Efficient Two-Party Secure Aggregation via Incremental Distributed Point Function

Nan Cheng  
University of St. Gallen  
St. Gallen, Switzerland  
nan.cheng@unisg.ch

Aikaterini Mitrokotsa  
University of St. Gallen  
St. Gallen, Switzerland  
katerina.mitrokotsa@unisg.ch

Feng Zhang  
Nanyang Institute of Technology  
Nanyang, China  
gudengxia@gmail.com

Frank Hartmann  
University of St. Gallen  
St. Gallen, Switzerland  
frank.hartmann@unisg.ch

Abstract—Computing the maximum from a list of secret inputs is a widely-used functionality that is employed either indirectly as a building block in secure computation frameworks, such as ABY (NDSS’15) or directly used in multiple applications that solve optimisation problems, such as secure machine learning or secure aggregation statistics. Incremental distributed point function (I-DPF) is a powerful primitive (IEEE S&P’21) that significantly reduces the client-to-server communication and are employed to efficiently and securely compute aggregation statistics.

In this paper, we investigate whether I-DPF can be used to improve the efficiency of secure two-party computation (2PC) with an emphasis on computing the maximum value and the k-th (with k unknown to the computing parties) ranked value from a list of secret inputs. Our answer is affirmative, and we propose novel secure 2PC protocols that use I-DPF as a building block, resulting in significant efficiency gains compared to the state-of-the-art. More precisely, our contributions are: (i) We present two new secure computation frameworks that efficiently compute secure aggregation statistics bit-wisely or batch-wisely; (ii) we propose novel protocols to compute the maximum value, the k-th ranked value from a list of secret inputs; (iii) we provide variations of the proposed protocols that can perform batch computations and thus provide further efficiency improvements; and (iv) we provide an extensive performance evaluation for all proposed protocols.

Our protocols have a communication complexity that is independent of the number of secret inputs and linear to the length of the secret input domain. Our experimental results show enhanced efficiency over state-of-the-art solutions, particularly notable when handling large-scale inputs. For instance, in scenarios involving an input set of five million elements with an input domain size of 31 bits, our protocol \( \Pi_{\text{Max}} \) achieves an 18% reduction in online execution time and a 67% decrease in communication volume compared to the most efficient existing solution.

1. Introduction

Privacy-preserving secure aggregation techniques are becoming increasingly crucial as the volume of sensitive data surges due to widespread digital device usage and the growth of business data [15], [16], [20]. These techniques safeguard the privacy of data holders, while enabling the extraction of valuable insights through aggregation analysis.

Secure computations of the maximum or k-th ranking element within a private dataset \( X \) are essential within secure aggregation analyses, finding applications across diverse sectors. For instance, in the context of smart grids, secure computation of the maximum value within a dataset \( X \) (representing collective energy usage) is key for operational integrity while upholding user privacy. It serves a foundational role in grid infrastructure planning, pinpointing peak load capacities that dictate crucial system enhancements to avert failures. It also plays a pivotal role in conducting grid stress tests, gauging system resilience during peak demand periods. Furthermore, maximum computation facilitates equitable energy distribution across diverse sources, particularly during times of high demand. It also aids in the detection of consumption anomalies [18], which could signal equipment failures or energy theft. Moreover, analyzing peak demand trends is vital for refining load forecasting models, a cornerstone for strategic energy procurement and generation planning.

Secure maximum computation also plays a vital role in scientific collaboration, especially within a consortium or union of medical data sources. For example, it could help determine the highest recorded level of a particular biomarker across different patient groups from various institutions, without revealing the data of any single patient. This is crucial in understanding the cause of a disease or effectiveness of a treatment, thus indicate a need for further investigation or intervention.

Motivation. However, existing privacy-preserving techniques that find the maximum of a large dataset are unsatisfactory, i.e., as it is the case in two recent secure aggregation frameworks [1], [9] that provide efficient solutions only for a set that comprises of short integers. On the other hand, given a set \( X \) of size \( m \), for protocols that employ secure comparison pair-wisely in \( O(m) \) iterations, because secure comparison is a costly non-linear primitive, the concrete efficiency of such general comparison-based solutions do not scale well when the number of inputs \( m \) is very large.

Problem and setting. Thus, in this paper, we consider a large number of clients holding secret values who want to outsource the computation to two powerful servers for secure aggregation analysis. More precisely, each client submits secret sharing (SS) of their secret value to two servers, we consider as the outsourced secure functionalities the following: (i) \( \mathcal{F}_{\text{MAX}} \): Computing the maximum of the clients’ secret inputs and outputs the SS of the targeted value; (ii) Function hiding \( \mathcal{F}_{\text{KRE}} \): With additional SS of a
secret index $k$ as input which hides the function definition, outputs the SS of the $k$-th ranked element of the clients’ secret inputs.

We consider that each client provides as input one/multiple Boolean SS and outsources the computation of the desired functionality to two honest-but-curious computing servers. This implies that the servers adhere strictly to the protocol’s prescribed steps. However they might extract sensitive information from their data during execution if possible. Our goal is to keep the result of the computation (i.e., maximum or the $k$-th ranked value) in secret shared form to the two servers so that it can be used as input for other functionalities. Of course the final result could also be revealed to a target receiver. All our protocols run in an outsourced distributed computing model. In this model, there is a large amount of clients who submit their secret shared input to two outsourced servers, and these servers execute the desired functionality in a privacy-preserving way. To achieve better online efficiency, we design our protocols in the offline/online model, where in the offline phase independent correlated randomness is generated among the servers, which are subsequently consumed in the online evaluation phase.

Our Contributions. We proposed protocols addressing these two functionalities effectively even dealing with a large set as input. Our protocols rely on incremental distributed point functions (I-DPFs), which were recently introduced by Boneh et al. [3]. In their work, they use I-DPF as the encoding of a client’s input such that it enables a client to secret-share the labels on the nodes of an exponentially large binary tree in a concise manner, while significantly reducing the communication overhead between the client and server. In our work, we also employ I-DPFs as a building block of our protocols, however, we use it differently and we adapt it to a new secure aggregation framework that efficiently computes secure aggregation tasks.

More specifically, our proposed protocols leverage secure incremental prefix counting and secure comparisons to achieve their desired functionality. The protocols that calculate the maximum or the $k$-th ranked element compute their target value bit-wisely from the most significant bit (MSB) to the least significant bit (LSB). The protocol, that verifies a given maximum candidate, determines the bit (MSB) to the least significant bit (LSB). The protocol, that utilizes function secret sharing for secure comparison.

Remark that our computation framework proposed in Fig. 4 or Fig. 5 also supports the computation of other secure aggregation statistics, due to page limit we have put related discussions in Appendix C.

1.1. Related work

For a set $X$ of size $m$ with each element of $n$ bits, from existing literature we distinguish two approaches that compute the maximum from a secret set, i.e., the comparison-based approach and encoding-based approach.

Comparison-based approach. This approach necessitates $m − 1$ secure comparisons in $\log m$ rounds to determine the maximum of $X$. Within this comparison-based approach, the overall efficiency for computing the maximum is determined by the efficiency of the secure comparison protocol employed. There is a rich literature on secure comparison protocols, for the purpose of secure maximum computation three secure comparison protocols are analysed in [7], namely Damgård et al.’s [13], Garay et al.’s [14], and Couteau [10]’s protocols. Damgård et al. [13]’s protocol is based on homomorphic encryption. Garay et al. [14]’s work employs the encryption scheme from [11]. Couteau [10]’s protocol applies a block decomposition technique and executes the comparison using Oblivious Transfer. When applied to computing the maximum, these protocols exhibit different communication complexities. Specifically, the method employed by Damgård et al. [13] requires a communication complexity of $O((mn(n + k))\log m)$ across $O(\log m)$ rounds. The approach employed by Garay et al. [14] necessitates $O(m)$ in $O(\log m \log n)$ rounds and the technique introduced by Couteau [10] requires $O(m n)$ in $O(\log m \log \log n)$ rounds. However, the scalability of these protocols is limited, particularly for large data sets, due to the high computational cost of secure comparisons. The number of communication rounds in these implementations is dependent on the number of the secret input values $m$.

MP-SPDZ [17] is a general secure multi-party computation framework, it provides implementations of some of the state-of-the-art secure comparison protocols, like the one from Catrina et al. [6] that works in a prime field, and the one from Damgård et al. [12] that works in a ring. Both of these two protocols perform the secure comparison between two secret integers by extracting the most significant bit. Boyle et al. [5]’s function secret sharing based secure comparison protocol is also another alternative performing secure comparison, and is considered the most efficient protocol by now in terms of online efficiency as it requires only $\ell$ bits communication in one round where $\ell$ indicates the length of the output domain.

Encoding-based approach. There are also other different protocols proposed to find the maximum from a secret set. Zhang et al. [21] introduced a bit-wise protocol for computing the maximum that works from the most significant bit (MSB) to the least significant bit (LSB). This protocol employs special encoding on the client inputs and relies on ciphertexts that encode each bit of a secret integer using a probabilistic scheme. It considers an unreliable aggregator and produces results with an accuracy of at least
TABLE 1: Comparative overview of protocols for securely computing the maximum. This table sets out the communication complexity comparison among our protocols $\Pi_{\text{Max}}$, solutions that employ existing secure comparison protocols from (GSV07, DGK07, Cou18, Catrina10, Damgaard19), and the encoding-based protocols (ZCZ15, Prio, Prio+). Here $m$ is the size of the input set $X$, $n$ the input domain size, $\kappa$ the corresponding security parameter.

<table>
<thead>
<tr>
<th>Ref.</th>
<th>Primitve</th>
<th>Leakage</th>
<th>Total Comm.</th>
<th>Rounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>GSV07 [14]</td>
<td>OT</td>
<td>$O(\min(n + \kappa))$</td>
<td>$O(n)$</td>
<td>$O(\log m)$</td>
</tr>
<tr>
<td>DGK07 [13]</td>
<td>DGK</td>
<td>-</td>
<td>$O(mn/\log \kappa)$</td>
<td>$O(mn)$</td>
</tr>
<tr>
<td>ZCZ15 [21]</td>
<td>-</td>
<td>$O(\kappa mn)$</td>
<td>$O(\kappa mn)$</td>
<td></td>
</tr>
<tr>
<td>Pri0 [9]</td>
<td>0</td>
<td>$O(2^n \kappa)$</td>
<td>$O(2^n \kappa)$</td>
<td></td>
</tr>
<tr>
<td>Prio+ [1]</td>
<td>0</td>
<td>$O(2^n \kappa)$</td>
<td>$O(2^n \kappa)$</td>
<td></td>
</tr>
<tr>
<td>Our $\Pi_{\text{Max}}$</td>
<td>I-DPF</td>
<td>$O(\kappa mn)$</td>
<td>$O(\kappa mn)$</td>
<td>$n + 1$</td>
</tr>
</tbody>
</table>

TABLE 2: Comparative overview of protocols for securely computing the k-th ranked element. The table sets out our protocol against existing methods. Here $m$ is the size of the input set $X$, $n$ the input domain size and $\kappa$ the security parameter. For ZCZ15 $p$ is the output precision. For CHH[22], $S$ denotes the range of database. For our protocol $\Pi_{\text{BatchKre}}$, $\omega$ denotes the batch size. The symbol $\bullet$ indicates the presence of information leakage in the corresponding protocol, on the contrary the symbol $\circ$ indicates no information leakage in the corresponding protocol.

