 | 1.149 | 1.409 | 4.576 |
| $2^{16}$ | RR [14] | 1.515 | 0.624 | 2.139 | 0.103 | 1.684 | 3.143 | 19.227 |
| | C-RR [14] | 1.617 | 0.624 | 2.241 | 0.101 | 1.688 | 3.226 | 20.086 |
| | RB(0.03) [15] | 1.302 | 0.624 | **1.926** | 0.152 | 1.733 | 3.033 | **17.513** |
| | RB(0.1) [15] | 1.372 | 0.624 | 1.996 | 0.102 | 1.683 | **3.031** | 18.046 |
| | Multi-PGBF | 1.513 | 0.624 | 2.137 | **0.100** | 1.673 | 3.128 | 19.248 |
| | C-Multi-PGBF | 1.513 | 0.624 | 2.137 | 0.107 | **1.672** | 3.130 | 19.201 |
| $2^{20}$ | RR [14] | 19.915 | 10.077 | 29.992 | 1.594 | 5.816 | 27.739 | 255.221 |
| | C-RR [14] | 22.037 | 10.077 | 32.114 | 1.313 | 5.724 | 29.260 | 272.790 |
| | RB(0.03) [15] | 16.756 | 10.077 | **26.833** | 2.485 | 6.298 | **25.920** | **229.413** |
| | RB(0.1) [15] | 17.876 | 10.077 | 27.953 | 1.559 | 5.522 | 25.999 | 237.987 |
| | Multi-PGBF | 19.957 | 10.077 | 30.034 | **1.156** | **5.356** | 27.329 | 255.116 |
| | C-Multi-PGBF | 22.044 | 10.077 | 32.121 | 1.188 | 5.589 | 29.137 | 272.729 |
| $2^{24}$ | RR [14] | 313.611 | 176.093 | 489.704 | 33.064 | 77.611 | 446.120 | - |
| | C-RR [14] | 349.404 | 176.093 | 525.497 | 25.154 | 72.436 | 466.802 | - |
| | RB(0.03) [15] | 263.960 | 176.093 | **440.053** | 45.152 | 85.381 | **415.276** | - |
| | RB(0.1) [15] | 281.880 | 176.093 | 457.973 | 29.248 | 70.976 | 416.691 | - |
| | Multi-PGBF | 315.161 | 176.093 | 491.254 | 24.192 | **69.032** | 438.739 | - |
| | C-Multi-PGBF | 351.584 | 176.093 | 527.676 | **23.171** | 70.772 | 467.421 | - |

and $7.88\% \sim 9.53\%$ faster, respectively. Because RB-OKVS has a lower expansion rate, the protocol with RB-OKVS(0.03) and RB-OKVS(0.1) can have lower communication costs. Therefore, when the bandwidth is low (e.g., $\geq 10$ Mbps), their protocol can run faster. For WAN with 1 Mbps bandwidth, the protocol with RB(0.03) can respectively run $2.81\% \sim 10.07\%$ and $2.87\% \sim 15.88\%$ faster.

*2) Insecurity of [10]:* In addition to RS21 [10], we find another random oblivious transfer (ROT) based two-party PSI protocol [18]. In their protocol, they also utilize OKVS. However, their protocol is insecure. Their protocol is highly similar to a previous PSI protocol [28] that also utilizes ROT and OKVS. To utilize ROT, the authors [28] rely on a correlation-robust security assumption [22], as previous OT-based PSI protocols [22], [4], [5], [8], [6], [2]. Specifically, a hash function $H$ is $\kappa$-Hamming correlation robust if, for any two bit string vectors $a_1, a_2, \cdots, a_n$ and $b_1, b_2, \cdots, b_n$ ($a_i, b_i \in \{0,1\}^\omega$ and the Hamming distance of $b_i$, $\|b_i\| \geq \kappa$), the following distribution, introduced by sampling a random bit string $s \in \{0,1\}^\omega$,

$$H(a_1 \oplus (s \wedge b_1), H(a_2 \oplus (s \wedge b_2), \cdots, H(a_n \oplus (s \wedge b_n)$$

is pseudorandom. Here '$\oplus$' and '$\wedge$' are bitwise AND and XOR, respectively. The core condition in this assumption is that the Hamming weight $\|b_i\| \geq \kappa$. Then $s \wedge b_i$ can select $\geq \kappa$ bits from the random $s$ to ensure the pseudorandomness. If $b_i = 0$, it has $H(a_i \oplus (s \wedge b_i) = H(a_i \oplus (s \wedge 0)) = H(a_i)$; otherwise, it gets high probability $s \wedge b_i$ is pseudorandom and $H(a_i \oplus (s \wedge b_i)$ is pseudorandom. Correspondingly, $\omega$ should be sufficiently long to ensure the Hamming weight $\geq \kappa$. In [28], it sets $\omega \geq 428$. However, in [18], the author neglects this assumption and simply sets $\omega = \lambda + \log(n_x n_y) \leq 88$, which is far from sufficient to ensure the Hamming weight $\geq \kappa$, thus insecure. In their paper, because $\omega$ is short, their protocol can have better performance than [10], which is actually not true. In addition to a two-party PSI protocol, their protocols for PSI variants (e.g., circuit PSI) are also similar to previous protocols [29], [10]. Since the basic PSI is insecure, these protocols are also insecure, and their reported performance is incorrect. More analyses are shown in the Appendix F.

**Parameters:** The number of parties $b$, the number of corrupted parties $t < b$, and the set size $b$.
**Inputs:** Each party $P_i$ ($i \in [b]$) inputs a set $X_i = \{x_j^i\}_{\forall j \in [n]}$.
**Outputs:** $P_1$ gets the intersection $I = \bigcap_{i=1}^{n} X_i$. The other parties receive $\perp$.

Fig. 8: The ideal functionality of multi-party PSI.

### B. Multi-party PSI

The ideal functionality of multi-party PSI is shown in Figure 8. In multi-party PSI, each party inputs its set and one party gets the intersection, while the other parties learn nothing.

*1) O-Ring and K-Star [13]:* Wu et al. [13] proposed two efficient multi-party PSI protocols *O-Ring* and *K-Star*, both of which use 3H-GCT [3] because its encoding can be detached. Among the $b$ parties, their protocols can be secure against $t < b$ arbitrary colluding parties. Since both protocols have

the same communication costs but O-Ring runs no faster than K-Star, we only replace 3H-GCT with Multi-PGBF and C-Multi-PGBF in K-Star. After experiments in the same settings as [13], we find that Multi-PGBF/C-Multi-PGBF can improve the running time by $1.15\times \sim 5.04\times$/$1.18\times \sim 5.03\times$ and $1.10\times \sim 1.99\times$/$1.02\times \sim 1.87\times$ in the LAN and WAN, respectively. Also, Multi-PGBF/C-Multi-PGBF can reduce $0.95\% \sim 10.56\%$/$0.95\% \sim 4.35\%$ communication costs. We also compare K-Star with other OKVSs (i.e., RR/C-RR [14] and RB-OKVS(0.03)/RB-OKVS(1.0) [15]) as in two-party PSI. The results show that K-Star with Multi-PGBF/C-Multi-PGBF runs the fastest in the most of settings in both LAN and WAN. K-Star with RB-OKVS(0.03) [15]) has the lowest communication costs due to the smallest expansion rates. More detailed comparisons are shown in Table VIII of Appendix G.

*2) O-Ring and K-Star can support unbalanced scenarios:* Tu et al. [19] proposed an unbalanced multi-party PSI protocol, in which some parties have much smaller sets than the others. In their protocol, they claimed O-Ring and K-Star did not support unbalanced input scenarios. However, after reviewing O-Ring and K-Star, we find their statement is not actually true. The possible reason for this misunderstanding is that in O-Ring and K-Star, it simply configures the same set sizes for all parties as previous multi-party PSI protocols (e.g., [7], [30]). By simply configuring a set size for each party, it can easily support unbalanced cases.

O-Ring and K-Star can support unbalanced settings because they are oblivious pseodrorandom function (OPRF) based. In the protocol, each one of $t$ parties $P_{n-t+1}, P_{n-t+2}, \cdots, P_n$ acts as an OPRF sender to interact with each one of the other $n-1$ parties who act as OPRF receivers. For OPRF, it is easy to support unbalanced cases (e.g., [10], [14]); then O-Ring and K-Star can also support. In O-Ring and K-Star, the authors used the OPRF protocol [10] that has communication complexity $O(n_x)$, where $n_x$ is the receiver's set size. To optimize communication costs in O-Ring and K-Star in unbalanced cases, it is supposed to select parties with small set sizes as the OPRF receivers. Therefore, we modify the protocols of O-Ring and K-Star [13] by sorting the parties based on their set sizes in ascending order (i.e., $|X_i| \leq |X_{i+1}|$, $i \in [1, n-1]$). Due to the page limit, the full protocols of the modified O-Ring and K-Star are shown in Appendix H-A.

