# 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 $\Pi_{\mathsf{Max}}$ 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 $\mathcal{O}(m)$ 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)* $\mathcal{F}_{\mathsf{MAX}}$: Computing the maximum of the clients' secret inputs and outputs the SS of the targeted value; *(ii)* Function hiding $\mathcal{F}_{\mathsf{KRE}}$: 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 computation 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 output bit in merely three rounds. Concretely, our contributions can be summarised as follows:

1) We presented a new secure computation framework in Fig. 4 that efficiently computes some secure aggregation tasks bit-wisely, we also presented its variant in Fig. 5 that further reduces communication complexity by computing target functionality batch-wisely.

2) We present two efficient protocols $\Pi_{\mathsf{Max}}$ and $\Pi_{\mathsf{BitKre}}$ that compute the maximum respectively the $k$-th ranked element of a set $X$ bit-wisely.

3) We extend $\Pi_{\mathsf{BitKre}}$ to support batch-wise computation and thus obtain $\Pi_{\mathsf{BatchKre}}$, compare it $\Pi_{\mathsf{BitKre}}$, $\Pi_{\mathsf{BatchKre}}$ gives us reduced communication complexity.

4) We have implemented and outsourced our protocols $\Pi_{\mathsf{Max}}$, $\Pi_{\mathsf{BitKre}}$ and $\Pi_{\mathsf{BatchKre}}$. In addition, we demonstrated their efficiency advantage over state of the art solutions by conducting a practical performance comparison with established state of the art solutions ($\Pi_{\mathsf{ScMax1}}, \Pi_{\mathsf{ScMax2}}$) from MP-SPDZ [17], and two basic comparison-based solutions $\Pi_{\mathsf{ICMax}}, \Pi_{\mathsf{NaiveKre}}$ which we built up from scratch that utilize 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.

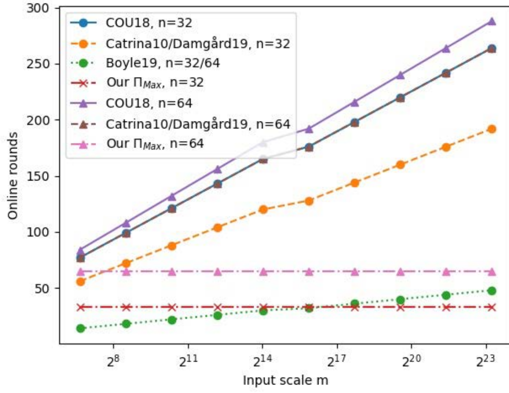
## 1.1. Related work

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.

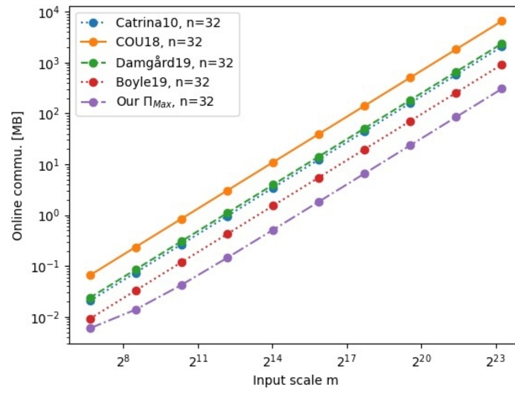
*Comparison-based approach.* This approach necessitates $m-1$ secure comparisons in $\log m$ rounds to determine the maximum of $X$. Within this comparison-based approach, the overall efficiency for computing the maximum is determined by the efficiency of the secure comparison protocol employed. There 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 $\mathcal{O}(mn(n+\kappa))$ across $\mathcal{O}(\log m)$ rounds. The approach employed by Garay *et al.* [14] necessitates $\mathcal{O}(mn)$ in $\mathcal{O}(\log m \log n)$ rounds and the technique introduced by Couteau [10] requires $\mathcal{O}(mn)$ in $\mathcal{O}(\log m \log\log n)$ 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




(a) Online commu. rounds comparison




(b) Online commu. volume comparison

Figure 1: Concrete online evaluation cost comparison

TABLE 1: Comparative overview of protocols for securely computing the maximum. This table sets out the communication complexity comparison among our protocols $\Pi_{\mathsf{Max}}$, 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.