<table>
<thead>
<tr>
<th>Ref.</th>
<th>Primitive</th>
<th>Leakage</th>
<th>Total Comm.</th>
<th>Rounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMP10 [2]</td>
<td>-</td>
<td>$O(m^2 \kappa)$</td>
<td>$O(n)$</td>
<td></td>
</tr>
<tr>
<td>ZCZ15 [21]</td>
<td>-</td>
<td>$O(\kappa mn)$</td>
<td>$O(n)$</td>
<td></td>
</tr>
<tr>
<td>TKK''20 [19]</td>
<td>YGC</td>
<td>$O(\kappa mn)$</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AHE1</td>
<td>$O(\kappa mn^2 \kappa n)$</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AHE2</td>
<td>$O(\kappa mn^2 \kappa n)$</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>CHH''22 [7]</td>
<td>OT</td>
<td>$O(\kappa mn)$</td>
<td>$O(\log S)$</td>
<td></td>
</tr>
<tr>
<td>Our $\Pi_{\text{BitKre}}$</td>
<td>I-DPF</td>
<td>$\sim mn$</td>
<td>$1 + 2n/\omega$</td>
<td></td>
</tr>
<tr>
<td>Our $\Pi_{\text{BatchKre}}$</td>
<td>I-DPF</td>
<td>$\sim mn$</td>
<td>$1 + 4n/\omega$</td>
<td></td>
</tr>
</tbody>
</table>

1 $− \kappa/n$, where $n$ is the integer length and $\kappa$ is the security parameter. However, this approach requires revealing each bit of the maximum to all participants and disallows accidental dropouts during execution. The recently proposed frameworks for secure aggregation Prio [9] and Prio+ [1] focus on providing input verification schemes to contain possible malicious behavior from clients. Prio employs efficient zero-knowledge proofs, namely SNIP. Prio+ uses Boolean secret sharing of the clients’ inputs. Both frameworks also propose protocols that compute the maximum functionality. In their settings, multiple clients submit $M = 2^n$ sized encoding of private integers from small domain $\{0, 1\}^n$ to two computing servers. The maximum can be computed by employing an OR sub-protocol $M$ times non-interactively. However, since the encoding size of the secret input grows exponentially with the size of input domain, their maximum protocol only works efficiently with a relative small input domain (e.g., an input domain capped by a few thousands). To compute the maximum/minimum when dealing with a large input domain, the authors in Prio also proposed a c-approximation variant protocol that uses a similar idea to their original one. More precisely, they divide the input range $\{0, \ldots, B-1\}$ into $b = \log B$ bins, then they use the small-range protocol over all $b$ bins, to compute the approximate statistic.

**Compute the k-th ranked value (KRE) from a secret set.** In Table 2 we present the complexities comparison between our proposed protocols $\Pi_{\text{BatchKre}}, \Pi_{\text{BatchKre}}$, and existing works. Notably the works in [2], [19], [21] and [7] assume the multiple party computation setting. Among all the listed protocols in Table 2 our protocols and [7], [21] have the least communication complexity. The protocols in [7] and [21] involve a multi-party setting and exhibit information leakage during the protocol execution. The
latter is not the case in our protocols. In [7] multiple iterations are involved and comparisons between each local set and public parameter $m_i$ are made in the $i$-th iteration. Here the number of elements greater or smaller than $m_i$ in the union of all databases is revealed.

**Comparison between our protocols and state-of-the-art.** Regarding protocols that compute $\text{F}_{\text{MAX}}$, in Table 1 we compare the communication complexities between our protocol $\Pi_{\text{Max}}$ and works mentioned. When we focus on the efficiency of online evaluation, from Table 1 we see that works in [10] [12] [5] represent state-of-the-art, which requires relatively low online communication volume and rounds. Notably the works in [21] and [7] assume a multiple party computation setting, while Prio [9] and Prio+ [1] consider the outsourcing computation model similar to our setting. Despite these differences we list and compare them to provide a broader overview of the existing literature. Our proposed protocols for secure maximum computation based on I-DPPs (which is one of the latest extension of DPP) are executed in $O(n)$ communication rounds.

Comparing our protocols to those encoding-based protocols in [9] and [1], ours are not as efficient as theirs in terms of communication rounds as the encoding-based protocols perform the online evaluation non-interactively. However, in terms of the communication volume required for every secret input submitted from the clients to each server, our protocol requires only $n$ bits compared to a communication complexity of $O(2^n)$ in theirs, our protocols reduce significant amount of communication volume as well as memory usage used when processing same size of inputs. Thus, for small input domain (bounded by a few thousands), the protocols in [9, 1] might have better online efficiency over ours when the encoded size of client’s input are still reasonable; however, when the the size of input domain increases, e.g. for $n = 20$, $\kappa = 32$, to get an accurate maximum computation using the protocol in Prio it would require $32 \times 2^{20}$ bits (4GB) to represent one secret input on one server which is practically prohibitive.

Our protocols also differ from the works in [10], [13], [14] where communication rounds depend on the set size $m$. Furthermore, our scheme guarantees no information leakage, unlike the scheme in [21] which requires revealing the maximum in clear to all participants in their protocol design.

For a more concrete online evaluation cost comparison with the above three protocols, in Fig. 1 we present concrete comparison results between our protocol $\Pi_{\text{Max}}$ and three comparison-based protocols that compute the maximum of a set where the internal secure comparison protocols are separately instantiated from [10], [12] and [5]. To get the overall costs illustrated in 1, we rely on a single secure comparison evaluation cost reported on these three works, where in [10] the secure comparison protocol SC$_l$ costs 622 bits in 9 rounds for $n = 32$, and 1286 bits in 10 rounds for $n = 64$; in [12] it costs $6n - 8$ bits in $\log(n - 1) + 2$ rounds; and in [5] there is an theoretical optimal cost which is $n$ bits in one round. Thus, for $n = 32$, the overall online cost for computing the maximum using secure comparison from [10] is approximately $2.5nm$ bits in $11 \log m$ rounds; approximately $(8n - 2)m$ bits in $8 \log m$ rounds from [12]; approximately $3nm$ bits in $2 \log m$ rounds from [5]; while in our protocol $\Pi_{\text{Max}}$, it costs approximately $mn$ bits in $n + 1$ rounds. When the input set size $m$ is greater than $2^{16.5}(0.92 \times 10^8)$, as shown in Fig. 1 that our protocol $\Pi_{\text{Max}}$ requires the least online rounds. Furthermore, independently of the value of $m$, our protocol $\Pi_{\text{Max}}$ requires the least communication volume which is a third of that in the protocol introduced in [5] (the best of SOTA).

**A generalization of secure maximum computation.** Compared to the computation of the maximum from a secret set, in a related line of research Aggarwal et al. [2] and Gowri et al. [7] have focused on a generalized computation problem where participants have confidential, ordered data sets and aim to compute the maximum or $k^{th}$ ranked element from their collective data. In their pioneering work Aggarwal et al. [2] introduced the use of secure binary search for identifying the target value, requiring $O(\log N)$ communication rounds, where $N$ is the input domain size. Gowri et al. [7] extended the functionality of this framework allowing more general comparison-based operations such as finding the convex hull of a set of points or job scheduling problems. They employed Oblivious Transfer (OT) for the secure comparison sub-protocol.

## 2. Preliminary

In this paper, we refer to $S_0, S_1$ as the two computing servers in our protocols. We use following notations throughout this paper:

- $\varepsilon$: An empty string.
- $b$: The negation of a bit $b$.
- $[n]$: A set of integers ranging from 1 to $n$ where $n$ is a positive integer.
- $s_i$: The $i^{th}$ bit of a binary string $s$ counting from left to right where $i \leq |s|$.
- $s[i..j]$: The substring from the $i^{th}$ bit to the $j^{th}$ bit of a binary string $s$ counting from left to right where $i, j \leq |s|$.
- $a_i$: The $i^{th}$ element of a vector $\vec{a}$ where $i < |\vec{a}|$.
- $x^b$: 2-out-of-2 Boolean secret sharing over $\mathbb{Z}_2$.
- $x^A$: 2-out-of-2 arithmetic secret sharing over $\mathbb{Z}_N$ for $N \in \{2^p, p\}$ where $p$ is an odd prime.
- $x^b_b$: The Boolean secret share of $x$ held by party $b$ where $b \in \{0, 1\}$.
- $x^A_b$: The arithmetic secret share of $x$ held by party $b$ where $b \in \{0, 1\}$.
- $F_{\text{BC}}(x, \ell) - \text{bits decomposition: Upon receiving an integer } x \text{ and output length specification } \ell, \text{ it converts } x \text{ to its bits representation format } y (\text{padding zeros by the start until } |y| = \ell), \text{ then it outputs } y$.
- $F_{\text{BC}}(s) - \text{bits composition: Upon receiving a binary string } s \text{ of length } \ell, \text{ it outputs an integer } y = \sum_{i=1}^{\ell} 2^{\ell-i} \cdot s[i]$.

### 2.1. Secure Two-party Computation over Additive Secret Sharing

We use the following two-party secure computation functionalities in our protocols:

- $F_{\text{Reveal}} - \text{secret sharing revelation: Upon receiving either a Boolean secret sharing } [b]^b \text{ or an arithmetic}$
secret sharing $[x]^A$, it reveals the corresponding bit $b$ or value $x$ to the two servers.

- $F_{AB}$ - arithmetic to Boolean secret sharing conversion: Upon receiving an arithmetical secret sharing $[x]^A$ where $x \in \{0, 1\}$, it outputs the corresponding Boolean secret sharing $[b]^B$.
- $F_{BA}$ - Boolean to arithmetic secret sharing conversion: Upon receiving a Boolean secret sharing $[b]^B$, it outputs $[b]^A$.
- $F_{NZ}$ - Nonzero check over a secret sharing: Upon receiving an arithmetical secret sharing $[x]^A$, it outputs $[1]^A$ if $x$ is not zero, otherwise it outputs $[0]^A$.

2.2. Incremental Distributed Point Function

We can informally define a point function that has a value of $\beta \in \mathbb{G}$ at a special point $\alpha \in \{0, 1\}^n$ and is zero everywhere else. Naturally this function can be represented as a vector of $2^n$ elements, where only a single element is non-zero. Distributed point functions [4] are a way to secret share this vector among two parties. Distributed point functions consist of two routines. In the Gen$(\alpha, \beta) \rightarrow k_0, k_1$ routine, two keys are produced which represent the secret shares. The individual keys can be evaluated as Eval$(k, x) \rightarrow \mathbb{G}$ giving the value of the secret shared vector at the point $x$. Combining the evaluation results for $k_0$ and $k_1$ will yield the corresponding result of the underlying point function. The advantage of distributed point functions is that the secret shares have the size $O(n)$, while naive secret sharing would yield share of size $2^n$. Importantly, an adversary who learns either $k_0$ or $k_1$ learns nothing about $\alpha$ and $\beta$.

In our case, we need a slightly different functionality than the one provided by point functions. More specifically, we want a function that returns a non-zero value $\beta_\ell$ whenever a query point $x \in \{0, 1\}^k$ is a prefix of $\alpha \in \{0, 1\}^n$. To address this, we introduce All-Prefix Point Functions. These functions are defined by a tuple $(\alpha, (G_1, \beta_1), \cdots, (G_n, \beta_n))$ (shorthand $(\alpha, \beta)$), where $\alpha \in \{0, 1\}^n$ and for every $\ell \in [n]$, $G_\ell$ is the description of an Abelian group and $\beta_\ell \in G_\ell$. The function is then defined as follows:

$$f_{\alpha, \beta} : \bigcup_{x \in \{0, 1\}^k} \bigcup_{e \in \{0, 1\}^n} x \rightarrow \bigcup_{x \in \{0, 1\}^n} G_e,$$ given by

$$f_{\alpha, \beta}(x_1 \cdots x_k) = \begin{cases} 
\beta_\ell & \text{if } x_1 \cdots x_\ell = \alpha_1 \cdots \alpha_\ell, \\
0 & \text{otherwise.}
\end{cases}$$

All-Prefix Point Functions can be visualised as a binary tree with $2^n$ leaves, where there is a single non-zero path, whose nodes have non-zero values $\beta_\ell$.

Incremental Distributed Point Functions (I-DPF) is a class of functions that allows secret sharing of the binary trees of All-prefix Point Functions, similar to Distributed Point Functions. The concept is schematically depicted in Fig. 2. In [3] Boneh et al. first proposed I-DPF to address the secure computation of $\ell$-heavy hitters from a set of strings. Like Distributed Point Functions, there is one routine for key generation, but two routines for evaluation, as we will see in the following.

According to the formal definition of I-DPF provided in [3], a $(2$-party) I-DPF scheme is a tuple of algorithms $(\text{Gen}, \text{EvalNext}, \text{EvalPrefix})$ such that:

- $\text{IDPF.Gen}(1^k, (\alpha, (G_1, \beta_1), \cdots, (G_n, \beta_n)))$ is a PPT key generation algorithm that given $1^k$ (security parameter) and a description $(\alpha, (G_1, \beta_1), \cdots, (G_n, \beta_n))$ of an all-prefix function, it outputs a pair of keys and public parameters $(k_0, k_1, pp = (pp_1, \cdots, pp_n))$, where $pp$ includes the public values $\kappa, n, G_1, \cdots, G_n$.
- $\text{IDPF.EvalNext}(b, st_{b-1}^{\ell}, pp, x_\ell)$ is a polynomial-time incremental evaluation algorithm that given a server index $b \in \{0, 1\}$, secret state $st_{b-1}^{\ell}$, public parameter $pp$, and input evaluation bit $x_\ell \in \{0, 1\}$, it outputs an updated state and output share value: $(st_b^{\ell}, y_b)$.
- $\text{IDPF.EvalPrefix}(b, k_0, pp, x_1 \cdots x_\ell)$ is a polynomial-time prefix evaluation algorithm that given a server index $b \in \{0, 1\}$, secret state $st_{b-1}^{\ell}$, public parameter $pp$, and input evaluation prefix $x_1 \cdots x_\ell \in \{0, 1\}^\ell$, it outputs a corresponding share value $y_b$.