In addition, there is another misunderstanding. In [19], to compare the performance of O-Ring and K-Star, the authors pad each small set to a set with the largest set size in O-Ring and K-Star. However, this has two issues: unfair comparisons, and insecurity when the input item domain is not large. We leave more detailed discussions in Appendix H-B.

After the above analysis, we evaluate the performance of the unbalanced multi-party PSI [19] and the above modified K-Star with Multi-PGBF and C-Multi-PGBF in Table VI. The results show that the modified K-Star can run the fastest when the network bandwidth is $\geq 10$ Mbps. Specifically, K-Star with Multi-PGBF and Cluster-Multi-PGBF is $1.09\times \sim 107.18\times$ and $1.07\times \sim 116.17\times$ as fast as [19]. However, the protocol [19] is generally $3.37\times \sim 93.67\times$ and $3.37\times \sim 104.16\times$ as cheap as K-Star with Multi-PGBF and Cluster-Multi-PGBF in communication costs, respectively. When the bandwidth is very low (i.e., 1 Mbps), the protocol [19] can run $1.41\times \sim 21.45\times$ and $1.44 \times 23.87\times$ as fast.

TABLE VI: Communication and total running time comparisons for unbalanced MPSI under different settings. $N$ and $M$ are the number of parties with large and small sets, respectively. $n_l$ and $n_s$ are the large set size and small set size, respectively. 'Comm.' is short for 'Communication'. In protocol, 'Mul.' and 'Clu.' are short for Multi-PGBF and C-Multi-PGBF, respectively.

| $(N, M)$ | $(n_l, n_s)$ | Protocol | Comm. (MB) | Total running time (s) | | | |
|---|---|---|---|---|---|---|---|
| | | | | 10 Gbps | 100 Mbps | 10 Mbps | 1 Mbps |
| $(1,2)$ | $(2^{22}, 2^{10})$ | MinBucket [19] | – | – | – | – | – |
| | | K-Star (Mul.) | **114.52** | 3.65 | **20.50** | **105.58** | – |
| | | K-Star (Clu.) | 127.50 | **3.48** | 21.87 | 116.81 | – |
| | $(2^{18}, 2^{10})$ | MinBucket [19] | **2.49** | 4.03 | 10.40 | 12.77 | **38.31** |
| | | K-Star (Mul.) | 8.47 | **0.23** | **6.26** | **11.71** | 77.76 |
| | | K-Star (Clu.) | 8.70 | 0.24 | 6.29 | 11.93 | 79.97 |
| | $(2^{14}, 2^{10})$ | MinBucket [19] | 2.49 | 1.94 | 8.69 | 10.77 | 36.55 |
| | | K-Star (Mul.) | **1.79** | **0.04** | **4.64** | 5.71 | **21.66** |
| | | K-Star (Clu.) | **1.79** | **0.04** | **4.64** | **5.65** | 21.70 |
| $(1,5)$ | $(2^{22}, 2^{10})$ | MinBucket [19] | – | – | – | – | – |
| | | K-Star (Mul.) | 357.73 | 5.37 | **42.22** | 308.78 | 3021.68 |
| | | K-Star (Clu.) | 397.77 | **5.28** | 44.91 | 343.22 | 3362.62 |
| | $(2^{18}, 2^{10})$ | MinBucket [19] | **2.88** | 5.73 | 17.55 | **21.62** | **69.65** |
| | | K-Star (Mul.) | 30.34 | **0.31** | **8.36** | 30.21 | 266.38 |
| | | K-Star (Clu.) | 31.03 | 0.32 | 8.45 | 30.91 | 271.96 |
| | $(2^{14}, 2^{10})$ | MinBucket [19] | **2.88** | 3.72 | 15.84 | 19.68 | **67.68** |
| | | K-Star (Mul.) | 9.69 | 0.06 | **5.84** | 12.76 | 95.56 |
| | | K-Star (Clu.) | 9.69 | **0.06** | **5.84** | **12.55** | 97.20 |
| $(5,5)$ | $(2^{22}, 2^{10})$ | MinBucket [19] | – | – | – | – | – |
| | | K-Star (Mul.) | **3349.08** | **24.32** | **316.34** | – | – |
| | | K-Star (Clu.) | 3729.79 | 25.09 | 338.19 | – | – |
| | $(2^{18}, 2^{10})$ | MinBucket [19] | **11.51** | 45.49 | 97.45 | – | – |
| | | K-Star (Mul.) | 236.55 | **1.17** | **25.54** | **206.29** | – |
| | | K-Star (Clu.) | 243.14 | 1.23 | 26.20 | 211.23 | – |
| | $(2^{14}, 2^{10})$ | MinBucket [19] | **11.51** | 16.61 | 69.85 | 92.69 | 362.91 |
| | | K-Star (Mul.) | 40.23 | 0.16 | 8.67 | 39.62 | **356.55** |
| | | K-Star (Clu.) | 40.23 | **0.14** | **8.66** | **39.46** | 358.09 |
| $(5,10)$ | $(2^{22}, 2^{10})$ | MinBucket [19] | – | – | – | – | – |
| | | K-Star (Mul.) | **5355.83** | **35.98** | **494.28** | – | – |
| | | K-Star (Clu.) | 5962.14 | 38.17 | 537.53 | – | – |
| | $(2^{18}, 2^{10})$ | MinBucket [19] | **12.16** | 48.72 | – | – | – |
| | | K-Star (Mul.) | 398.81 | **1.89** | **44.20** | **349.60** | – |
| | | K-Star (Clu.) | 409.31 | 1.94 | 44.71 | 357.88 | – |
| | $(2^{14}, 2^{10})$ | MinBucket [19] | **12.15** | 19.53 | 84.40 | 111.49 | **418.96** |
| | | K-Star (Mul.) | 86.13 | **0.25** | **16.69** | **84.01** | 768.11 |
| | | K-Star (Clu.) | 86.13 | 0.35 | 16.69 | 85.15 | 765.46 |
| $(10,10)$ | $(2^{22}, 2^{10})$ | MinBucket [19] | – | – | – | – | – |
| | | K-Star (Mul.) | **14639.11** | 212.12 | **1319.75** | – | – |
| | | K-Star (Clu.) | 16302.93 | **201.49** | 1461.94 | – | – |
| | $(2^{18}, 2^{10})$ | MinBucket [19] | **22.95** | 158.88 | 324.27 | **405.33** | – |
| | | K-Star (Mul.) | 1036.10 | **5.45** | **103.47** | 894.39 | – |
| | | K-Star (Clu.) | 1064.90 | 6.18 | 106.36 | 916.88 | – |
| | $(2^{14}, 2^{10})$ | MinBucket [19] | **22.95** | 52.58 | 222.68 | 303.47 | **1223.28** |
| | | K-Star (Mul.) | 178.00 | **1.54** | **30.66** | – | – |
| | | K-Star (Clu.) | 178.00 | 2.33 | 30.81 | **167.76** | – |

## VII. Conclusion and Future Work

In this paper, we present two elegant and efficient OKVSs: Multi-PGBF and C-Multi-PGBF. Multi-PGBF gains the highest encoding efficiency when the set size is not large. C-Multi-PGBF is a more cache-friendly variant and can encode the fastest when the set size is large. When it needs to encode multiple times with the same keys, Multi-PGBF and C-Multi-PGBF have the encoding detachment advantage. Also, Multi-PGBF can be a KVS without extra costs. After applying Multi-PGBF and C-Multi-PGBF to the state-of-the-art two-party and multi-party PSI protocols, the protocols can run the fastest in high bandwidth.

We find that all existing OKVSs only support encoding key-value pairs statically. However, as a key-value data structure, it is also supposed to support dynamic operations in real-world applications. For example, after encoding, one may need to delete/update/add a key-value pair. Multi-PGBF and C-Multi-PGBF can easily support these operations. For example, to

update a key's value, one can simply assign the updated final share in its anchor bin and update the keys' values in its outer peels recursively. In the future, we will extend Multi-PGBF and C-Multi-PGBF to support these dynamic operations.

## REFERENCES


[1] C. Dong, L. Chen, and Z. Wen, "When private set intersection meets big data: an efficient and scalable protocol," in *CCS 2013*, A. Sadeghi, V. D. Gligor, and M. Yung, Eds. ACM, 2013, pp. 789–800.