| Ref. | Primitive | Comm.[bits] Offline | Comm.[bits] Online | Online rounds |
|---|---|---|---|---|
| DGK07 [13] | DGK | - | $\mathcal{O}(mn(n+\kappa))$ | $\mathcal{O}(\log m)$ |
| GSV07 [14] | OT | $\mathcal{O}(\kappa mn/\log\kappa)$ | $\mathcal{O}(mn)$ | $\mathcal{O}(\log m \log n)$ |
| Cou18 [10] | OT | $\mathcal{O}(\kappa mn/\log\kappa)$ | $\mathcal{O}(mn)$ | $\mathcal{O}(\log m \log\log n)$ |
| Catrina10 [6] | Truncation | - | $\mathcal{O}(m(n+\kappa))$ | $\mathcal{O}(\log m \log n)$ |
| Damgård19 [12] | Bit triple | $\mathcal{O}(mn)$ | $\mathcal{O}(mn)$ | $\mathcal{O}(\log m \log n)$ |
| Boyle19 [5] | DCF | $\mathcal{O}(mn)$ | $\mathcal{O}(mn)$ | $\mathcal{O}(\log m)$ |
| ZCZ15 [21] | - | - | $\mathcal{O}(\kappa nm)$ | $\mathcal{O}(n)$ |
| Prio [9] | - | 0 | $\mathcal{O}(2^n\kappa)$ | 0 |
| Prio+ [1] | - | 0 | $\mathcal{O}(2^n\kappa)$ | 0 |
| Our $\Pi_{\mathsf{Max}}$ | I-DPF | $\mathcal{O}(\kappa mn)$ | $\mathcal{O}(mn)$ | $n+1$ |

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 $\Pi_{\mathsf{BatchKre}}$, $\omega$ 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.

| Ref. | Primitive | Leakage | Total Comm. | Rounds |
|---|---|---|---|---|
| AMP10 [2] | - | $\bullet$ | $\mathcal{O}(mn^2)$ | $\mathcal{O}(n)$ |
| ZCZ15 [21] | - | $\bullet$ | $\mathcal{O}(\kappa nm)$ | $pn$ |
| TKK$^+$20 [19] | YGC | $\circ$ | $\mathcal{O}(\kappa m^2)$ | 4 |
| | AHE1 | $\circ$ | $\mathcal{O}(\kappa m^2 nt)$ | 4 |
| | AHE2 | $\circ$ | $\mathcal{O}(\kappa m^2 nt)$ | 4 |
| CHH$^+$22 [7] | OT | $\bullet$ | $\mathcal{O}(\kappa mn)$ | $\mathcal{O}(\log S)$ |
| Our $\Pi_{\mathsf{BitKre}}$ | I-DPF | $\circ$ | $\sim mn$ | $1+2n$ |
| Our $\Pi_{\mathsf{BatchKre}}$ | I-DPF | $\circ$ | $\sim mn$ | $1+4n/\omega$ |

$1-n/2^\kappa$, 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 subprotocol $M$ 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 $\{0,\cdots,B-1\}$ into $b = \log_c B$ 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 $\Pi_{\mathsf{BitKre}}$, $\Pi_{\mathsf{BatchKre}}$, 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 $\mathcal{F}_{\mathsf{MAX}}$, in Table 1 we compare the communication complexities between our protocol $\Pi_{\mathsf{Max}}$ 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-DPFs (which is one of the latest extension of DPF) are executed in $\mathcal{O}(n)$ 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 $\mathcal{O}(2^n)$ 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] and [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 $\Pi_{\mathsf{Max}}$ 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 $\mathsf{SC}_3$ 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 $21.5nm$ 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 $\Pi_{\mathsf{Max}}$, it costs approximately $nm$ bits in $n + 1$ rounds. When the input set size $m$ is greater than $2^{16.5} (0.92 \times 10^5)$, as shown in Fig. 1 that our protocol $\Pi_{\mathsf{Max}}$ requires the least online rounds. Furthermore, independently of the value of $m$, our protocol $\Pi_{\mathsf{Max}}$ 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 $k^{\text{th}}$ 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 $\mathcal{O}(\log N)$ 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 $\mathsf{S}_0, \mathsf{S}_1$ as the two computing servers in our protocols. We use following notations throughout this paper:

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

- $\mathcal{F}_{\mathsf{Reveal}}$ - *secret sharing revelation*: Upon receiving either a Boolean secret sharing $[b]^{\mathsf{B}}$ or an arithmetic

secret sharing $[x]^{\mathsf{A}}$, it reveals the corresponding bit $b$ or value $x$ to the two servers.

- $\mathcal{F}_{\mathsf{A2B}}$ - *arithmetic to Boolean secret sharing conversion*: Upon receiving an arithmetical secret sharing $[x]^{\mathsf{A}}$ where $x \in \{0,1\}$, it outputs the corresponding Boolean secret sharing $[x]^{\mathsf{B}}$.
- $\mathcal{F}_{\mathsf{B2A}}$ - *Boolean to arithmetic secret sharing conversion*: Upon receiving a Boolean secret sharing $[b]^{\mathsf{B}}$, it outputs $[b]^{\mathsf{A}}$.
- $\mathcal{F}_{\mathsf{NZ}}$ - *Nonzero check over a secret sharing*: Upon receiving an arithmetical secret sharing $[x]^{\mathsf{A}}$, it outputs $[1]^{\mathsf{A}}$ if $x$ is not zero, otherwise it outputs $[0]^{\mathsf{A}}$.
- $\mathcal{F}_{\mathsf{LessEqualThan}}$ - *LessEqual check between two values*: Upon receiving two arithmetical secret sharing $[x]^{\mathsf{A}}$ and $[y]^{\mathsf{A}}$, it outputs $[1]^{\mathsf{A}}$ if $x \leq y$, otherwise $[0]^{\mathsf{A}}$.