Assuming that the keys are generated according to $(k_0, k_1, pp) \leftarrow \text{IDPF.Gen}(1^k, (\alpha, (G_1, \beta_1), \cdots, (G_n, \beta_n)))$ each party $b \in \{0, 1\}$ can calculate its output share $y_b$ for a prefix $x_1 \cdots x_\ell$ as:

1: $st_0^{\ell} \leftarrow k_b$
2: for $j = 1$ to $\ell$ do
3: $(st_j^{x_b}, y_b) \leftarrow \text{IDPF.EvalNext}(b, st_{j-1}^{x_b}, pp_j, x_j)$
4: end for
5: return $y_b$.

For these output shares derived using IDPF.EvalNext, we require that $y_0^{t_0} + y_1^{t_1} = f_{\alpha, \beta}(x_1 \cdots x_\ell) \equiv 0_{G_\ell}$ holds at all times. This ensures that the reconstructed value is always equal to the output of the original All-Prefix Point Function for the prefix in question.

Similarly, for the output shares $y_b^t$ of party $b \in \{0, 1\}$ output by IDPF.EvalPrefix, we require that $y_0^{t_0} + y_1^{t_1} = f_{\alpha, \beta}(x_1 \cdots x_\ell)$ holds at all times. It is assumed that the keys are generated according to $(k_0, k_1, pp) \leftarrow \text{IDPF.Gen}(1^k, (\alpha, (G_1, \beta_1), \cdots, (G_n, \beta_n)))$.

Regarding security, it is guaranteed that an adversary who learns either $k_0$ or $k_1$ will not gain information about the special point $\alpha$ or the values $\beta_1, \cdots, \beta_n$.

2.3. Conditional Evaluation

CondEval is a cryptographic primitive that was initially introduced in [8], it inputs a Boolean secret sharing
an arithmetic secret share \([x]^{A}\), a function secret sharing of a function \(f\) and a binary operator \(\circ\), outputs the arithmetical secret sharing \([s \circ f(x)]^{A}\), i.e.,

\[
[s \circ f(x)]^{B} \leftarrow \text{CondEval}([b]^{B}, \circ, f, [x]^{A}).
\]

More specifically, CondEval works in the pre-processing model, it contains a Setup algorithm ran in the offline phase and a Eval algorithm ran in the online phase. For a Boolean secret sharing \([s]^{B}\), an arithmetic secret share \([x]^{A}\), a function \(f\), and a binary operator \(\land\), CondEval is realized by

\[
K_{\lambda} \leftarrow \text{CondEval.Setup}(\lambda, 1^{\land}, pp, f) \quad \text{and} \quad [s \land f(x)]^{B} \leftarrow \text{CondEval.Eval}(\lambda, K_{\lambda}, [s]^{B}, [x]^{A})
\]

respectively in the offline and online phase.

In the offline phase, a trusted third party takes the operation \(\land\), a specific function \(f\) can be secret shared, and public parameters \(pp\) to generate and distribute conditional evaluation key shares \(K_{\lambda}\) to two servers. These keys are used for secure computations in the online phase. In the online phase, upon the secret share \([s]^{B}\) and the arithmetic share \([x]^{A}\) are ready, two servers collaboratively perform CondEval.Eval in a single round and obtain \([s \land f(x)]^{B}\). Consequently, some of our protocols leverage CondEval to achieve an optimized design with fewer rounds.

### 3. High-level Overview

In this section, we introduce the high-level computation framework for our proposed protocols that securely determine either the maximum/minimum or the \(k\)-th ranked element of a set \(X\). Before delving into our framework, we first introduce the technique of "incremental prefix counting" proposed by Boneh et al. in [3] that inspired our work, where this technique was originally proposed for computing \(t\)-heavy hitters.

**How to find \(t\)-heavy hitters.** The authors in [3] outline an efficient method for identifying all \(t\)-heavy hitters from a set of strings, here, a \(t\)-heavy hitter is a string (value) that appears more than \(t\) times in a set of strings. The set of strings is made by the inputs of many clients who send their strings (values) to a service provider. The service provider wants to find all \(t\)-heavy hitters without knowing the clients strings. To do this, they use a technique called 'incremental prefix counting'. In this technique each client produces two keys from their string and sends them to two servers. The servers make a list of valid prefixes \(L\). For each prefix \(p_{i}\) in \(L\) the servers make two new prefixes \(p_{i,t} = p_{i}[t]\) for \(t \in \{0,1\}\) and count the occurrence of each \(p_{i,t}\) in the set. If \(p_{i,t}\) appears more than \(t\) times, they add it to \(L\) and remove \(p_{i}\) from \(L\). This way the servers can find the \(t\)-heavy hitters from the set without learning any individual string from the dataset.

Let us consider at a concrete example to understand Boneh et al.’s [3] method better. We have a set \(X\) with ten strings of two bits each: \(X = \{11, 10, 01, 00, 10, 00, 11, 10, 10\}\). We aim to identify all \(t\)-heavy hitters in \(X\) with \(t = 2\).

In the initial step, beginning with the prefix query tree's root (referenced in Fig. 3), the servers evaluate the prefixes ‘0’ and ‘1’ within \(X\). They discover three occurrences of ‘0’ and seven of ‘1’. Since both exceed the threshold \(t = 2\), both are included in the prefix list \(L\), resulting in \(L = \{0, 1\}\) at the end of the first round.

The servers repeat this step. They count how many times ‘0’ is followed by 0 or 1. The count for ‘00’ is 2 and for ‘01’ it is 1. So they only add ‘00’ to \(L\). They do the same for ‘10’ and ‘11’. The count for ‘10’ is 5 and for ‘11’ it is 2. Both are more than or equal to \(t\), so they add them to \(L\). Now \(L = \{00, 10, 11\}\) gives all the \(t\)-heavy hitters in \(X\).

Boneh et al. [3] use I-DPF keys to do ‘incremental prefix counting’ fast. However, this method reveals every intermediate prefix count result to the servers, i.e., referring to Fig. 3, every node value of this prefix query tree of \(X\) is leaked to both servers.

### 3.1. Overview of our Idea

We propose a new framework that can do prefix-count queries without revealing any partial counts to the servers. This makes ‘incremental prefix counting’ more private and secure. In the approach described in the previous paragraph (\(t\)-heavy hitters), the I-DPF key pairs are computed directly from the elements in \(X\). Then, the I-DPF function is evaluated in an algorithmic fashion using the prefixes in \(L\). We construct the I-DPF keys in a different manner. We describe below the intuition behind our approach.

Let us assume that we choose a random value \(\alpha \leftarrow \mathbb{Z}_{2^{n}}\). We derive the I-DPF keys as

\[
((k_{0}, k_{1}), pp) \leftarrow \text{IDPF.Gen}(1^{n}, \alpha, (G_{1}, 1, \cdots, (G_{n}, 1))
\]

and we set the initial state to \(S_{t}^{0} = k_{0}\). Now when we evaluate the I-DPF keys for the first bit we have two cases:

- For \(\alpha^{1} \oplus 0\) we have:
  \[(S_{t}^{0}, [1]^{A}) \leftarrow \text{IDPF.EvalNext}(b, S_{t}^{0}, pp, \alpha^{1} \oplus 0).
\]
- For \(\alpha^{1} \oplus 1\) we have:
  \[(S_{t}^{0}, [0]^{A}) \leftarrow \text{IDPF.EvalNext}(b, S_{t}^{0}, pp, \alpha^{1} \oplus 1).
\]

Let us assume that we add the results from the servers. If we get 0, it means that the bits are different. If we get 1, it means the bits are the same. We can use XOR to check this. For example, if \(x^{1} = 1\) and \(q^{1}\) are the first bits of some values \(x\) and \(q\), then \(x^{1} \oplus q^{1} = 0\) means \(x^{1}\) and \(q^{1}\) are equal, and \(x^{1} \oplus q^{1} = 1\) means \(x^{1}\) and \(q^{1}\) are different. We can do this for the rest of the bits in \(\alpha\), \(x\) and \(q\) in a similar fashion.

The random values \(\alpha\) and \(q\) are used to hide the real inputs (i.e., act as masks). We use a different \(\alpha_{j}\) for each input \(x_{j}\) and a common \(q\) for all of them. The servers get some parts of \([x]^{B}, [\alpha]^{B}\) and \([q]^{B}\) and reveal the string \(t^{j} = q^{j} \oplus x_{j} \oplus \alpha_{j}\) for each input \(x_{j}\). These strings do not reveal anything about the inputs \(x_{j}\) because of the way \(\alpha\) and \(q\) are chosen. The servers then use the IDPF.EvalNext
function on the strings \( t^j \) to find out if \( q_i \) is equal to \( x^j_i \), i.e., they get a Boolean secret sharing of 1 iff \( q^j_i \equiv x^j_i \). Since the same \( q \) is employed for all \( x_j \), it is possible to count how often \( q_i \) appears in the inputs by computing on the strings \( t_j \).

As we will see in the following it is possible to derive the protocols to compute the maximum and \( k \)-th ranked element by combining the prefix counts for \( q_i \) with information about the cardinality of the certain ’helper sets’.

### 3.2. Comparison to Private Heavy Hitters

Our work employs ‘incremental prefix counting’ as the private heavy hitters introduced in [3]. However, there are notable differences in the following three aspects:

- **Targeted Functionality.** In [3], the authors targeted only the computation of subset histograms and identifying multiple private heavy-hitters without explicitly addressing Max/KRE computation. In contrast, our work specifically targets computing the max/kre functionality, yielding a singular output.

- **Information Leakage.** Furthermore [3] allows both servers to learn i) the set of all heavy strings, and ii) the count of strings starting with each heavy string (section 5.1 of [3]). Conversely, in our method, both servers learn nothing related with the input or output.

We emphasize that the computational framework from [3] is unsuitable for computing \( F_{\text{MAX}} \) or \( F_{\text{KRE}} \). In [3], the servers learn every prefix count in the computation of \( \tau \)-heavy hitters, while our method incorporates additional masking over ‘incremental prefix counting’ to hide prefix counts from servers, as detailed in section 3.3. We also enhance concrete online efficiency by minimizing the required online rounds for computing \( F_{\text{MAX}} \); for computing \( F_{\text{KRE}} \), we utilize the recent round-reducing CondEval primitive [8] and introduce a batch variant for further efficiency improvement. Addressing this functionality with optimal performance is an open and challenging problem that is very different from computing the heavy hitters problem.

### 3.3. Our Computation Framework

Now we are ready to present our computation framework, which involves an offline/online model. We assume that both servers are semi-honest. This means all parties follow the protocol correctly, but might try to extract information from the transcript view they see during the protocol execution.

**Offline phase.** Let’s assume the existence of a trusted third party (TTP) \( T \), a data set \( X \) of \( m \) inputs and a domain size \( n \) for each \( x_j \in X \). In the offline phase for each input \( x_j \in X \), \( T \) follows the following:

- It picks a random value \( \alpha_j \in \left< 0, 2^n \right> \) and calculates \([\alpha_j]_B\)\.
- It runs IDPF.Gen\(\{\alpha_j, (G_1, 1), \cdots, (G_n, 1)\}\) for each \( i \in [n] \) using the algorithm from [3]. This gives two keys \((k_{i,0}, k_{i,1})\) and a public parameter \( pp \).
- It chooses a random binary string \( q \in \{0, 1\}^n \) and computes \([q_i]_A\), \( i \in [n] \). Then for each \( b \in \{0, 1\}, T \) sends \( K_b = \{(k_{i,b}, [\alpha_j]_B)_{j \in [n]}, (k_{m,b}, [\alpha_m]_B)\} \) and \( Q_b = \{[q_i]_A, \cdots, [q_i]_b\} \)

**Input:** Input bit \( b \in \{0, 1\} \) identifying server \( S_b \), and selecting inputs \((K_0, Q_b)\) from the offline phase and \([x_j]_b\) where \( j \in [m] \) from the online phase.