[2] B. Pinkas, M. Rosulek, N. Trieu, and A. Yanai, "PSI from paxos: Fast, malicious private set intersection," in *EUROCRYPT 2020*, ser. Lecture Notes in Computer Science, A. Canteaut and Y. Ishai, Eds., vol. 12106. Springer, 2020, pp. 739–767.

[3] G. Garimella, B. Pinkas, M. Rosulek, N. Trieu, and A. Yanai, "Oblivious key-value stores and amplification for private set intersection," in *CRYPTO 2021*, ser. Lecture Notes in Computer Science, T. Malkin and C. Peikert, Eds., vol. 12826. Springer, 2021, pp. 395–425.

[4] B. Pinkas, T. Schneider, and M. Zohner, "Faster private set intersection based on OT extension," in *USENIX Security 2014*, K. Fu and J. Jung, Eds. USENIX Association, 2014, pp. 797–812.

[5] B. Pinkas, T. Schneider, G. Segev, and M. Zohner, "Phasing: Private set intersection using permutation-based hashing," in *USENIX Security 2015*, J. Jung and T. Holz, Eds. USENIX Association, 2015, pp. 515–530.

[6] B. Pinkas, T. Schneider, and M. Zohner, "Scalable private set intersection based on OT extension," *ACM Trans. Priv. Secur.*, vol. 21, no. 2, pp. 7:1–7:35, 2018.

[7] V. Kolesnikov, N. Matania, B. Pinkas, M. Rosulek, and N. Trieu, "Practical multi-party private set intersection from symmetric-key techniques," in *CCS 2017*, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. ACM, 2017, pp. 1257–1272.

[8] B. Pinkas, M. Rosulek, N. Trieu, and A. Yanai, "Spot-light: Lightweight private set intersection from sparse OT extension," in *CRYPTO 2019*, ser. Lecture Notes in Computer Science, A. Boldyreva and D. Micciancio, Eds., vol. 11694. Springer, 2019, pp. 401–431.

[9] M. Chase and P. Miao, "Private set intersection in the internet setting from lightweight oblivious PRF," in *CRYPTO 2020*, ser. Lecture Notes in Computer Science, D. Micciancio and T. Ristenpart, Eds., vol. 12172. Springer, 2020, pp. 34–63.

[10] P. Rindal and P. Schoppmann, "VOLE-PSI: fast OPRF and circuit-psi from vector-ole," in *EUROCRYPT 2021*, ser. Lecture Notes in Computer Science, A. Canteaut and F. Standaert, Eds., vol. 12697. Springer, 2021, pp. 901–930.

[11] M. Rosulek and N. Trieu, "Compact and malicious private set intersection for small sets," in *CCS 2021*, Y. Kim, J. Kim, G. Vigna, and E. Shi, Eds. ACM, 2021, pp. 1166–1181.

[12] N. Chandran, N. Dasgupta, D. Gupta, S. L. B. Obbattu, S. Sekar, and A. Shah, "Efficient linear multiparty PSI and extensions to circuit/quorum PSI," in *CCS '21*, Y. Kim, J. Kim, G. Vigna, and E. Shi, Eds. ACM, 2021, pp. 1182–1204. [Online]. Available: https://doi.org/10.1145/3460120.3484591

[13] M. Wu, T. H. Yuen, and K. Y. Chan, "O-ring and k-star: Efficient multiparty private set intersection," in *USENIX Security 2024*, D. Balzarotti and W. Xu, Eds. USENIX Association, 2024. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/wu-mingli

[14] S. Raghuraman and P. Rindal, "Blazing fast PSI from improved OKVS and subfield VOLE," in *CCS 2022*, H. Yin, A. Stavrou, C. Cremers, and E. Shi, Eds. ACM, 2022, pp. 2505–2517.

[15] A. Bienstock, S. Patel, J. Y. Seo, and K. Yeo, "Near-optimal oblivious key-value stores for efficient psi, PSU and volume-hiding multi-maps," in *USENIX Security 2023*, J. A. Calandrino and C. Troncoso, Eds. USENIX Association, 2023. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/bienstock

[16] M. Dietzfelbinger and S. Walzer, "Efficient gauss elimination for near-quadratic matrices with one short random block per row, with applications," in *ESA 2019*, ser. LIPIcs, M. A. Bender, O. Svensson, and G. Herman, Eds., vol. 144. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, pp. 39:1–39:18. [Online]. Available: https://doi.org/10.4230/LIPIcs.ESA.2019.39

[17] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM Transactions on Networking*, vol. 8, no. 3, pp. 281–293, 2000.

[18] G. Ling, P. Tang, J. Shan, L. Xiang, and W. Qiu, "Faster than ever: A new lightweight private set intersection and its variants," in *33rd Annual Network and Distributed System Security Symposium, NDSS 2026, San Diego, California, USA, February 23-27, 2026*. The Internet Society, 2026. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/faster-than-ever-a-new-lightweight-private-set-intersection-and-its-variants/

[19] B. Tu, B. Zhu, Y. Cao, and Y. Chen, "Minbucket MPSI: breaking the max-size bottleneck in multi-party private set intersection," in *33rd Annual Network and Distributed System Security Symposium, NDSS 2026, San Diego, California, USA, February 23-27, 2026*. The Internet Society, 2026. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/minbucket-mpsi-breaking-the-max-size-bottleneck-in-multi-party-private-set-intersection/

[20] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[21] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[22] V. Kolesnikov, R. Kumaresan, M. Rosulek, and N. Trieu, "Efficient batched oblivious PRF with applications to private set intersection," in *CCS 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. ACM, 2016, pp. 818–829.

[23] H. Chen, K. Laine, and P. Rindal, "Fast private set intersection from homomorphic encryption," in *CCS 2017*, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. ACM, 2017, pp. 1243–1255.

[24] H. Chen, Z. Huang, K. Laine, and P. Rindal, "Labeled PSI from fully homomorphic encryption with malicious security," in *CCS 2018*, D. Lie, M. Mannan, M. Backes, and X. Wang, Eds. ACM, 2018, pp. 1223–1237. [Online]. Available: https://doi.org/10.1145/3243734.3243836

[25] S. Raghuraman, P. Rindal, and T. Tanguy, "Expand-convolute codes for pseudorandom correlation generators from LPN," in *CRYPTO 2023*, ser. Lecture Notes in Computer Science, H. Handschuh and A. Lysyanskaya, Eds., vol. 14084. Springer, 2023, pp. 602–632. [Online]. Available: https://doi.org/10.1007/978-3-031-38551-3_19

[26] E. Kushilevitz and R. Ostrovsky, "Replication is NOT needed: SINGLE database, computationally-private information retrieval," in *FOCS '97*. IEEE Computer Society, 1997, pp. 364–373. [Online]. Available: https://doi.org/10.1109/SFCS.1997.646125

[27] M. Hao, W. Liu, L. Peng, H. Li, C. Zhang, H. Chen, and T. Zhang, "Unbalanced circuit-psi from oblivious key-value retrieval," in *USENIX Security 2024*, D. Balzarotti and W. Xu, Eds. USENIX Association, 2024. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/hao-meng-unbalanced

[28] M. Wu, T. H. Yuen, and S. Yiu, "Efficient private set intersection by utilizing oblivious transfer extension," in *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security, ASIA CCS 2025, Hanoi, Vietnam, August 25-29, 2025*. ACM, 2025, pp. 104–117. [Online]. Available: https://doi.org/10.1145/3708821.3733886

[29] 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*, ser. Lecture Notes in Computer Science, Y. Ishai and V. Rijmen, Eds., vol. 11478. Springer, 2019, pp. 122–153. [Online]. Available: https://doi.org/10.1007/978-3-030-17659-4_5

[30] O. Nevo, N. Trieu, and A. Yanai, "Simple, fast malicious multiparty private set intersection," in *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Y. Kim, J. Kim, G. Vigna, and E. Shi, Eds. ACM, 2021, pp. 1151–1165. [Online]. Available: https://doi.org/10.1145/3460120.3484772


## APPENDIX A
## THE THEORETICAL BOUNDS OF PGBF'S PEELABILITY

In this section, we theoretically prove the upper bound and lower bound of PGBF's peelability.