**Output:** \([c]_B\) where \( c = f(x) \).

```plaintext
1: for \( i = 1 \) to \( n \) do
2: \([q_i]_B \leftarrow F_{\text{A2B}}([q_i]_A)\)
3: end for
4: for \( j = 1 \) to \( m \) do
5: \(st^0_{j,b} \leftarrow k_{j,b}\)
6: \(t^j \leftarrow F_{\text{reveal}}([q \oplus x_j \oplus \alpha_j]_B)\)
7: end for
8: for \( i = 1 \) to \( n \) do
9: for \( j = 1 \) to \( m \) do
10: \((st^i_{j,b}, [\alpha_j]_B) \leftarrow \text{IDPF.EvalNext}(b, st^{i-1}_{j,b}, pp, t^i_j)\)
11: end for
12: \([\mu]_A \leftarrow \sum_{i=1}^{n} [\beta^i]_A\)
13: if \( i < n \) then
14: \((\delta_i, \ast) \leftarrow \Pi_f(i, [\mu]_A, [q_i]_A, \ast)\)
15: \([c_i]_B \leftarrow [\delta_i]_i + [q_i]_B\)
16: else
17: \([c_i]_B \leftarrow \Pi_f(i, [\mu]_A, [q_i]_A, \ast)\)
18: end if
19: if \( i < n \) and \( \delta_i = 1 \) then
20: for \( j = 1 \) to \( m \) do
21: \((st^i_{j,b}, [\alpha_j]_B) \leftarrow \text{IDPF.EvalNext}(b, st^{i-1}_{j,b}, pp, -t^i_j)\)
22: end for
23: end if
24: end for
25: Outputs \([c]_B \leftarrow [c_1]_B \cdots [c_n]_B\).
```

Figure 4: Our bit-wise computation framework.

to server \( S_b \).

Note that it’s also possible to generate \( K = (K_0, K_1) \) and \( Q = (Q_0, Q_1) \) without the TTP. This requires a two-party secure computation scheme that generating required key pairs among two servers.

**Online phase.** In the online phase, the clients send \([\{x_{1,b}, \cdots, x_{m,b}\}]\) to the server \( S_b \) for each \( b \in \{0, 1\} \). These values correspond to the Boolean secret sharing of a multi-set \( X = \{x_1, \cdots, x_m\} \).

Now we are ready to present our basic framework as illustrated in Fig. 4, which computes the desired functionality \( f \) over \( X \) in a bit-wise manner. In the offline phase server \( S_b \) gets \( K_b \) and \( Q_b \) for each \( b \in \{0, 1\} \). In the online phase server \( S_b \) gets some parts of \( X \) from all clients using Boolean secret sharing. When all inputs are there, the servers compute a masked string \( t^j = \oplus_{j \in [n]} \alpha_j \) (of size \( n \)) for each \( j \in [n] \). This is the input for the I-DPF evaluation functions. Then we use \( c_i \) for the \( i \)-th bit of \( f(X) \) for each \( i \in [n] \). To compute the new target bit \( c_i \), the servers use arithmetic secret sharing to count the prefixes of the string \([c_1]_B \cdots [c_i]_B \) over \( X \) (CountPrefix), which is indicated by \([\mu]_A \) in Fig. 4. Then \([\mu]_A \) is passed to a sub-protocol \( \Pi_f \) which will output a masked bit \( \delta_i = c_i \oplus q_i \) (ComputeBit). In case \( \delta_i = 1 \) and \( i \leq n \) the servers update their I-DPF states (UpdateState). Specifically, for each \( i \in [n] \) the servers do these three steps:
TABLE 3: Extra input/output of $\Pi_{\text{BitMax}}$

<table>
<thead>
<tr>
<th>$i$</th>
<th>Extra Input</th>
<th>Extra Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$[q_1+1]^A$, $[v_{i-1}]^A$, $[w_{i-1}]^A$</td>
<td>$[v_i]^A$, $[w_i]^A$</td>
</tr>
</tbody>
</table>

1) CountPrefix: For each $j \in [m]$, $S_0$ does the following:

- It runs IDPF.EvalNext($b$, $s_{\delta-1}^j$, $pp$, $t_i^j$) to get a new I-DPF state $s_{\delta-1}^j$.
- It also gets $[\beta_i]^A$, which is either 0 or 1.
- It adds up all $[\beta_i]^A$ to get $[\mu_i]^A$ which indicates the amount of the prefix string $[c_1] \cdots [c_{i-1}]$ that appears in $X$.

2) ComputeBit: After obtaining $[\mu_i]^A$, this framework feeds $[\mu_i]^A$ along with $[q_i]^A$ to the corresponding 2PC protocol $\Pi_f$. This gives a masked bit $\delta_i = c_i \oplus q_i$. For the MAX and KRE computations, we use $\Pi_{\text{BitMax}}$ and $\Pi_{\text{BitKre}}$ respectively.

3) UpdateState: If $\delta_i = 1$, it means that the random bit $q_i$ (generated in the offline phase) is different from $c_i$. Then the I-DPF states need to be updated. The servers do this by running IDPF.EvalNext($b$, $s_{\delta-1}^j$, $pp$, $t_i^j$) again for each $j \in [m]$. They get a new state $s_{\delta-1}^j$ for each input. These states count the prefixes of the string $[c_1] \cdots [c_{i-1}]$, which appear in $X$. This step is only performed when $i < n$.

At each iteration the servers get the Boolean secret sharing of a new target bit $c_i$ and the correct I-DPF states for the prefix $[c_1] \cdots [c_i]$. After executing $n$ iterations the protocol outputs the final Boolean secret sharing of the target binary string $[c]^B$. In the following sections we outline the construction and correctness of our protocols for computing the maximum and $k$-th ranked element. For brevity, the formal security proofs are included in Appendix A.

4. Bit-wise Constructions

In this section, we present sub-protocols designed for computing the $i$-th bit of the maximum and the k-th ranked element of $X$. While for the sub-protocol that compute the $i$-th bit of the minimum of $X$, we attach it in Appendix C.1.

4.1. Realizing $\Pi_{\text{BitMax}}$

We present $\Pi_{\text{BitMax}}$, (described in Protocol 1) a protocol that securely determines the $i$-th bit of the maximum value in $X$. The interface for $\Pi_{\text{BitMax}}$ is described as:

$$(\delta_i, [c]^B, *) \leftarrow \Pi_{\text{BitMax}}(i, [\mu]^A, [q_i]^A, *)$$

To use this interface it is required to provide at least $i$ and $[\mu]^A$ as inputs. At least $\delta$ and $[c]^B$ will be returned as outputs. There may be other inputs or outputs which are indicated by the symbol $*$. The exact interface of $\Pi_{\text{BitMax}}$ depends on the value of $i$. These variations are detailed in Tab. 3.

Our goal is to find the $i$-th bit of the maximum of $X$ in Protocol 1. We use two variables $[v]^A$ and $[w]^A$ to keep track of the progress. $[v]^A$ is the number of values in $X$ that could be the maximum, and $[w]^A$ is the product of $[v]^A$ and the probability that the $i$-th bit of the maximum is 0, which we denote by $[q_i+1]^A$. We start with $[v]^A = SS\text{share}(m)$, where $m$ is the size of $X$. We update both $[v]^A$ and $[w]^A$ whenever we determine a new bit of the maximum.

Correctness. We establish the correctness of Protocol 1 as follows. In the $i$-th iteration where $i \in [n]$ let $p = c_1 \cdots c_{i-1}$ denote the prefix of the maximum of $X$ up to the $i$-1 bit. Recall that $[\mu^A]$ represents the count of strings in $X$ that have the prefix $p|q_i$. We then distinguish between two cases based on the actual value of $q_i$:

- $q_i = 0$: We have $[v]^A = [v]^A \cdot (1 - [q_i]^A) = [v]^A$. When invoking $F_{\text{NZ}}([\mu^A] - [w]^A)$ the nonzero check is performed on $[\mu^A] - [w]^A$. If this check passes, it indicates that not all remaining maximum candidates start with the prefix $p|0$. In simpler terms, at least one string starts with the prefix $p|1$.

In both scenarios, if the corresponding non-zero check passes, we can be certain that the $i$-th bit of the maximum is 1. By analogy if the non-zero check does not pass the $i$-th bit is 0, independently of the value of $q_i$. Therefore Protocol 1 accurately computes the Boolean secret sharing of the maximum value of $X$ after $n$ iterations. Note that when $\delta$ is revealed, both $[v]^A$ and $[w]^A$ are updated correctly. More specifically if $\delta = 0 (c_i \equiv q_i)$, then $[v]^A$ and $[w]^A$ are correct. Conversely if $\delta = 1 (c_i \equiv 1 - q_i)$, then $[v]^A$ and $[w]^A$ are correct and $[v]^A - [w]^A$ tells the amount of strings that start with $p|1 - q_i = c_1 \cdots c_{i-1}|1 - q_i$, which is the amount of all candidates of the maximum by the $i$-th iteration.

Protocol 1 Bit-wise maximum protocol-$\Pi_{\text{BitMax}}$

Functionality: $(\delta_i, *) \leftarrow \Pi_{\text{BitMax}}(i, [\mu]^A, [q_i]^A, *)$

Input: $[\mu]^A$ and $[q_i]^A$, possibly other inputs based on the value of $i$.


1: if $i = 1$ then
2: $[v]^A \leftarrow SS\text{share}(m)$
3: $[w]^A \leftarrow m - 1 - [q_i]^A$
4: end if
5: $[c]^B \leftarrow F_{\text{NZ}}([\mu] - [w]^A)$
6: if $i < n$ then
7: $[v]^A \leftarrow [\mu]^A$
8: $[w]^A \leftarrow [\mu]^A$
9: $[v]^A \leftarrow [v]^A \cdot (1 - [q_{i+1}]A)$
10: $[w]^A \leftarrow [v]^A \cdot [v]^A$
11: end if
12: if $i < n$ then
13: $\delta_i \leftarrow F_{\text{reveal}}([c_i] \oplus [q_i]^B)$
14: $[v]^A \leftarrow [v]^A$
15: $[w]^A \leftarrow [w]^A$
17: else
18: Outputs $[c]^B$.
19: end if

4.1.1. Rounds Optimization on Protocol 1. In Protocol 1, assuming $F_{\text{NZ}}$ is instantiated by function secret sharing in the pre-processing model that requires only a
single communication round, this gives us a total communication round of \(2n\) for computing the maximum of \(X\) within the computation framework. However, for each \(i \in [1, \ldots, n-1]\), we observe that it’s possible to further reduce the overall communication rounds by combining the \(F_{\text{reveal}}\) operation from the \(i\)th iteration with the \(F_{\text{NZ}}\) from the next \((i+1)\)th iteration, enabling a cost of mere \(n+1\) communication rounds computing the maximum of \(X\).

For this to work, we propose a modification that involves pre-computing additional prefix counts over \([X]^B\) before invoking Protocol 1. Specifically, instead of limiting ourselves to computing the prefix count of \(q_1\) over \([X]^B\), we expand our computation to include the prefix counts of \(q_1\), \(-q_1\), \(q_2\), and \(-q_1\) over \([q_2]^B\), respectively, thus obtaining \([\mu]^A\), \(\varrho_1^A\), \(\varrho_0^A\), and \(\varrho_1^A\) along with associated internal I-DPF evaluation states. With these preliminary computations, the execution of Protocol 1 becomes more efficient. In the second round, alongside performing \(F_{\text{reveal}}\) as outlined in Protocol 1, the servers undertake additional tasks:

- They compute two candidate bits, one of which will indicate the second comparison bit, through:
  \[
  \begin{align*}
  [C]_i^B &\leftarrow F_{\text{NZ}}([\mu_0]^A - [w_0]^A) \quad \text{and} \\
  [C]_i^B &\leftarrow F_{\text{NZ}}([\mu_1]^A - [w_1]^A).
  \end{align*}
  \]
- They also prepare a list of candidate terms for the third comparison bit computation using \(F_{\text{NZ}}\), which include:
  \[
  \begin{align*}
  [w_0]^A &\leftarrow (1 - [g_3]^A)[\mu_0]^A, \\
  [w_1]^A &\leftarrow (1 - [g_3]^A)[\mu_1]^A, \\
  [w_1]^A &\leftarrow (1 - [g_3]^A)[\mu - \mu_0]^A, \\
  [w_1]^A &\leftarrow (1 - [g_3]^A)[\mu - \mu_1]^A.
  \end{align*}
  \]

Upon the second round’s completion and the revelation of \(\delta_1\), the servers adjust their computation of the prefix count over \(c_1\|q_2\|q_3\) and \(c_1\|\neg q_2\|q_3\) based on \(\delta_1\):

- if \(\delta_1 = 0\), from the I-DPF states held by evaluating \(q_1\|q_2\), servers do one more bit prefix count over \(q_1\|q_2\|q_3\), and from the I-DPF states by evaluating \(q_1\), servers do two more bits prefix count over \(q_1\|\neg q_2\|q_3\);
- otherwise, from the I-DPF states held by evaluating \(-q_1\|q_2\), servers do one more bit prefix count over \(-q_1\|q_2\|q_3\), and from the I-DPF states by evaluating \(-q_1\), servers do two more bits prefix count over \(-q_1\|q_2\|q_3\).