### A. Upper bound of peelability

Cuckoo hashing [20] is another probabilistic hash-based data structure that can store $n$ key-value pairs. Similar to

the Bloom filter, cuckoo hashing utilizes $k$ hash functions $\mathcal{H} = \{h_1(\cdot), h_2(\cdot), \cdots, h_k(\cdot)\}$ and a hash table $T$ of size $m$. The difference is that cuckoo hashing aims to directly store each key's value $v_i$ into one of the $k$ bins in $T$ rather than store '1's in all the $k$ bins. We first notice that the structure of cuckoo hashing and PGBF is similar. Both of them utilize $k$ uniform hash functions to find a unique bin in a hash table for each key. Also, cuckoo hashing has been widely studied [5], [6], [7], [23]. Therefore, studying the similarity between cuckoo hashing and PGBF is helpful for us to find the peelability of the PGBF.

**Theorem A.1.** *Given the same $m$, $n$, and $k$, the failure probability of a single PGBF is no less than the failure probability of a cuckoo hashing.*

*Proof.* It is noted that a single PGBF needs to consider all its $k$ hashed bins while a cuckoo hashing only needs to consider one. For the same hash table with size $m$, $n$ key-value pairs, and $k$ uniform hash functions, if a PGBF can find all the $n$ keys' anchors, then a cuckoo hashing must find all the $n$ items' inserted locations by simply taking the anchor locations; while even if a cuckoo hashing can find all the $n$ keys' inserted locations, a PGBF may not find all the $n$ keys' anchors. We can also easily observe that by a simple example. The two keys $x$ and $y$ collide by $h_1(x) = h_1(y)$ and $h_2(x) = h_2(y)$, a cuckoo hashing can assign $x$'s value $v_x$ into $h_1(x)$ and $y$'s value $v_y$ into $h_2(y)$; however, the probability that a PGBF can finish assigning $v_x = B_{h_1(x)} \oplus B_{h_2(x)}$ and $v_y = B_{h_1(y)} \oplus B_{h_2(y)}$ is $2^{-\omega}$, where $\omega$ is the bit length of the value. □

**Lemma A.2.** *Given the same $n$, $k$, and failure probability $\epsilon$, the expansion rate of a single PGBF is no less than that of the cuckoo hashing.*

*Proof.* From Theorem A.1, we know for the same expansion rate $\eta = m/n$ and $k$, the failure probability of a cuckoo hashing is no greater than that of a PGBF. Therefore, given $n$ and $k$, to achieve the same failure probability, a PGBF has to increase the table size from $m = \eta n$ to $m' = \eta' n$. Then its expansion rate $\eta'$ will be greater than $\eta$. □

Theorem A.1 is significant to deduce the lower bound of the expansion rate of the PGBF for the same failure probability, which is the one of the cuckoo hashing, as is shown in Lemma A.2. It is well known that getting the expansion rate of a cuckoo hashing is non-trivial. Since there are no specific theoretical bases, many researchers do empirical evaluations to get cuckoo hashing's failure probabilities [5], [6], [7], [23]. The common practice is that they first do some limited iterations of experiments to get the failure probability and then extrapolate the results to get a lower failure probability (e.g., $2^{-\lambda} = 2^{-40}$). For example, to determine the concrete failure probability, Pinkas et. al. [6] performed $2^{30}$ iterations of cuckoo hashing by inserting 1024 items for $k$. Since there are many research outcomes about cuckoo hashing, we can use them to find the failure probability lower bound of the PGBF. Then we can get valid parameters in our PGBF.

### *B. Lower bound of peelability*

We continue to find the lower bound of the peelability (i.e., the upper bound of the failure probability).

**Definition A.1** (Isomorphism)**.** A graph $\mathcal{G} = \{V, E\}$ and a hypergraph $\mathcal{G}' = \{V', E'\}$ are isomorphic if there is a bijection $f$ from $V$ to $E'$ such that for any $e = (v_1, v_2) \in E$ iff. there is a vertex $v' \in V'$ and $v'$ is connected by $f(v_1) \in E'$ and $f(v_2) \in E'$.

Without considering *self-cancellation*, denote the onion $R = P_1, P_2, \cdots, P_t$, where $P_i = \{x_{i,1}, x_{i,2}, \cdots, x_{i,n_i}\}$ is the $i$-th peel and $P_1$ is outermost peel, $n_i$ is the number of keys in $P_i$ and $n_1 + n_2 + \cdots + n_t = n$. Let us follow our peeling process to construct a graph $\mathcal{G}$:

1) Create a virtual start vertex $x_0$;
2) Virtually connect $x_0$ to all vertices (keys) in $P_1$;
3) For each $x_{i,j} \in P_i$ $(i = 2, 3, \ldots, t, j = 1, 2, \ldots, n_i)$, connect $x_{i,j}$ with a vertex $x$ in the outer peels $P_{i'}$ $(i' < i)$ if they are connected. Here two vertices are connected when they are mapped to the same bin;
4) For any two vertices in the same peel, also connect them if they are mapped to the same bin.

In the above construction, it is noted that:

1) The virtual vertex and its edges are not included in $\mathcal{G}$;
2) The core can be taken as the innermost peel $P_t$;
3) A vertex has $k$ hash locations. If it is mapped to the same bin $k' \leq k$ times, it will have $k'$ self-loops (i.e., self-connected).
4) Our peeling process can be taken as a special breadth-first search (BFS) for $\mathcal{G}$ (start from $x_0$ to the core $P_t$). In a normal BFS, one can choose any neighbour vertex for further searching. Here one can only choose its connected vertices in the inner adjacent peel.

Based on $S = \{(x_i, (h_1(x_i), h_2(x_i), \cdots, h_k(x_i)))\}_n$, we can construct a random hypergraph $\mathcal{G}' = \{e'_i = (h_1(x_i), h_2(x_i), \cdots, h_k(x_i)) | \forall x_i \in X\}$, where $X$ is the key set. Also, we construct a graph $\mathcal{G}$ as above (i.e., connect all two vertices if they are connected by merging steps (3) and (4)).

**Theorem A.3.** *$\mathcal{G}$ and $\mathcal{G}'$ are isomorphic.*

*Proof.* Intuitively, they are both constructed from $S$. Therefore, they should be equivalent. Specifically,

- For any vertex $x_i \in V$, there is a corresponding hyperedge $f(x_i) = (h_1(x_i), h_2(x_i), \cdots, h_k(x_i)) \in E'$; vice versa.
- For any edge $e = (x_i, x_j) \in E$, there are two corresponding connected hyperedges $f(x_i)$ and $f(x_j)$ in $E'$; vices versa.

□

**Lemma A.4.** *Denote $\mathcal{G}$'s 2-core size as $s$ and $\mathcal{G}'$'s 2-core size as $s'$. If $\mathcal{G}$ and $\mathcal{G}'$ are isomorphic, we have $s = s'$.*

*Proof.* The 2-core of a (hyper)graph is the maximal subgraph among subgraphs with minimum degree 2. Because of iso-

morphism, when deleting one vertex $x_i$ from $\mathcal{G}$, we delete a hyperedge $f(x_i) = (h_1(x_i), h_2(x_i), \cdots, h_k(x_i)) \in E'$; vice versa. Therefore, $s = s'$. □

**Lemma A.5.** *Peeling $\mathcal{G}$ and $\mathcal{G}'$ get the same failure probability.*

*Proof.* We can easily deduce it from Lemma A.4. □

Now we consider the before-mentioned *self-cancelled* cases. A key may be mapped to the same bin for multiple times. If the mapping times are even, the XOR sum of any share in this bin will be zero. The *self-cancelled* cases of a key happens with probability $\geq 1/m$, which cannot be ignored. Therefore, in PGBF, if a key has even duplicated mappings (e.g., $d$ mappings), we reduce the related count in the counting table by $d$. In $\mathcal{G}$, we correspondingly remove the key's $d$ self-loops; then we have PGBF's graph $\mathcal{G}^*$.

**Lemma A.6.** *Denote the size of PGBF onion's core as $s^*$, we have $s^* \leq s'$.*

*Proof.* PGBF's core size is equal to $\mathcal{G}^*$'s core size. Since $\mathcal{G}^*$ is a sub-graph of $\mathcal{G}$, it is easy to know $s^* \leq s = s'$. □

**Lemma A.7.** *The failure probability of PGBF is no greater than peeling $\mathcal{G}'$ to get its empty 2-core.*

*Proof.* We can easily deduce it from Lemma A.6. □

Here we find the failure probability upper bound. By utilizing this upper bound, we can have the following Lemma.

**Lemma A.8.** *Given the same $n$, $k$, core-size $s'$, and failure probability $\epsilon$, the expansion rate of a single PGBF is no greater than that of the cuckoo graph.*

*Proof.* From Lemma A.7, we know that given the same $n$, $\eta = m/n$, $k$, and the failure probability, the core size of PGBF is no greater than the core of a cuckoo graph (i.e., 2-core of a random hypergraph). Therefore, given the same $n$, $k$, and the core size, a PGBF can reduce $\eta = m/n$ to achieve the same failure probability. In other words, given the same settings, a PGBF has a no greater expansion rate than peeling a cuckoo graph to get its 2-core. □

From Lemma A.8, we can utilize the existing parameter settings to set the expansion rates of PGBF. For example, PaXoS [2] and 3H-GCT [3] are built based on cuckoo graph by respectively setting $k = 2$ and $k = 3$. The 2-core size upper bound in PaXoS and 3H-GCT are in $O(\log(n))$ and respectively set as $\log(n)$ and $0.5\log(n)$. Their expansion rates are respectively set as 2.4 and 1.3. Therefore, to achieve the same negligible failure probability, if setting $k = 2$ and core-size upper bound as $\log(n)$, PGBF can set $\eta \leq 2.4$; if setting $k = 3$ and core-size upper bound as $0.5\log(n)$, PGBF can set $\eta \leq 1.3$. It is noted that setting both a smaller core size and a smaller expansion rate than PaXoS and 3H-GCT are also possible. However, we find their parameter settings can result in failure cases with non-negligible probability when the set size is not large (e.g., $n \leq 2^{12}$).

## APPENDIX B
## MORE DETAILS ABOUT MULTI-PGBF INSTANTIATIONS

From Table II, we can find that the core size upper bound $\bar{\alpha}$ is small compared with input set size $n$ (i.e., $\bar{\alpha} = O(\log(n))$. Then the input sizes for the following PGBFs will be small and make it easier to set the parameters. For very small set sizes (e.g., $n = 2$ and $n = 3$), we can calculate $f(m, n, \alpha)$. However, the discussions are still complicated for larger ones (e.g., $4 \leq n \leq 18$). Since Multi-PGBF is a recursive structure, we also fit the experimental results to get $f(m, n, \alpha)$ for the following PGBFs.

In the instantiations, we set the number of PGBFs $u = 3$ because $u = 3$ works well to make the failure probability negligible in the settings. By utilizing more PGBFs can be easier to amplify the failure probability by using less space (i.e., a smaller $m$). However, with the increase of number of hash functions $k$, the decoding efficiency will be decreased. For the second and third PGBF, we need to find appropriate expansion rates $\eta_2$ and $\eta_2$ to achieve negligible $\epsilon$. To narrow the searching ranges, we review the probability in equation (6). If $\bar{\alpha}_1 \geq 3$, it is easy to know

$$\begin{aligned}\epsilon \geq f(m_1, n, 3) \sum_{j_2=2}^{f^{-1}(m_2,3,2^{-\lambda})} \cdots \sum_{j_{u-1}=2}^{f^{-1}(m_{u-1},j_{u-2},2^{-\lambda})} f(m_2, 3, j_2) \\ \cdots f(m_{u-1}, j_{u-2}, j_{u-1}) g(m_u, j_{u-1}) \\ + f(m_1, n, 2) f(m_2, 2, 2) \cdots f(m_u, 2, 2),\end{aligned}$$

where $3 \geq j_2 \geq j_3 \geq \cdots \geq j_{u-1} \geq 2$. Since we know $(f(m, 3, 2), f(m, 3, 2), f(m, 2, 2))$ for very small set sizes and $(f(m_1, n, 2), f(m_1, n, 3))$ for large set sizes, we can easily compute the above probability, which can provide a lower bound. After setting $u = 3$, we have

$$\begin{aligned}\epsilon \geq & f(m_1, n, 3) f(m_2, 3, 3) \sum_{j_3=2}^{3} f(m_3, 3, j_3) \\ & + f(m_1, n, 3) f(m_2, 3, 2) f(m_3, 2, 2) \\ & + f(m_1, n, 2) f(m_2, 2, 2) f(m_3, 2, 2).\end{aligned}$$

By utilizing this lower bound, it becomes easier to find the $m_2$ and $m_3$. Further setting $m_2 = m_3$ will make it easier.

After the above analysis, we set the parameters of Multi-PGBF in Table III. In this table, for the large set size $n = 2^{20}$, we set the core size of the first PGBF and the expansion rates of the second and third PGBFs as the ones of $n = 2^{16}$. For the core size, from Table II, one can easy to find that the larger the $n$, the smaller the core size upper bound $\bar{\alpha}$. Therefore, we simply set $\bar{\alpha}_1 = 9$ for $n = 2^{20}$ as the one for $n = 2^{16}$. The reason why we do not fit the experimental results for $n = 2^{20}$ is that the maximum core size we observe after a large number of trials is only 2. Too few points make it non-trivial to fit. After setting $\bar{\alpha}_1 = 9$ in the first PGBF, by setting the same expansion rates for the second and third PGBF as $n = 2^{16}$, Multi-PGBF can also hold a negligible failure probability. Though assuming a large core costs a larger table size, the added space is relatively very small because $(\eta_2\bar{\alpha}_1 + \eta_3\bar{\alpha}_2)/n < 2^{-13}$.

## APPENDIX C
## OBLIVIOUSNESS PROOF FOR MULTI-PGBF

**Theorem C.1.** *Multi-PGBF in Fig. 4 achieves obliviousness.*

*Proof.* Denote $d(x_i)$ as a bit vector with length $m$, in which its $h_{j,z}(x_i)$th bit is as '1' ($\forall j \in [u], z \in [k_i]$) and the remaining bits as '0'. Then the encoding of Multi-PGBF can be written as a system of linear equations

$$\begin{bmatrix} d(x_1) \\ d(x_2) \\ \vdots \\ d(x_n) \end{bmatrix} \begin{bmatrix} D_1 \\ D_2 \\ \vdots \\ D_m \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}.$$

Since Multi-PGBF can achieve a negligible failure probability, it is easy to know the above system of linear equations is solvable with an overwhemingly large probability. Also, the left bit matrix $[d(x_1), d(x_2), \cdots, d(x_n)]^T$ is non-singular (i.e., the rank of it is $n$). The reason is that the $n$ keys $x_1, x_2, \cdots, x_n$ have $n$ unique anchor bins in $D$. Specifically, the peeling of Multi-PGBF is to find an ordering sequence by their anchors, which corresponds to row swappings in the system of linear equations. Denoting $(x'_1, v'_1), (x'_2, v'_2), \cdots, (x'_n, v'_n)$ is the ordering sequence (i.e., $(x'_i = S_i[1], v'_i = S_i[2])$ in Figure 4, and $S_i[j]$ is the $j$th item of the quadruple $S_i$), one can swap rows of $[d(x_1), d(x_2), \cdots, d(x_n)]^T$ and $[v_1, v_2, \cdots, v_n]^T$ to rewrite the above system to be

$$\begin{bmatrix} d(x'_1) \\ d(x'_2) \\ \vdots \\ d(x'_n) \end{bmatrix} \begin{bmatrix} D_1 \\ D_2 \\ \vdots \\ D_m \end{bmatrix} = \begin{bmatrix} v'_1 \\ v'_2 \\ \vdots \\ v'_n \end{bmatrix}.$$

After peeling, one needs to do unpeeling to sequentially assign the values from $x'_n$ to $x'_1$, which corresponds to sequentially solve the linear equation from the $n$th row to the 1st row. Without loss of generality, denote the unpeeling goes to assign the value of $x'_i$ and $h_{j,z^*}(x'_i)$ is its anchor location. Then all $d(x'_{i+1})[h_{j,z^*}(x'_i)]$, $d(x'_{i+2})[h_{j,z^*}(x'_i)], \cdots, d(x'_n)[h_{j,z^*}(x'_i)]$ must be '0's; otherwise $h_{j,z^*}(x'_i)$ cannot be $x'_i$'s anchor and the final share of $v'_i$ cannot be assigned. In other words, $d(x'_i)$ is linearly independent with $d(x'_{i+1}), d(x'_{i+2}, \cdots, d(x'_n))$ for $i = n-1, n-2, \cdots, 1$. Then we know $[d(x_1), d(x_2), \cdots, d(x_n)]^T$ is non-singular. Denote $D_h$ contains the hashed bins of $D$ and the non-anchors bins in $D'$ are assigned with random values. Then given $[d(x_1), d(x_2), \cdots, d(x_n)]^T$ is non-singular and $[v_1, v_2, \cdots, v_n]^T$ is uniformly distributed in $\mathcal{V}^n$, one can know the solution $D_h$ is also uniformly distributed in $\mathcal{V}^m$. For the remaining non-hashed bins, they are assigned with random values in $\mathcal{V}$. Therefore, $D$ is uniformly distributed in $\mathcal{V}^m$. For any two key sets $X^1$ and $X^2$ from $\mathcal{K}^n$ and a random value set $V$ from $\mathcal{V}^n$, if both encoding succeed, $D^1 \leftarrow Encode(X^1, V)$ and $D^2 \leftarrow Encode(X^1, V)$ are also uniformly distributed and computationally indistinguishable. In other words, Multi-PGBF achieves obliviousness.