By following the established naming convention, we denote the obtained prefix count results over \(c_1\|q_2\), \(c_1\|\neg q_2\|q_3\), and \(c_1\|\neg q_2\|q_3\) as \([\mu]^A\), \([\varrho_1]^A\), \([\varrho_0]^A\), and \([\varrho_1]^A\) respectively.

In the third round, the servers reveal the second masked comparison bit \([c_{\delta_1} \oplus q_2]^B\), denoted as \(\delta_2\), and perform two tasks similar to the previous iteration, aiming to compute candidate bits and prepare terms for subsequent comparison bit computations. Servers repeat above procedure until the \((n-1)\)th round, however by the last \(n\)th iteration, servers need only reveal \(\delta_{n-1}\) and compute two candidate comparison bits. Without any further communication, this gives us \([c_{\delta_{n-1}}]^B\), which exactly represents the final target comparison bit.

This optimized methodology is outlined in detail in Tab. 4. Compared to the original Protocol 1, this optimization reduces the communication rounds from \(2n\) to \(n+1\), albeit with an increase in computation overhead due to the additional prefix counts and the slightly higher costs associated with secret sharing-based equality checks and multiplication operations. Applying this round optimization technique on sub-protocol 1, and integrate this resulting protocol variant within the computational model in Fig. 4, results in our finalized protocol for computing \(F_{\text{Max}}\), which we refer to as \(\Pi_{\text{Max}}\). The communication complexity of \(\Pi_{\text{Max}}\) involves exactly \((m+1)n + 10kn - 11k\) bits, distributed over \(n + 1\) rounds, incurring a computational cost of \(3mn\) invocations of I-DPF.EvalNext on each server.

### 4.2. BitMax to BitKre

Secure maximum computing and the \(k\)-th ranked element computation are two closely related functionalities. The computation of the maximum can be generalized to the computation of the first ranked element, but not vice versa. Thus, in the same setting the \(k\)-th ranked element computation often comes with higher cost than the maximum computation. In our proposed protocols, the \(k\)-th ranked element computation protocol 2 has slightly more computational cost than the maximum computation Protocol 1.

Before discussing the details of our bit-wise \(k\)-th ranked element computation protocol in Protocol 2 let us again refer to the example in Fig. 3. Remember that the nodes in this graph represent the prefix counts. Assume \(k = 6\), let us explore how to find the \(k\)-th ranked element (in descending order) in set \(X\) based on Fig. 3. We proceed as follows.

In the first step, we execute a prefix query for the bit string 1 and obtain a result of 7, which exceeds \(k\). Therefore, we leave \(k\) unchanged and determine that the first bit of the \(k\)-th ranked element is 1. Moving to the next iteration, the protocol performs a prefix query for 11 and receives a result of 2, which is less than \(k\). As a result, \(k\) is updated to \(k = k - 2 = 4\) and the correct bit is determined to be 0.

This example serves as a useful guide for comprehending the protocol outlined in Protocol 2. The protocol adopts the same approach we have previously described, but conducts each operation—including comparisons and internal value updates—in a secure manner.

### 4.3. Realizing \(\Pi_{\text{BitKre}}\)

In Protocol 2, we introduce \(\Pi_{\text{BitKre}}\), which securely computes the \(i\)th bit of the \(k\)-th ranked element of \(X\). The interface for \(\Pi_{\text{BitKre}}\) is described as:

\[
(\delta_i, *) \leftarrow \Pi_{\text{BitKre}}(i, [\mu]^A, [g_i]^A, [k]^A, *).
\]

This interface requires at least inputs of \(i\) and \([\mu]^A\) and outputs \(\delta_i\). The symbol * denotes potential additional input/output, and the precise interface of are detailed in Tab. 5. The scalar \(v\) represents the number of remaining target value candidates and is initially set to \(m\). At a high level, the protocol aims to achieve two objectives:
Given that our secure computation framework mandates then executing the above operations would take two In a typical scenario, converting from the left and right branches of the current prefix query tree

c\] ensures that the revelation of a bit \( \delta_i \) \( \equiv c_i \oplus q_i \), we can do both the revelation of \( \delta_i \) and the computation of

\[
\begin{align*}
& t_1^A = [q_i]^A \cdot [r]^A, \\
& t_2^A = [q_i]^A \cdot [r]^A.
\end{align*}
\]

in one round. We can express \( c_i \) as \( \delta_i \oplus q_i = \delta_i + q_i - 2\delta_i \cdot q_i \).

Substituting this expanded form of \( c_i \) and \( [t_1]^A \), \( [t_2]^A \) into the previous operations, we get the following equations for updating \([k]^A\) and \([v]^A\):

\[
\begin{align*}
[k]^A &= [k]^A + (\delta_i - 1) \cdot [r]^A + (1 - 2\delta_i) \cdot [q_i]^A \cdot [r]^A \\
&= [k]^A + (\delta_i - 1) \cdot [r]^A + (1 - 2\delta_i) \cdot [t_2]^A, \\
[v]^A &= (1 - \delta_i) \cdot [l]^A + (2\delta_i - 1) \cdot [q_i]^A \cdot [r]^A \\
&= (1 - \delta_i) \cdot [l]^A + (2\delta_i - 1) \cdot [t_1]^A + \delta_i \cdot [r]^A + (1 - 2\delta_i) \cdot [t_2]^A.
\end{align*}
\]

This optimization streamlines the update process for \([k]^A\) and \([v]^A\) in Protocol 2, effectively reducing the number of communication rounds required.

Protocol 2 Bit-wise k-th ranked element computation


Input: An index \( i \in [n], [\mu]^A, [\mu]^A \) and the target ranking \([k]^A\), it also inputs \([v]^A\) if \( i > 1 \).

Output: A bit \( \delta_i \) and the updated \([v]^A\), \([k]^A\); otherwise \([c_i]^B\).

1: if \( i = 1 \) then
2: \( [v]^A \leftarrow F_{\text{Share}}(m) \)
3: end if
4: \( [q_i]^B \leftarrow F_{\text{Share}}([v]^A) \)
6: \( [r_1]^A \leftarrow [\mu]^A \)
7: \( [c_i]^B \leftarrow \text{CondEval}([q_i]^B, \land, f_c([k]^A, [r_1]^A)) \)
8: \( [c_i]^B \leftarrow \text{CondEval}([1 - q_i]^B, \land, f_c([k]^A, [r_0]^A)) \)
9: \( [c_i]^B \leftarrow [c_i]^B \oplus [c_i]^B \)

10: if \( i < n \) then
15: \( \delta_i \leftarrow F_{\text{Share}}([c_i]^B \oplus [q_i]^B) \)
16: \( [k]^A \leftarrow [k]^A + (\delta_i - 1) [l]^A + (1 - 2\delta_i) [t_2]^A \)
17: \( [v]^A \leftarrow (1 - \delta_i) [l]^A + (2\delta_i - 1) [t_1]^A + \delta_i [r]^A + (1 - 2\delta_i) [t_2]^A \)
18: Outputs \( (\delta_i, [k]^A, [v]^A) \).
19: else
20: Outputs \([c_i]^B\).
21: end if

Correctness. The correctness of our Protocol 2 is already shown when we present the construction above.

Efficiency. Integrating protocol 2 within the computational framework in Fig. 4 results in our finalized protocol.

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{1st Round} & \text{2nd Round} & \text{3rd Round} & \cdots & \text{nth Round} \\
\hline
[c_i]^B & F_{\text{NZ}}([\mu - w]^P) & \delta_1 \leftarrow F_{\text{Share}}([c_i \oplus q_1]^P) & \cdots & \delta_{n-1} \leftarrow F_{\text{Share}}([c_i \oplus q_n]^P) \\
[w_0]^A & \leftarrow [v_0]^A \cdot [1 - q_2]^A & [c_0]^B & F_{\text{NZ}}([\mu_0 - w_0]^P) & [c_0]^B \leftarrow F_{\text{NZ}}([\mu_0 - w_0]^P) \\
[w_1]^A & \leftarrow [v_1]^A \cdot [1 - q_2]^A & \cdots & \cdots & \cdots \\
\hline
\end{array}
\]

\[
\text{TABLE 5: Extra input/output of } \Pi_{\text{BitKre}}
\]
for computing $F_{KRE}$, which we refer to as $\Pi_{\text{BitKre}}$. In Protocol 2 the first round corresponds to the operations in lines 7,8,11, while the second round corresponds to the computation of $[t_0]^A, [t_2]^A$ in lines 13,14,15. Thus, $\Pi_{\text{BitKre}}$ involves exactly $(m+1)n + 8(n-1)\kappa + 2(L+1+\kappa)$ bits, distributed over 2n rounds, incurring a computational cost of $3nm/2$ invocations of IDPF.EvalNext on average on each server, here $L$ denotes passed decryption key length when evaluating CondEval.

5. Batch-wise Variants

Our bit-wise protocols compute the target value from the input domain of size $n$, at the communication complexity of $O(n)$, which is independent on the size of the secret input set $m$. This is different with comparison-based approaches, where amount of communication rounds is affected by $m$ (see tables 1 and 2). This property enhances the scalability of our protocols for dataset $X$, making them more adaptable to larger data sets where the input domain size does not hinder performance. However, the communication rounds of the bit-wise protocols can still be inefficient in some cases. For example if $n$ is big or the network has high latency, the total communication time could become a performance bottleneck.

Therefore we want to know how to further reduce the number of communication rounds required for our target functionality. We remind ourselves that in the original computation framework from Fig. 4 that a single bit is computed per iteration. Thus, a straightforward intuition is try to compute multiple target bits, i.e., a batch in a lower number of iterations. We denote by $w$ the batch size and assume $n$ is a multiple of $w$. Thus, if we compute the target string batch-wisely, with a lower number of iterations $d = n/w$ required, we could reduce the communication complexity from $O(n)$ to $O(d)$, potentially reducing communication rounds required in the end.

We denote by $[\vec{v}]^A$ a resulting prefix count vector of a prefix list $p\|((\psi \oplus F_{\text{BDC}}(2^\omega - 1, \omega)), \cdots, p\|((\psi \oplus F_{\text{BDC}}(0, \omega))$ over $X$ for $\psi = g_{q-1}g_{q-2} \cdots g_1$, where $\psi$ is the $q_{\text{th}}$ batch of the random string $q$, thus, $[\vec{v}]^A$ indicates an unordered prefix count vector. However, for meaningful computation with these values, we need somehow convert this unordered vector $[\vec{v}]^A$ to an ordered vector $[\vec{v}]^A$, by ordered vector we mean a resulting vector over an ordered prefix list

$$p\|((F_{\text{BDC}}(2^\omega - 1, \omega), \cdots, p\|((F_{\text{BDC}}(0, \omega)).$$

To realize this conversion, the trick is to employ a conversion matrix $M_i$ within each iteration $i \in d$. The conversion matrices $M_i$ are prepared during the offline phase from $\psi$. We outlined the details of generating $M_i$ in Fig. 6, where the resulting $M_i$ contains $\tau$ rows, each representing a one-hot vector of size $\tau$. In a one-hot vector only one element is one, while all other elements are zero. For instance let us assume $\omega = 2, \tau = 4$ and $\psi = 1[0]$. Then the corresponding matrix $M_{10}$ is:

$$\begin{bmatrix}
0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0
\end{bmatrix}.$$

These matrices are secret shared with the two servers as $[M_i]^A$, which implies that the servers cannot infer any information about $q$ from them. With $[M_i]^A$ ready in the $i^{\text{th}}$ iteration, it allows the servers to convert the prefix count results in each iteration from an unordered state to an ordered state, i.e., from $[\vec{v}]^A$ to $[\vec{v}]^A$. Specifically, this is done by computing

$$[\vec{v}]^A \leftarrow [\vec{v}]^A \times [M_i]^A.$$

Now with this insight, we introduce a batch-wise framework as depicted in Fig. 5, consisting of the following key processes:

1) **Initialization**: The process begins with initializing the I-DPF states, detailed in line 6, followed by revealing the vectors $t_j$ in line 7.

2) **Iterative Computation**: For each $i \in d$, the following steps are executed:

   a) Computation of the prefix count vector $[\vec{v}]^A$ for the unordered prefix list, covered in lines 10-17.

   b) Invocation of the function $\Pi_f$ with $[\vec{v}]^A$ and $[M_i]^A$ and other parameters passed in line 18, which computes and returns the masked target batch $\delta_i$.

   c) The I-DPF states are then updated in lines 19-24 to align with the prefix $c_1\|\cdots\|c_{\omega}$, based on the value of $\delta_i$.