□

## APPENDIX D
## MORE DETAILED COMPARISONS BETWEEN MULTI-PGBF AND PAXOS/3H-GCT

The differences between Multi-PGBF and PaXoS/3H-GCT are listed as follows:

1) **Basis**. PaXoS/3H-GCT are constructed based on the cuckoo graph (i.e., graph for PaXoS and hypergraph for 3H-GCT). We design our PGBF and Multi-PGBF based on CBF, in which we play the classical balls-into-bins game in statistics (i.e., throw $nk$ balls into $m$ bins). In subsection III-D, we have theoretically proved that peeling on the CBF can be equal to peeling on a graph, which is isomorphic to the cuckoo graphic. Therefore, to study the peelability of a random cuckoo graphic, we can turn to play the corresponding balls-into-bins game. We believe our design provides a new angle to study a random cuckoo graphic. Also, in subsection IV-B, we have provided a concrete theoretical failure probability for Multi-PGBF. For PGBF, we also have equation (3). In subsection IV-C, we have provided our concrete instantiations for Multi-PGBF by studying $f(m, n, \alpha)$ with empirical evaluations. If one can know the theoretical $f(m, n, \alpha)$, then the concrete failure probability lower bound of a random cuckoo graph can be provided, which is of great value.
2) **Peeling strategy**. Though PaXoS/3H-GCT uses 'peeling' to describe the core part of their encoding process, Multi-PGBF is the one that really peels. Multi-PGBF does the peeling process peel by peel (i.e., set by set), while PaXoS and 3H-GCT do it key by key. It is natural to peel an onion but not to peel a ball of threads, which corresponds to their 'peeling' strategy. Specifically, they take depth-first search (DFS) to continually find vertices connected only by a single edge or (hyperedge) in the cuckoo graph. However, we take a strategy similar to breadth-first search (BFS) by exploiting CBF to identify the key anchors (see Appendix A-B for more details). In each peel, the order of the keys does not affect the encoding correctness and these keys can be peeled independently.
3) **Encoding.** Multi-PGBF runs in super-linear encoding complexity (i.e., $O(kn\log(n))$). PaXoS and 3H-GCT respectively need $O(\lambda n + (\lambda + \log(n))^3)$ and $O(\lambda n + (\lambda + 0.5\log(n))^3)$. The non-linear parts $O((\lambda + \log(n))^3)$ and $O((\lambda + 0.5\log(n))^3)$ are used to handle the non-empty core by using the normal Gaussian elimination. Even though their encoding efficiency is claimed as $O(\lambda n)$, they are not purely linear. When the set size is small (e.g., $n = 2^8$), it is obvious that the non-linear part $O((\lambda + \log(n))^3)$ (or $O((\lambda + 0.5\log(n))^3)$) dominates the linear part $O(\lambda n)$. In other words, PaXoS and 3H-GCT will degenerate to the normal Gaussian elimination. From Table IV, Multi-PGBF gain the highest encoding efficiency. Specifically, Multi-PGBF is respectively

$10.42\times \sim 17.59\times$ and $10.76\times \sim 22.86\times$ as fast as PaXoS and 3H-GCT.

4) **Decoding efficiency**. To decode a key, all structures (except for the interpolation) need to compute XOR sums. However, Multi-PGBF is much more efficient than PaXoS and 3H-GCT. In concrete instantiations, Multi-PGBF only needs to compute the XOR sum for $k = 9$ shares. However, PaXoS and 3H-GCT respectively need to compute the XOR sum for $2 + (\lambda + \log(n))/2$ and $3+(\lambda+0.5\log(n))/2$ shares in average. From Table IV, Multi-PGBF also gains the highest decoding efficiency. Specifically, to decode $n$ items, Multi-PGBF are only respectively $20.19\times \sim 45.73\times$ and $20.01\times \sim 39.09\times$ as fast as them.
5) **Fine-tuned expansion rates.** From Lemma A.7, we know the expansion rate of Multi-PGBF is no greater than cuckoo graph based PaXoS/3H-GCT by using the same parameters (i.e., $n$, $k$, $\epsilon$). The expansion rate of Multi-PGBF can be as low as 1.23 for $n \geq 2^{20}$, while that of PaXoS and 3H-GCT are respectively 2.4 and 1.3. More importantly, PaXoS/3H-GCT takes fixed expansion rates (i.e., 2.4/1.3) for all set sizes, which is incorrect. For example, when $n = 2^{11}$, we find failure cases for both of them after only $2^{20}$ trials, even they set the largest 2-core size bounds (i.e., $\log n$ and $0.5 \log n$). From Table II, we can find that our expected core-size bound is $\bar{\alpha}_1 = 18 > 0.5 \times 12$ when $n = 2^{12}$ and $\eta_1 = 1.3$ (the same as 3H-GCT), which is not expected. In Table III, we provide fine-tuned expansion rates for different set sizes. It is easy to find that the smaller the set size, the larger the expansion rate.

In addition to the above differences, in 3H-GCT [3], the authors also proposed a star structure to amplify the failure probability by doing 'replicas'. For example, if they use 2 'replicas' to encode a set, in which each 'replicas' fails in probability $\epsilon$. Then the total failure probability will be $\epsilon^2$. Simply doing 'replicas' brings both extra communication costs and computation costs. In Multi-PGBF, in the meanwhile of filtering, each PGBF also plays the role of amplifying the failure probability, which can be easily found from equation (2). There is no need to deliberately do 'replicas' in Multi-PGBF. One more difference is that there may contain zero shares in the hash table of PaXoS when solving the 2-core by using Gaussian elimination, making PaXoS not oblivious. This deficiency is pointed out and rectified in [10] by replacing the zero shares with random ones. It is noted that this deficiency also applies to 3H-GCT and we also detect it in the source code (we fix it in the comparison.). There is such a concern in Multi-PGBF.

Furthermore, Multi-PGBF provides a general solution to construct an OKVS. Actually, we can integrate any single structure into our design. For example, we can have the first one as a PGBF, then the second one as a cuckoo graph; or we can have the first one as a PGBF, the second one as a GBF, then the third one as an interpolation. PaXoS/3H-GCT can be a special case where the first one is a cuckoo graph and the second one is an interpolation.

## APPENDIX E
## THE ENCODING COST BREAKDOWN OF MULTI-PGBF

The breakdown encoding cost of Multi-PGBF is shown in Table VII. In this table, we break the encoding costs into *Init*, *Peeling*, and *Unpeeling*. In *Init*, it initiates the parameters, the hash functions, and variables. In *Peeling*, it builds the CBF and finds the anchor bins for the keys. In *Unpeeling*, it assigns the final share of each key's value into its anchor bin.

From this table, one can find that the peeling phase accounts for the major costs of encoding (i.e., $70.2\% \sim 81.7\%$). The unpeeling phase is much cheaper than the peeling phase. In comparison, the unpeeling phase only accounts for $18.3\% \sim 29.9\%$ encoding time. Since the peeling phase is only related to the keys rather than the values, the peeling phase can be done just once. The unpeeling phase is related with the values. When the encoding needs to be called multiple times with the same keys, the peeling can be done just once and the unpeeling can be done for multiple times. For the same keys, if encoding $\theta$ times, this detachment improvement can save $70.1\%(1 - 1/\theta) \sim 81.7\%(1 - 1/\theta)$ computation costs. In the multi-party PSI protocol (see subsection VI-B), a party may need to encode $\theta = 2$ times with the same keys, which can save $35.05\% \sim 40.85\%$ encoding costs. Having the detachment also has another advantage. In some applications, the peeling can be done offline since the keys are already known by the parties; then after getting the values to be encoded in the interactions, the unpeeling can be done online. Having this detachment, the online time can be reduced. This advantage can also be found in multi-party PSI protocol [13].