   A notable aspect of this framework is its computational demand, which intensifies with an increase in $\omega$. This surge in complexity is attributed to the exponential growth in the number of potential outcomes $\tau = 2^\omega$, underscoring a trade-off between execution efficiency and communication complexity.

5.1. Realizing $\Pi_{\text{BatchKre}}$

We introduce $\Pi_{\text{BatchKre}}$ which securely computes the $i^{\text{th}}$ batch of the KRE of $X$, the interface for $\Pi_{\text{BatchKre}}$ is defined as:

$$((\delta), \leftarrow \Pi_{\text{BatchKre}}(i, [\vec{v}]^A, [M_i]^A, [q]^B, [k]^A, *))$$

It interface requires at least inputs of prefix query result vector $[\vec{v}]^A$, conversion matrix $[M_i]^A$, a partial bit mask $[q]^B$ and the value $[k]^A$, outputs a masked target batch $\delta$ and an updated $[k]^A$ for $i < d$. For the full details of $\Pi_{\text{BatchKre}}$, please refer it to Appendix B.

Integrating protocol 3 within the computational framework in Fig. 5 results in our finalized protocol for computing $F_{KRE}$, which we refer to as $\Pi_{\text{BatchKre}}$. Note that it requires four rounds in Protocol 3, the first round involves converting $[\vec{v}]^B$ to $[\vec{v}]^A$, the second round is to perform $L_{\text{LessEqualThan}}$ check. The third round computes $[b_t]^B$ for $t \in [\tau]$, which aids in determining the target batch and in the final round $\sigma$ is revealed. Thus, $\Pi_{\text{BatchKre}}$ involves exactly $(m+1)n + \frac{2n}{\omega}(\tau + 2\tau^2 + 2(\tau - 1) + 4(\frac{\omega}{\omega} - 1))$ bits, distributed over $1 + (4m)/\omega$ rounds, incurring a computational cost of $(2^\omega n m)/\omega$ invocations of IDPF.EvalNext on average on each server.
6. Experimental Evaluation

In this section, by performing a detailed experimental evaluation, we report the comparison results of our finalized protocols $\Pi_{\text{Max}}$, $\Pi_{\text{BitKre}}$, and $\Pi_{\text{BatchKre}}$ to the state-of-the-art solutions.

**Experiment Setting.** Except for $\Pi_{\text{ICMax}}$ from MP-SPDZ [17] that is realized in C+++, all other protocols (including basis of our finalized protocols) are realized in Rust and can be found at Github\(^1\). All our experiments are performed on a server equipped with 32GB RAM, 12th Gen Intel(R) Core(TM) i7-12700K CPU model that ran Ubuntu 22.04 LTS. For the evaluation of the offline phase, we assume the existence of a Trustful Third Party (TTP) that distributes correlated randomness to the two computing servers, and we measured the offline phase overhead by running the corresponding offline phase key generation algorithm. For the online phase evaluation, we established a simulated WAN network within the above server. The round-trip time (RTT) latency and bandwidth are set to 80ms and 285Mbps, respectively.

6.1. Evaluation for $\mathcal{F}_{\text{MAX}}$

SOTA implementations for $\mathcal{F}_{\text{MAX}}$. As mentioned in the related work, to compute the maximum from a secret set of size $m$, state-of-the-art solutions employ a secure comparison protocol internally, with which they perform pair-wise comparisons in $O(\log m)$ iterations to determine the maximum. Here, the concrete efficiency depends on the underlying secure comparison protocol being used. To have a good efficiency understanding of such solutions, we employ two secure integer comparison protocols, respectively from [6] and [12], both are already implemented in MP-SPDZ [17]. Both of these two protocols compute the secure comparison result by extracting the most significant bit of the subtraction result of two integers being compared. However, the protocol from [6] works in a finite field $\mathbb{F}$, while the protocol from [12] in a ring $\mathbb{R}$. Additionally, we also implemented the recent secure integer comparison protocol in [5] from scratch, which works in a group $\mathbb{G}$.

Thus, from the state-of-the-art secure comparison protocol in [12], [6] and [5], we devised three basis, respectively denoted as $\Pi_{\text{ScMax1}}$, $\Pi_{\text{ScMax2}}$ and $\Pi_{\text{ICMax}}$. Different with our protocol $\Pi_{\text{Max}}$ that inputs/outputs Boolean secret sharing, $\Pi_{\text{ScMax1}}$, $\Pi_{\text{ScMax2}}$ and $\Pi_{\text{ICMax}}$ input the arithmetical secret sharing $[X]^A$ of set $X$ and output the arithmetical sharing of the maximum of $X$. Most of the source codes for $\Pi_{\text{ScMax1}}$ and $\Pi_{\text{ScMax2}}$ are readily available in MP-SPDZ [17], respectively can be found in the semi2k protocol and the semi-party protocol, however, on top of MP-SPDZ’s source code we add a python script describing the pair-wise comparison based maximum computation procedure to completely realize $\Pi_{\text{ScMax1}}$ and $\Pi_{\text{ScMax2}}$. For the complete implementation of $\Pi_{\text{ICMax}}$, on top of our interval containment function secret sharing implementation of the secure integer comparison protocol in [5], we implemented the same high-level pair-wise comparison based maximum computation algorithm as in $\Pi_{\text{ScMax1}}$ and $\Pi_{\text{ScMax2}}$.

We compared three existing solutions, $\Pi_{\text{ScMax1}}$, $\Pi_{\text{ScMax2}}$, and $\Pi_{\text{ICMax}}$, with our protocol $\Pi_{\text{Max}}$. The final comparison results are displayed in Tab. 6. In this table, the input domain of $X$, denoted as $n$, comprises 31 bits, and the size of the set, $m$, varies from $10^3$ to $5 \times 10^5$. For a fair comparison, in our implementation we have all protocols operate on a 32 bit ring except for $\Pi_{\text{ScMax2}}$, which works in a prime field of length $32 + \lambda$, where security parameter $\lambda$ is defined in [6] and aims to provide statistical privacy for the underlying secure comparison protocol, in our test of $\Pi_{\text{ScMax2}}$ we use the default setting of MP-SPDZ [17] where $\lambda = 40$. Additionally, for the optimal performance in $\Pi_{\text{ScMax1}}$ that we turned on optimization options use_split and use_edabit, here use_split enables local arithmetic-binary share conversion, and use_edabit further improve online efficiency for non-linear functionality. We set the security parameter $k$ as 128 for the instantiating of function secret sharing universally through all protocols.

**Offline phase comparison.** Considering the fundamental similarity in the utilization of function secret sharing by both our protocol, $\Pi_{\text{Max}}$, and the naive protocol, $\Pi_{\text{ICMax}}$, each assuming the presence of a Trusted Third Party (TTP) responsible for distributing function secret sharing key pairs and other correlated randomness to two computing servers—we restrict our comparison to an offline phase. Specifically, we compare $\Pi_{\text{Max}}$ and $\Pi_{\text{ICMax}}$ in terms of the time required for all the key

---

\(^{1}\) https://github.com/nann-cheng/FSS-KRE
generation and the final key size hold by each server, including corresponding function secret sharing key shares and associated correlated randomness. In above two protocols, the primary overhead in the offline phase for these two protocols stems from the generation and storage of function secret sharing keys. In \( \Pi_{\text{Max}} \), these are I-DPFs, while in \( \Pi_{\text{ICMax}} \), they are interval containment function secret sharing keys. Both protocols require roughly \( m \) function secret sharing keys for a later online input set \( X \) of size \( m \), leading to a linear increase in both offline key generation time and key size with the increasing of \( m \). However, as indicated in Tab. 6, \( \Pi_{\text{Max}} \) exhibits slightly better than \( \Pi_{\text{ICMax}} \) in terms of both key generation time and key size.

Let \( \lambda \) represent the seed size, \( n \) the input domain size, and \( \ell \) the output domain size. Based on the construction of I-DPF from [3], and the construction of an integer interval containment (IC) gate from [5] that we used in \( \Pi_{\text{ICMax}} \), we provide the theoretical analysis of their key generation efficiency disparity as follows.

- A single key share of I-DPF is of \( \lambda + n(\lambda + 2 + \ell) \) bits, while a single key share of IC is of \( \lambda + n(\lambda + 2 + \ell) + 2\lambda \) bits. Consequently, \( \Pi_{\text{Max}} \) requires \( 2\lambda \) bits less for each function secret sharing key compared to that in \( \Pi_{\text{ICMax}} \).

- Moreover, within the distributed key comparison function (DCF) secret sharing design that serves as the foundation of the interval containment gate design, it necessitates two invocations of a pseudo-random generator \( G : \{0, 1\}^\lambda \rightarrow \{0, 1\}^{2(\lambda + 1)} \) and three invocations of a pseudo-random group element converter \( \text{ConverG}_\lambda \) in contrast to two invocations of a lighter pseudo-random generator \( G : \{0, 1\}^\lambda \rightarrow \{0, 1\}^{2\lambda + 2} \) and one invocation of a pseudo-random group element converter \( \text{ConverG}_\lambda \) in I-DPF from [3].

Hence, it is evident that the IC key generation is more costly than the I-DPF key generation in terms of both key size and computation time. This observation is further supported by our experimental results in Tab. 6.

**Online phase comparison** Next, we compare the online phase costs among \( \Pi_{\text{ScMax}1}, \Pi_{\text{ScMax}2}, \Pi_{\text{ICMax}}, \) and \( \Pi_{\text{Max}} \). In Table 6, we record the actual communication rounds/volume and total online runtime of our benchmarks of all four protocols with varying input size \( m \). For protocol \( \Pi_{\text{ScMax}1} \) and \( \Pi_{\text{ScMax}2} \), we are limited to testing its performance over an input set \( X \) of size up to \( 10^5 \), as the program compiling hangs after more than ten minutes waiting when we attempt to test them with \( m = 10^6 \).

From the results shown in Table 6, it is evident that \( \Pi_{\text{Max}} \) requires the least communication volume, roughly a third of that required in \( \Pi_{\text{ICMax}} \) and one-eighth of that required in \( \Pi_{\text{ScMax}1} \). This is consistent with our analysis presented in the related work. Additionally, in terms of concrete communication rounds required, our protocol \( \Pi_{\text{Max}} \) requires \( n + 1 = 32 \) rounds regardless of the value of \( m \), while both \( \Pi_{\text{ScMax}1} \) and \( \Pi_{\text{ScMax}2} \) require \( 8\lceil \log(m) \rceil \) rounds and \( \Pi_{\text{ICMax}} \) needs \( 2\lceil \log(m) \rceil \) rounds. Lastly, we compare the online runtime of these four protocols and make the following observations and explanations. In our evaluation, \( \Pi_{\text{ScMax}1} \) and \( \Pi_{\text{ScMax}2} \) were found to perform the least effectively; necessitating a higher volume of communication and significantly more rounds compared to the others. Interestingly, when comparing \( \Pi_{\text{ScMax}1} \) with \( \Pi_{\text{ScMax}2} \), we observed that the operation domain has a small impact on the final online run time, as both protocols exhibit similar performance given the same inputs.

By comparing \( \Pi_{\text{ICMax}} \) to \( \Pi_{\text{Max}} \), as shown in Table 6, it is evident that the **final performance of two protocols differs on the actual value of input set size \( m \)**. Here, since the communication volume required in both protocols is relatively small, we assume that the final practical online evaluation efficiency is mostly affected by two factors: communication rounds and local computation time, *i.e.*, the more communication rounds, the more online evaluation time required; and the more local computational overhead, the more online evaluation time required. Before going into further analysis, from a theoretical point of view we present the run-time cost differences in terms of computation overhead within protocol \( \Pi_{\text{ICMax}} \) and \( \Pi_{\text{Max}} \). The evaluation algorithm of IC used in \( \Pi_{\text{ICMax}} \) requires \( 2\lambda \) invocations of a pseudo-random generator \( G : \{0, 1\}^\lambda \rightarrow \{0, 1\}^{2(\lambda + 1)} \), \( 2\lambda \) invocations of a pseudo-random group element converter \( \text{ConverG}_\lambda \) plus other arithmetical operation cost denoted as \( T(2\lambda) \); in contrast to \( 3\lambda \) invocations of a lighter pseudo-random generator \( G : \{0, 1\}^\lambda \rightarrow \{0, 1\}^{2\lambda + 2} \), \( 3\lambda \) invocation of a pseudo-random group element converter \( \text{ConverG}_\lambda \) and other arithmetical operation cost denoted as \( T(3\lambda) \) in our evaluation of I-DPF used in \( \Pi_{\text{Max}} \). Considering both a pseudo-random generator \( G \) and a pseudo-random group

### Table 6: Comparison of offline/online costs for protocols \( \Pi_{\text{ScMax}1}, \Pi_{\text{ScMax}2}, \Pi_{\text{ICMax}}, \) and \( \Pi_{\text{Max}} \).

<table>
<thead>
<tr>
<th>Phase</th>
<th>Measurements</th>
<th>Ref.</th>
<th>( 10^4 )</th>
<th>( 10^5 )</th>
<th>( 10^6 )</th>
<th>( 2 \times 10^6 )</th>
<th>( 5 \times 10^6 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offline</td>
<td>KeyGen Time (s)</td>
<td>( \Pi_{\text{ICMax}} ) [5]</td>
<td>0.008</td>
<td>0.093</td>
<td>0.864</td>
<td>8.465</td>
<td>16.61</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Our ( \Pi_{\text{Max}} )</td>
<td>0.007</td>
<td>0.064</td>
<td>0.611</td>
<td>5.573</td>
<td>12.122</td>
</tr>
<tr>
<td></td>
<td>Key Size/Server (MB)</td>
<td>( \Pi_{\text{ICMax}} ) [5]</td>
<td>0.74</td>
<td>7.07</td>
<td>70.41</td>
<td>703.8</td>
<td>1407.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Our ( \Pi_{\text{Max}} )</td>
<td>0.74</td>
<td>7.07</td>
<td>70.41</td>
<td>703.8</td>
<td>1407.6</td>
</tr>
<tr>
<td>Online</td>
<td>Rounds</td>
<td>( \Pi_{\text{ScMax}1} ) [12]</td>
<td>80</td>
<td>112</td>
<td>136</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \Pi_{\text{ScMax}2} ) [6]</td>
<td>80</td>
<td>112</td>
<td>136</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \Pi_{\text{ICMax}} ) [5]</td>
<td>20</td>
<td>28</td>
<td>34</td>
<td>40</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Our ( \Pi_{\text{Max}} )</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>Run Time (s)</td>
<td>( \Pi_{\text{ScMax}1} ) [12]</td>
<td>6.02</td>
<td>9.06</td>
<td>11.68</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \Pi_{\text{ScMax}2} ) [6]</td>
<td>6.44</td>
<td>9.20</td>
<td>11.85</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \Pi_{\text{ICMax}} ) [5]</td>
<td>1.66</td>
<td>2.35</td>
<td>3.34</td>
<td>9.80</td>
<td>17.35</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Our ( \Pi_{\text{Max}} )</td>
<td>2.60</td>
<td>2.83</td>
<td>3.60</td>
<td>8.45</td>
<td>14.43</td>
</tr>
<tr>
<td></td>
<td>Commu. Volume/server (MB)</td>
<td>( \Pi_{\text{ScMax}1} ) [12]</td>
<td>0.06</td>
<td>0.29</td>
<td>2.96</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \Pi_{\text{ScMax}2} ) [6]</td>
<td>0.062</td>
<td>0.61</td>
<td>6.18</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \Pi_{\text{ICMax}} ) [5]</td>
<td>0.011</td>
<td>0.114</td>
<td>1.144</td>
<td>11.444</td>
<td>22.88</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Our ( \Pi_{\text{Max}} )</td>
<td>0.005</td>
<td>0.038</td>
<td>0.370</td>
<td>3.696</td>
<td>7.392</td>
</tr>
</tbody>
</table>

...
element converter Convert\(_2\) realized with AES expansion, the overall expansion comparison within the corresponding evaluation algorithm in IC\(_{\text{Max}}\) and IC\(_{\text{Max}}\) are 8\(n\lambda + 2n(\ell + 2)\), 6\(n\lambda + 3n(\ell + 2)\). Thus, when comparing the cost of IC\(_{\text{Max}}\) to IC\(_{\text{Max}}\), it requires \(n(2\lambda - \ell - 2)\) bits more expansion and \(T(n)\) less cost on other computational overhead.

Assuming that the dominant factor determining the online computation-time arises from the arithmetical operation overhead part \((T(n))\) when \(m\) is small, and also assuming that the dominant factor determining the online run-time when \(m\) keeps on increasing arises from the AES expansion part, then this clearly explains the numbers shown in Table 6. When \(m \in \{10^5, 10^4, 10^3\}\), we see from Tab. 6 that IC\(_{\text{Max}}\) performs better online efficiency than IC\(_{\text{Max}}\), because either less communication rounds required, less computation overhead required, or both. Conversely, when \(m\) keeps on increasing, we see a clear online run-time efficiency advantage of IC\(_{\text{Max}}\) over IC\(_{\text{Max}}\), which cannot only be explained by less communication rounds used, e.g., when \(m = 5 \times 10^6\), IC\(_{\text{Max}}\) requires almost seven seconds less than that in IC\(_{\text{Max}}\) in terms of online run-time, while meanwhile the communication rounds difference between them two contributes to only 14 \(\times 80\) ms = 1.12s saving in theory.

### 6.2. Evaluation for \(\mathcal{F}_{\text{KRE}}\)

**SOTA implementations for \(\mathcal{F}_{\text{KRE}}\).** Given the absence of efficient solutions for \(\mathcal{F}_{\text{KRE}}\), we constructed \(\Pi_{\text{NaiveKre}}\), a straightforward comparison-based solution for computing the \(k\)-th ranked element from a secret shared set \(X\). It takes the arithmetical secret sharing \([k]^{\lambda}\) of an index \(k < m\) and the arithmetical secret sharing of set \(X\) as inputs, and outputs the arithmetical secret sharing of the \(k\)-th ranked element of \(X\).

\(\Pi_{\text{NaiveKre}}\) begins by performing a full sorting over \([X]^{\lambda}\) to obtain a vector \([V]^{\lambda}\) in \(O(m \log m)\) iterations. Subsequently, it identifies the \(k\)-th ranked element by computing the inner product between \([V]^{\lambda}\) and another vector \([I]^{\lambda}\). Here, \([I]^{\lambda}\) is of size \(m\), where the \(i\)-th element is the zero-check result over \(i \in [n]\). Notably, the input index \([k]^{\lambda}\) is given as a secret sharing, preventing the servers from utilizing binary search and necessitating a full sort to extract the \(k\)-th ranked element of \(X\). In our implementation of \(\Pi_{\text{NaiveKre}}\), we use the integer interval containment function secret sharing as the underlying secure comparison implementation, as the same as in IC\(_{\text{Max}}\).

In Table 7, we evaluate the performance of our protocols \(\Pi_{\text{BitKre}}\) and \(\Pi_{\text{BatchKre}}\) against \(\Pi_{\text{NaiveKre}}\). Note that we set the batch size \(\omega = 3\) in the benchmark of \(\Pi_{\text{BatchKre}}\). Thus, we use input domain size \(n = 30\) throughout all protocol benchmark in Table 7. Due to the significant communication rounds and resulting high runtime required for \(\Pi_{\text{NaiveKre}}\), we could only evaluate it up to \(m = 50\).

**Offline/Online phase comparison.** Apart from slightly higher communication volume when \(m = 10\) in our protocols, \(\Pi_{\text{BitKre}}\) and \(\Pi_{\text{BatchKre}}\) outperform \(\Pi_{\text{NaiveKre}}\) notably in terms of both offline and online efficiency in every measurement. This advantage stems from the underlying communication/computation complexity advantage of our protocols over \(\Pi_{\text{NaiveKre}}\). Furthermore, this performance gap becomes increasingly substantial as \(m\) increases.

### 7. Conclusion

In conclusion, comparing our proposed protocols with existing general solutions, in terms of online evaluation efficiency, all our proposed protocols exhibit better online communication volume efficiency than existing works, and all of them scale better than existing works. This is particularly true for our protocols that compute \(\mathcal{F}_{\text{Kre}}\), where it significantly excels the naive solution that utilize full sorting.

### Acknowledgements

This work was partially supported by the ArmaSuisse project with number CYD-C-2020011 titled “Private, Robust, and Efficient Computation of Aggregate Statistics”.

### References


A. Security Analysis

We consider a static corruption model where a Honest-but-Curious adversary $A$ chooses one of the two computing parties $S_0, S_1$ before the execution of the computations.

In the subsequent discussion, we focus exclusively on a formal security proof of our protocol $\Pi_{\text{Max}}$. The security analysis for other protocols $\Pi_{\text{rel}}, \Pi_{\text{Kre2}}$ follow a similar structure, and we omit it here.

A.1. Security of $\Pi_{\text{Max}}$

In the following we define an ideal functionality $F_{\text{Max}}$ that interacts with $S_0, S_1$, and the adversary $A$ is parameterized by a public known function $f_{\text{Max}}(X)$.

- **Input:** $F_{\text{Max}}$ inputs a Boolean secret sharing $[X]_b$ of a multi-set $X$.
- **Computation:** $F_{\text{Max}}$ reconstructs $X$ from $[X]_b$, computes $y = f_{\text{Max}}(X)$.
- **Output:** $F_{\text{Max}}$ sends $[y]_b$ to $S_b$ for $b \in \{0, 1\}$.

Also, we define another ideal functionality $F_{\text{BitMax}}$ works almost the same as $F_{\text{Max}}$ differently it inputs an additional index $i \in \{0, 1, \ldots, n\}$ and outputs $[y]_b$ instead of $[y]_b$ to $S_b$ for $b \in \{0, 1\}$.

**Theorem 1.** There is a PPT algorithm simulator $\text{Sim}_0$ that realizes ideal functionality $F_{\text{Setup}}$, which inputs only $(X, \pi, \text{Sim}_0, \Pi_{\text{Max}})$ and outputs $(K^*_e, Q^*_b)$, such that the output is computationally indistinguishable from the real offline execution in section 3.3.

**Proof.** From the security analysis of the I-DPF construction in [3] we know that there is a PPT algorithm simulator that outputs string $K^*_e$ that are computationally indistinguishable from the real world output $K_e$. Also, $Q_b$ is perfectly indistinguishable with $Q^*_b$ when the simulator $\text{Sim}_b$ chooses $Q^*_b \leftarrow \text{Sim}_b$.

**Theorem 2.** In the $(F_{\text{Setup}}, F_{\text{Max}})$-hybrid model, there is a PPT simulator $\text{Sim}_b$ that inputs $(X, i, \delta_i, \text{Sim}_0)$ and outputs $[y]_b$ to $S_b$ for $b \in \{0, 1\}$.

**Proof.** We construct a simulator $\text{Sim}_b$ which inputs $y = f_{\text{BitMax}}(X, i)$ and outputs the view for the server $S_b$ where $b \in \{0, 1\}$:

$-$ $S_b$ uniformly generates $\delta^*_i \leftarrow \{0, 1\}$, which is perfectly indistinguishable with $\delta_i$'s view $\delta_i$ in line 13 of Fig. 4.
As for all \( j \in [m], i \in [n] \), that \( S_b \) uniformly select \( \alpha_j \leftarrow \{0,1\}^n \), \( q_i \leftarrow \{0,1\} \), it holds that they are identical to that in the real execution of protocol \( \Pi_{\text{Max}} \).

For all other secret share within the real protocol execution, \( S_b \) generates random group elements which are perfectly indistinguishable to what in the real protocol execution.

\[ \square \]

**Definition 1.** (Security) There is a PPT simulator \( \text{Sim}_b \) such that \( \forall X \in Z_{2^n}^m \) and function \( f_{\text{Max}}(X) : \{0,1\}^{n \times m} \rightarrow \{0,1\}^{n} \), \( S_b \) realizes the ideal functionality \( F_{\text{Max}} \), such that its behavior is computationally indistinguishable from a real world execution of protocol \( \Pi_{\text{Max}} \) in the presence of a semi-honest adversary \( A \).

**Theorem 3.** In the \( (F_{\text{Setup}},F_{\text{BitMax}}) \)-hybrid model, protocol \( \Pi_{\text{Max}} \) securely realize the functionality \( F_{\text{Max}} \).

**Proof.** We construct a simulator \( \text{Sim}_b \) that accepts \( (K_b^i,Q_b^i,\sigma_i^* \}_{i \in [n]} \) as input, where \( (K_b^i,Q_b^i) \) is derived from the simulation output of \( F_{\text{Setup}} \) and \( \{\sigma_i^* \}_{i \in [n]} \) is from the simulation output of \( F_{\text{BitMax}} \). To accurately simulate the view transcript of the server \( \text{Server}_b \) controlled by the adversary \( A \), we focus on the following two key points that are sufficient for simulating the real-world execution of \( \Pi_{\text{Max}} \):

- \( \text{Sim}_b \) randomly select \( t_i \leftarrow \{0,1\}^{n} \) simulating \( S_b \)'s view in line 6 of Fig. 4, for all \( j \in [m] \). As for all \( j \in [m] \) that \( \alpha_j \) is uniformly randomly selected in \( \Pi_{\text{Max}} \), it holds that the view of \( A \) in our simulation is identical to that in the real execution of protocol \( \Pi_{\text{Max}} \).