## APPENDIX F
## ANALYSES OF OTHER PSI VARIANT PROTOCOLS IN [18]

In [10], the authors combined OPRF and OKVS to design their OPPRF protocol. Then they utilize cuckoo hashing, OPPRF, and a two-party computation primitive $\mathcal{F}_{2pc}$ to show a circuit PSI protocol as [29]. In [18], it does not combine OPRF and OKVS to show an OPPRF protocol. Instead, they claim that they design a circuit PSI protocol using cuckoo hashing, PSI, OKVS, and $\mathcal{F}_{2pc}$. In step 4 of their circuit PSI protocol, they claim both parties run their PSI protocol, except that the sender does not send the OPRF values to the receiver. Actually, it is OPRF rather than PSI. In PSI, the receiver knows the intersection, which is not allowed in circuit PSI. Therefore, they actually utilize cuckoo hashing, OPRF, OKVS, and $\mathcal{F}_{2pc}$, which is similar to [29], [10] that utilize cuckoo hashing, OPPRF, and $\mathcal{F}_{2pc}$. By using circuit PSI, it is easy to realize PSI-cardinality and PSI-sum using a specific $\mathcal{F}_{2pc}$, which has already been done in [29]. However, the authors claimed they proposed their PSI-cardinality and PSI-sum protocols. In addition to the similarity, because the basic PSI protocol is insecure, their other PSI variants are also insecure, and the reported performance of these protocols is incorrect.

TABLE VII: The encoding cost breakdown in ms for different set sizes of Multi-PGBF. In each cell, the data in '()' indicates the percentage of the breakdown cost over the total encoding cost.

| Encoding | $n = 2^{12}$ | $n = 2^{16}$ | $n = 2^{20}$ | $n = 2^{24}$ |
|---|---|---|---|---|
| *Peeling* | 0.249 (86.90%) | 2.894 (81.07%) | 68.226 (73.44%) | 2475.227 (81.38%) |
| *Unpeeling* | 0.038 (13.10%) | 0.676 (18.93%) | 24.670 (26.56%) | 566.396 (18.62%) |
| Total | 0.286 | 3.570 | 92.895 | 3041.623 |

## APPENDIX G
## PERFORMANCE COMPARISONS BETWEEN K-STAR [13] WITH DIFFERENT OKVSS

From Table VIII, as in the two-party PSI protocol, K-Star with Multi-PGBF and C-Multi-PGBF can still run the fastest in most settings. The exceptions in the running time are mainly when $(b, t) = (10, 9)$ in the WAN. Since RB-OKVS has a lower expansion rate, K-Star with RB-OKVS(0.1) has lower communication costs. Therefore, when the number of parties $b$ and the number of colluding parties $t$ are both large, the communication advantage of K-Star with RB-OKVS(0.1) becomes more obvious, thus running faster than K-Star with Multi-PGBF and C-Multi-PGBF. Specifically, for $(b, t) = (10, 9)$ and $n \in 2^{16}, 2^{20}$, it is respectively $4.82\% \sim 9.22\%$ and $4.85\% \sim 20.03\%$ faster.

In other settings, K-Star with Multi-PGBF and C-Multi-PGBF can run much faster. For example, in the LAN for settings except for $(b, t) = (10, 9)$, they can be $28.38\% \sim 67.38\%$ and $25.14\% \sim 65.75\%$ faster than K-Star with RB-OKVS(0.1). K-Star with C-RR [14] is faster than it with RR [14]. K-Star with Multi-PGBF and C-Multi-PGBF can still obtain the running efficiency. For example, compared with K-Star with C-RR [14], K-Star with Multi-PGBF can run $2.35\% \sim 19.73\%$ faster in the LAN.

## APPENDIX H
## MORE ANALYSIS FOR O-RING AND K-STAR [13] IN UNBALANCED SETTINGS

### A. The modified mPSI protocols O-Ring and K-Star [13]

### B. Padding for [13] is inappropriate

In [19], the authors pad the small sets to a set with size equal to the largest set for O-Ring and K-Star when they compare their unbalanced multi-party PSI protocol with O-Ring and K-Star. This raises two concerns. First, this results in a lot of extra costs for O-Ring and K-Star, making their comparisons unfair. For example, in their experiment, the large set size can be as large as $|Y| = 2^{22}$, which is $\frac{|Y|}{|X|} = \frac{2^{22}}{2^{10}} = 2^{12}$ times as large as the small set of size $|X|$. However, even assuming O-Ring and K-Star do not support unbalanced cases, there is no need to do the padding. For each small set $X$, the party only needs to prepare an OKVS table of the large size $|Y|$ and encode $X$ into this table. Second, simply padding for O-Ring and K-Star brings a security concern if the input domain is not large. Let us denote the input item bit length is $\sigma$. Assuming that all other parties have the same set $Y$ and the receiver has a set $X \subset Y$, the probability that $P_1$ knows at least one padded item $x \in Y \backslash X$ is $p = 1 - (\binom{2^\sigma - |Y|}{|Y| - |X|} / \binom{2^\sigma - |X|}{|Y| - |X|})$. When $\sigma = 32$, $p \approx 1$. In other words, $P_1$ almost certainly knows at least one item of $Y \backslash X$, which is not allowed in multi-party PSI. Even when $\sigma = 64$ as in [19], $p \approx 2^{-20}$, which is also non-negligible (i.e., $> 2^{-\lambda} = 2^{40}$). Therefore, padding not only incurs extra costs, but also results in a security concern, which is not expected.

In [19], it only considers $t = b - 1$. However, in O-Ring/K-Star [13], and previous multi-party PSI protocols (e.g., [7], [30]), it assumes there are $\leq t$ colluding parties. It is unknown how the multi-party PSI protocol [19] can be extended for an arbitrary $t$. Considering the maximum number of colluding party (i.e., $n-1$), their protocol can work for any $t \in [1, n-1]$. For their protocol, the total communication and computation complexity are both in linear with the number of parties $b$ (i.e., $O(bn)$). However, O-Ring/K-Star [13] has $O(btn)$ for both total communication and computation. Therefore, for a large $t$ (e.g., $t = n - 1$), the complexity becomes quadratic in $b$ (i.e., $O(b^2 n)$); for a small $t$ (e.g., $t = 1$), the complexity becomes linear in $b$ (i.e., $O(bn)$) as [19]. For a small $t$, O-Ring/K-Star [13] has better performance than a larger $t$.

TABLE VIII: The total running time (in seconds) and communication cost (in MB) comparisons in the multi-party PSI protocol *K-Star* [13] by using different OKVSs. 'RB' is short for RB-OKVS. The set size $n$, the number of parties $b$, and the number of colluding parties $t$ are the same as *K-Star* [13]. For LAN, the bandwidth is 1 Gbps and the latency is 0.06 ms. For WAN, the bandwidth is 10 Mbps and the latency is 80 ms. The best results are marked in bold.