- \( \text{Sim}_b \) passes \( \sigma_i^* \) to \( A \) simulating \( S_b \)'s view in line 13 of Fig. 4.

\[ \square \]

By the composability of secure protocols, it is sufficient to show that our protocol \( \Pi_{\text{Max}} \) is secure against a Honest-but-Curious adversary \( A \).

**B. Full construction of \( \Pi_{\text{BatchKre}} \)**

In Protocol 3, we present the full construction of \( \Pi_{\text{BatchKre}} \) that securely computes the \( i \)th batch of the KRE of \( X \). Here, the servers start by computing the ordered vector \( [v_i]^A \) using the conversion matrix \( [M]^A \) and the initial vector \( [\bar{v}]^A \). Subsequently, \( \forall t \in [\tau] \), the secret sharing \( [v_i^t]^A \) is computed as \( \sum_{t=1}^i [v_i^t]^A \), a \( F_{\text{LessEqualThan}} \) comparison is executed on each \( [v_i^t]^A \) producing a comparison result \( [c_i^t]^A \). Denote \( v^\omega \) as \( \{v_1^\omega, \ldots, v_{\tau}^\omega\} \) and \( \bar{c} \) as \( \{c_1, \ldots, c_\tau\} \). Since the elements in \( v^\omega \) are monotonically increasing, the vector \( \bar{c} \) initially contains zeros. Starting from a specific index in \( \bar{c} \), all subsequent elements will have a value of one. Let \( \xi \) be the index of the first non-zero element in \( \bar{c} \). This marks target batch containing the corresponding prefix of the \( k^m \) ranked element of set \( X \). To identify \( \xi \) the following conditional equation is utilized for each \( t \) in \( [\tau] \):

- \( [b_i^t]^A \leftarrow [c_i^t]^A \) if \( t = 1 \),
- \( [b_i^t]^A \leftarrow [c_i^t]^A \cdot (1 - [c_i-t]^A) \) if \( t > 1 \).

**FUNCTIONALITY \( F_{\text{ConvMatrix}}(q) \):**

**Input:** A binary string \( q \) of length \( \omega \).

**Output:** A \( \tau \times \tau \) matrix \( M \).

1. Set \( M \) as an \( \tau \times \tau \) matrix initiated as all zeros.
2. \( \xi \leftarrow \{q_1q_2 \cdots q_\omega\} \) or \( \{1-q_1\cdots q_\omega\} \)
3. for \( i = 1 \) to \( \tau \) do
4. \( \eta_i \leftarrow F_{\text{BDC}}(\tau - i, \omega) \)
5. for \( j = 1 \) to \( \tau \) do
6. \( \xi \leftarrow F_{\text{BDC}}(\tau - j, \omega) \)
7. \( s_j \leftarrow \{0\}^\omega \)
8. for \( k = 1 \) to \( \omega \) do
9. if \( \eta_k = 1 \) then
10. \( s_k = \xi \)
11. else
12. \( s_k = \neg \xi \)
13. end if
14. end for
15. \( M[i][j] \leftarrow Q_{\sigma_{\tau - F_{\text{BDC}}(s)}} \)
16. end for
17. end for
18. Outputs \( M \).

![Figure 6: Conversion matrix generation.](image)

Notice that only \( b_\xi \) will be one and all other \( b_t \) values for \( t \neq \xi \) will be zero. This enables us to compute the masked bits of the target batch \( \delta \) accordingly with

\[
\delta_i^B \leftarrow \delta_i^B \oplus (s_i \wedge F_{\text{A2B}}([h_i]^A)), \quad \text{for } i \in [\omega].
\]

And this also enables us to update \( k \) with

\[
[k_i]^A \leftarrow [b_1]^A \cdot [k]^A + \sum_{i=2}^{\tau} [b_i]^A \cdot ([k_i]^A - [v_i-1]^A).
\]

**C. Further supported functionality**

Here we show that our proposed secure computation framework 4 and its variant 5 are not limited to perform only maximum or k-th ranked element over a set \( X \). They also support other functionalities, including computing the minimum of \( X \), verifying whether a given Boolean secret-shared value \( [q]^B \) is equal to the maximum of a set \( [X]^B \), or computing one common number of \( X \).

**C.1. Computing the Minimum**

With a few modifications on protocol 1 we can also compute the minimum of \( X \). For this to work, in protocol 1, we perform a zero check instead of a non-zero check and adjust the computation of \( [m]^A \). Concerning the latter, there are two instances where we prepare \( [w]^A \).

First, during initialization when \( i = 1 \), we compute \( [w]^A = m \cdot [b_0]^A \). Second, for \( i \in (1, n) \), we use following computations instead:

\[
[w_1]^A \leftarrow [v_1]^A \cdot [b_1]^A,
[w_2]^A \leftarrow [v_2]^A \cdot [b_1]^A.
\]

These adjustments ensure the extraction of the \( i \)th bit of the minimum value of \( X \), as we outline below.
Protocol 3 The batch-wise k-th ranked element protocol

Functionality: $(\delta, \ast) \leftarrow \Pi_{\text{MaxAdd}}(i, [\sigma]^A, [M]^A, [g]^B, [k]^A, \ast)$

Input: A vector $\vec{v}$ of length $\tau$, a binary string $q$ of length $\omega$ and $[k]^A$.

Output: A binary string $\sigma$ of length $\omega$, and an updated $[k]^A$ if $i < d$.

1: $[\vec{v}]^A \leftarrow [\pi]^A \times [M]^A$ \> Compute ordered prefix query results.
2: for $t = 1$ to $\tau$ do
3: $[\pi]^A \leftarrow \sum_{t=1}^{\tau} [v]^A$
4: $[\pi]^A \leftarrow F_{\text{EqualThan}}([\vec{v}]^A, [v]^A)$ \> Compute ordered comparison bits.
5: end for
6: $[\sigma]^B \leftarrow [\pi]^B$
7: for $t = 1$ to $\tau$ do
8: $s \leftarrow F_{\text{RDC}}(t - t, \omega)$
9: if $t = 1$ then
10: $[b_1]^A \leftarrow [\sigma]^A$
11: else
12: $[b_1]^A \leftarrow [c_i]^A \cdot (1 - [c_i-1]^A)$
13: end if
14: for $i = 1$ to $\omega$ do
15: $[\sigma|^B \leftarrow [\sigma]^B \oplus (s_i \land F_{\text{AZM}}([b_i]^A))$
16: end for
17: end for
18: $s \leftarrow F_{\text{RDC}}([\sigma]^B)$
19: if $i < d$ then
20: $[k]^A \leftarrow [b_1]^A \cdot [k]^A + \sum_{t=1}^\tau [b_1]^A \cdot ([k]^A - [v]^A)$ \> Update $[k]^A$ for use in the next iteration.
21: end if
22: Outputs $(\sigma, [k]^A)$.
23: else
24: Outputs $\sigma$.
25: end if

- Otherwise, We have $[w]^A = [\pi]^A$, $[q_i]^A = [0]^A$. When invoking $F_{\text{ZeroCheck}}([\pi]^A - [\pi]^A)$, the zero check is performed on $[\pi]^A$. If this check passes, it means that all remaining candidates for the minimum start with the prefix $[\pi]^A$.

In both cases, if the corresponding zero check passes, we can be certain that the $i^{th}$ bit of the minimum is 1; otherwise, it is 0.

C.2. Detailed construction of $\Pi_{\text{MaxVry}}$

To verify whether a Boolean secret-shared value $[a]^B$ is equal to the maximum of a set $[X]^B$, a straightforward method might involve using protocol 1 to first calculate the maximum $[c]^B$ and then check for equality between $[c]^B$ and $[a]^B$. Yet, this direct approach leads to communication costs proportional to the input size $n$. Is it possible to check for the maximum value with fewer communication rounds?

Interestingly using I-DPF and the primitive CondEval, we constructed a protocol $\Pi_{\text{MaxVry}}$, which verifies the maximum value within a mere three rounds of communication. Protocol $\Pi_{\text{MaxVry}}$ inputs $[X]^B$ and $[a]^B$, and outputs $[c]^B$, which indicates if $a$ is the maximal value in $X$. The protocol achieves this by computing $[c]^B = [c_0]^B \land [c_1]^B$, where $c_0$ checks the existence of $a$ in $X$, and $c_1$ determines whether any element greater than $a$ is present in $X$.

In the following we present protocol $\Pi_{\text{MaxVry}}$ that performs this maximum verification in just three communication rounds. The protocol inputs $[X]^B$ and $[a]^B$, and outputs $[c]^B$, where $c$ indicates if $a$ is the maximum element in $X$. In essence, we compute $[c]^B = [c_0]^B \land [c_1]^B$ where $c_0$ confirms the existence of the value $a$ in $X$, and $c_1$ indicates if there is greater value than $a$ exists in $X$.

Protocol 4 The Maximum verification protocol

Functionality: $[y]^B \leftarrow \Pi_{\text{MaxVry}}([X]^B, [K], [a]^B)$

Input: For each $b \in \{0, 1\}$, $\sigma_b$ inputs $K_b$ in the offline phase; $[a]^B$ where $a \in \mathbb{Z}_2^n$ and $[x]^B(j \in [m])$ in the online phase.

Output: A Boolean Secret Sharing $[c]^B$ where

1: for $j = 1$ to $m$ do
2: for $b = 0$ to $1$ do
3: $s_{j,b}^0 \leftarrow k_{j,b}$
4: end for
5: $t_1^j \leftarrow F_{\text{reveal}}([a \oplus x_j \land \alpha_j]^B)$
6: end for
7: $w^B \leftarrow F_{\text{REVEAL}}([0])$
8: $B \leftarrow F_{\text{REVEAL}}([0])$
9: for $i = 1$ to $n$ do
10: $(a_i^B) \leftarrow F_{\text{RB2A}}([a_i^B])$
11: for $j = 1$ to $m$ do
12: for $b = 0$ to $1$ do
13: $(s_{j,b}^0, t_1^j, b_{j,b}) \leftarrow \text{IDPF.EvalNext}(b, s_{j,b}^1, 0, pp, t_1^j)$
14: $v_i^j \leftarrow 1 \oplus t_i^j$
15: $(s_{j,b}^1, t_1^j, b_{j,b}) \leftarrow \text{IDPF.EvalNext}(b, s_{j,b}^1, 0, pp, v_i^j)$
16: end for
17: end for
18: $[\mu]^B \leftarrow \sum_{j=1}^m [\beta]^B$
19: end for
20: $[\beta]^B \leftarrow \sum_{j=1}^m [\beta]^B$
21: $[c_0]^B \leftarrow F_{\text{AZM}}([\beta]^B)$
22: $w^B \leftarrow \sum_{j=1}^m ([1 - [\alpha_j]^B]) \cdot [\mu]^B)$
23: $[c_1]^B \leftarrow \text{CondEval}([c_0]^B \land F_{\text{AZM}}([w]^B))$

In our approach, most steps can be executed locally, except for the following operations in protocol 4:

i. In line 10, servers run $F_{\text{RB2A}}$ on each bit of $[a]^B$ and obtain $[a_i]^B$, which are used later for calculating $[w]^B$.
ii. In lines 21 and 22, where servers compute $[c_0]^B$ and $[w]^B$.
iii. In line 23, where servers perform CondEval.

While this protocol incurs a computational overhead of $2nm$, which is 0.5nm more compared to what in protocol 1, it compensates with the advantage of constant communication rounds. This makes it a suitable trade-off for applications with small to medium-sized client inputs. For larger datasets, a careful evaluation of the trade-offs is necessary to choose the most efficient protocol for the task at hand.

To extend this protocol for verifying a minimum value, a single modification is needed: in line 17 of protocol 4, the calculation for $[w]^B$ should be adjusted to:

$[w]^A \leftarrow [w]^A + [\alpha_i]^A \cdot [\mu_i]^A$
This change ensures that \( c_1 \) will indicate no existing integer in \( X \) that is smaller than \( a \), effectively accomplishing the task of minimum verification.

**C.3. Computing the common number**

As for the protocol design of the computation of the common number of \( X \), where target string is computed bit-wisely (e.g., from MSB to LSB), specifically, the \( i^{th} \) target bit is identified by comparing \( \text{PrefixCount}(p||1) \) and \( \text{PrefixCount}(p||0) \), where \( p \) denotes the determined target prefix.