| $(b,t)$ | (3,1) | | | (3,2) | | | (10,1) | | | (10,4) | | | (10,9) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $n$ | $2^{12}$ | $2^{16}$ | $2^{20}$ | $2^{12}$ | $2^{16}$ | $2^{20}$ | $2^{12}$ | $2^{16}$ | $2^{20}$ | $2^{12}$ | $2^{16}$ | $2^{20}$ | $2^{12}$ | $2^{16}$ | $2^{20}$ |
| **Running time** (LAN, 1 Gbps) | | | | | | | | | | | | | | | |
| 3H-GCT [3] | 0.054 | 0.253 | 5.800 | 0.080 | 0.364 | 8.004 | 0.094 | 0.522 | 10.994 | 0.220 | 0.943 | 18.632 | 0.391 | 1.530 | 29.246 |
| RR [14] | 0.024 | 0.127 | 2.038 | 0.059 | 0.213 | 3.297 | 0.041 | 0.236 | 3.714 | 0.184 | 0.645 | 10.930 | 0.338 | 1.239 | 20.005 |
| C-RR [14] | 0.021 | 0.119 | 1.251 | 0.052 | 0.182 | 2.356 | 0.039 | 0.237 | 2.385 | 0.196 | 0.672 | 10.506 | 0.359 | 1.307 | 19.997 |
| RB(0.03) [15] | 0.065 | 0.299 | 4.572 | 0.126 | 0.446 | 6.883 | 0.103 | 0.644 | 10.401 | 0.249 | 0.997 | 17.280 | 0.417 | 1.531 | 25.261 |
| RB(0.1) [15] | 0.035 | 0.166 | 2.269 | 0.070 | 0.256 | 3.695 | 0.080 | 0.367 | 4.942 | 0.218 | 0.750 | 11.215 | 0.359 | 1.270 | 20.276 |
| Multi-PGBF | 0.020 | 0.105 | **1.150** | **0.046** | 0.175 | **2.139** | 0.038 | 0.188 | 2.347 | **0.178** | **0.626** | **9.485** | 0.340 | **1.211** | **18.159** |
| C-Multi-PGBF | **0.019** | **0.101** | 1.153 | 0.050 | **0.175** | 2.249 | **0.036** | **0.188** | **2.190** | 0.179 | 0.632 | 10.001 | **0.330** | 1.224 | 19.657 |
| **Running time** (WAN, 200 Mbps) | | | | | | | | | | | | | | | |
| 3H-GCT [3] | 1.543 | 2.348 | 9.806 | 2.603 | 3.461 | 15.710 | 2.979 | 5.634 | 22.106 | 4.562 | 6.985 | 67.660 | 4.764 | 9.307 | 100.969 |
| RR [14] | 1.320 | 2.030 | 5.757 | **2.176** | 2.863 | 10.008 | 2.711 | 5.167 | 13.578 | 3.793 | 6.078 | **46.520** | 4.016 | 8.388 | 84.259 |
| C-RR [14] | 1.327 | 1.991 | 5.299 | 2.206 | 2.901 | 9.493 | **2.696** | 5.195 | 13.409 | 3.794 | 6.158 | 62.072 | 4.007 | 8.673 | 94.543 |
| RB(0.03) [15] | 1.368 | 2.196 | 8.044 | 2.217 | 3.068 | 13.015 | 2.808 | 5.589 | 19.792 | 3.861 | 5.947 | 52.544 | 4.052 | **7.877** | 75.543 |
| RB(0.1) [15] | 1.334 | 2.038 | 5.838 | 2.213 | 2.901 | 9.999 | 2.764 | 5.255 | 14.643 | 3.799 | **5.850** | 49.002 | 4.014 | 7.901 | **74.361** |
| Multi-PGBF | **1.313** | 2.005 | **4.936** | 2.205 | **2.818** | **8.853** | 2.714 | **5.120** | **12.490** | 3.778 | 5.989 | 48.274 | **3.998** | 8.301 | 81.915 |
| C-Multi-PGBF | 1.325 | **1.972** | 5.231 | 2.194 | 2.848 | 9.351 | 2.713 | 5.148 | 13.317 | **3.766** | 5.941 | 57.748 | 4.013 | 8.304 | 92.988 |
| **Communication cost** | | | | | | | | | | | | | | | |
| 3H-GCT [3] | 0.476 | 2.773 | 44.380 | 1.675 | 7.674 | 111.477 | 0.742 | 6.759 | 121.429 | 14.737 | 63.946 | 904.237 | 32.729 | 137.471 | 1910.699 |
| RR [14] | 0.469 | 2.666 | 39.631 | 1.657 | 7.396 | 99.607 | 0.724 | 6.477 | 108.371 | 14.588 | 61.666 | 808.080 | 32.413 | 132.624 | 1707.708 |
| C-RR [14] | 0.469 | 2.856 | 43.876 | 1.657 | 7.890 | 110.217 | 0.724 | 6.978 | 120.042 | 14.588 | 65.716 | 894.024 | 32.413 | 141.236 | 1889.144 |
| RB(0.03) [15] | **0.435** | **2.265** | **33.313** | **1.570** | **6.355** | **83.812** | **0.632** | **5.420** | **90.994** | **13.874** | **53.143** | **680.151** | **30.899** | **114.500** | **1437.638** |
| RB(0.1) [15] | 0.443 | 2.396 | 35.553 | 1.591 | 6.697 | 89.412 | 0.654 | 5.765 | 97.154 | 14.049 | 55.938 | 725.509 | 31.271 | 120.445 | 1533.394 |
| Multi-PGBF | 0.469 | 2.661 | 39.716 | 1.658 | 7.385 | 99.818 | 0.724 | 6.465 | 108.603 | 14.591 | 61.572 | 809.793 | 32.419 | 132.423 | 1711.323 |
| C-Multi-PGBF | 0.469 | 2.661 | 43.889 | 1.658 | 7.385 | 110.251 | 0.724 | 6.465 | 120.080 | 14.591 | 61.572 | 894.302 | 32.419 | 132.423 | 1889.730 |

**Parameters:** The statistical and computational security parameters are $\lambda$ and $\kappa$, respectively. The number of parties $n$. Sort the parties, such that the set size of $P_i$ is no larger than $P_{i+1}$ (i.e., $|X_i| \leq |X_{i+1}|$, $\forall i \in [1, n-1]$). The number of colluding parties $t$. The set size $\underline{m_i}$ of $P_i$. The item's bit length $\sigma$. A OPRF primitive $\mathcal{F}_{oprf}$. A PRF $F_{(\cdot)}(\cdot) : \{0,1\}^{\kappa} \times \{0,1\}^{\sigma} \mapsto \{0,1\}^{\ell}$.

**Inputs:** Each party $P_i$ ($i \in [n]$) inputs a set $X_i = \{x_j^i\}_{\forall j \in [m]}$.

**Protocol:**

1) **[Share distribution]**. Denote $b = n - t + 1$.
   a) **[OPRF]**. $P_i$ ($\forall i \in [b, n]$) respectively invokes an instance of $\mathcal{F}_{oprf}$ with $P_j$ ($\forall j \in [1, n] \backslash \{i\}$) to get $\{Q^i(x_u^i) = \bigoplus_{j=1}^{i-1} F_{k^{i,j}}(x_u^i) \oplus \bigoplus_{j=i+1}^{n} F_{k^{i,j}}(x_u^i)\}_{\forall u \in [m]}$ and $\{F_{k^{i,j}}(x_u^j)\}_{\forall u \in [m]}$, where $k^{i,j}$ is the common OPRF sender $P_i$'s OPRF key.
   b) **[Mask]**. If $b > 2$, $P_1$ respectively sends a random PRF key $s^{1,j}$ to $P_j$ ($\forall j \in [2, b-1]$).
2) **[Share collection]**.
   - $b > 2$. $P_i$ ($\forall i \in [2, b-1]$) computes $v_j^i = F_{s^{1,i}}(x_j^i) \oplus \bigoplus_{u=b}^{n} F_{k^{u,i}}(x_j^i)$ ($\forall j \in [m]$). $P_i$ ($\forall i \in [b, n]$) computes $v_j^i = Q^i(x_j^i) \oplus \bigoplus_{u=b}^{i-1} F_{k^{u,i}}(x_j^i) \oplus \bigoplus_{u=i+1}^{n} F_{k^{u,i}}(x_j^i)$ ($\forall j \in [m]$). $P_1$ computes $v_j^1 = \bigoplus_{u=2}^{b-1} F_{s^{1,u}}(x_j^1) \oplus \bigoplus_{u=b}^{n} F_{k^{u,1}}(x_j^1)$ ($\forall j \in [m]$).
   - $b = 2$. $P_i$ ($\forall i \in [1, b-1]$) computes $v_j^i = \bigoplus_{u=b}^{n} F_{k^{u,i}}(x_j^i)$ ($\forall j \in [m]$). $P_i$ ($\forall i \in [b, n]$) computes $v_j^i = Q^i(x_j^i) \oplus \bigoplus_{u=b}^{i-1} F_{k^{u,i}}(x_j^i) \oplus \bigoplus_{u=i+1}^{n} F_{k^{u,i}}(x_j^i)$ ($\forall j \in [m]$).
3) **[Share transition]**.
   - For O-Ring, in the OKVS chain (i.e., $P_2 \to P_3 \to \cdots \to P_n \to P_1$), each party $P_i$ has $\{(x_j^i, v_j^i)\}_{\forall j \in [m]}$. $P_2$ starts the transition and $P_1$ gets the transition value $v_j^1 = v_j^1 \oplus \bigoplus_{i=2}^{n} v_j^i$ for each item $x_j^1$. $P_1$ gets the intersection: $I = \{x_j^1 | v_j^1 = 0, \forall j \in [m]\}$.
   - For K-Star, $P_i$ ($i = 2, \cdots, n$) builds an OKVS table $T_i$ by encoding $\{(x_j^i, v_j^i)\}_{\forall j \in [m]}$ and sends it to $P_1$. Upon receiving these tables, $P_n$ also builds an OKVS table $T_n$ by encoding $\{(x_j^n, v_j^n \oplus \bigoplus_{u=2}^{n-1} Decode(T_u, x_j^n))\}_{\forall j \in [m]}$ and sends it to $P_1$. Then $P_1$ gets the intersection:

$$I = \{x_j^1 | v_j^1 \oplus \bigoplus_{u=2}^{n} Decode(T_u, x_j^1) = 0, \forall j \in [m]\}.$$

Fig. 9: Unbalanced O-Ring and K-Star. In **Parameters**, the sentence with an underline is the modification that is made to O-Ring and K-Star in [13].