## 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 $\mathsf{Gen}(\alpha, \beta) \rightarrow k_0, k_1$ routine, two keys are produced which represent the secret shares. The individual keys can be evaluated as $\mathsf{Eval}(k, x) \rightarrow \mathbb{G}$ 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 $\mathcal{O}(n)$, 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 $x \in \{0,1\}^*$ 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 $(\alpha, (\mathbb{G}_1, \beta_1), \cdots, (\mathbb{G}_n, \beta_n))$ (shorthand $(\alpha, \bar{\beta})$), where $\alpha \in \{0,1\}^n$ and for every $\ell \in [n]$, $\mathbb{G}_\ell$ is the description of an Abelian group and $\beta_\ell \in \mathbb{G}_\ell$. The function is then defined as follows:

$$f_{\alpha,\bar{\beta}} : \cup_{\ell \in [n]} \{0,1\}^\ell \rightarrow \cup_{\ell \in [n]} \mathbb{G}_\ell, \text{ given by}$$

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

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 $t$-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.

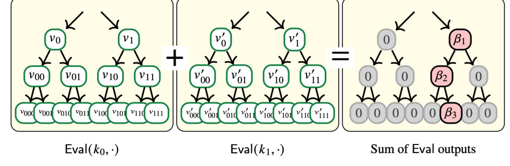



Figure 2: Visualization of the two binary trees for an I-DPF and the sum of the evaluations. Picture taken from [3].

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

- $\mathsf{IDPF.Gen}(1^\kappa, (\alpha, (\mathbb{G}_1, \beta_1), \cdots, (\mathbb{G}_n, \beta_n)))$ is a PPT key generation algorithm that given $1^\kappa$ (security parameter) and a description $(\alpha, (\mathbb{G}_1, \beta_1), \cdots, (\mathbb{G}_n, \beta_n))$ of an all-prefix function, it outputs a pair of keys and public parameters $(k_0, k_1, \mathsf{pp} = (\mathsf{pp}_1, \cdots, \mathsf{pp}_n))$, where $\mathsf{pp}$ includes the public values $\kappa, n, \mathbb{G}_1, \cdots, \mathbb{G}_n$.
- $\mathsf{IDPF.EvalNext}(b, \mathsf{st}_b^{i-1}, \mathsf{pp}_i, x_i)$ is a polynomial-time incremental evaluation algorithm that given a server index $b \in \{0,1\}$, secret state $\mathsf{st}_b^{i-1}$, public parameter $\mathsf{pp}_i$, and input evaluation bit $x_i \in \{0,1\}$, it outputs an updated state and output share value: $(\mathsf{st}_b^i, y_b^i)$.
- $\mathsf{IDPF.EvalPrefix}(b, k_b, \mathsf{pp}, x_1 \| \cdots \| x_\ell)$ is a polynomial-time prefix evaluation algorithm that given a server index $b \in \{0,1\}$, secret state $\mathsf{st}_b^{i-1}$, public parameter $\mathsf{pp}$, and input evaluation prefix $x_1 \| \cdots \| x_\ell \in \{0,1\}^\ell$, it outputs a corresponding share value $y_b^\ell$.

Assuming that the keys are generated according to $(k_0, k_1, \mathsf{pp}) \leftarrow \mathsf{IDPF.Gen}(1^\kappa, (\alpha, (\mathbb{G}_1, \beta_1), \cdots, (\mathbb{G}_n, \beta_n)))$ each party $b \in \{0,1\}$ can calculate its output share $y_b^\ell$ for a prefix $x_1 \| \cdots \| x_\ell$ as:

1: $\mathsf{st}_b^0 \leftarrow k_b$
2: **for** $j = 1$ to $\ell$ **do**
3: $\quad (\mathsf{st}_b^j, y_b^j) \leftarrow \mathsf{IDPF.EvalNext}(b, \mathsf{st}_b^{j-1}, \mathsf{pp}_j, x_j)$
4: **end for**
5: **return** $y_b^\ell$.

For these output shares derived using $\mathsf{IDPF.EvalNext}$, we require that: $y_0^\ell + y_1^\ell = f_{\alpha,\bar{\beta}}(x_1 \| \cdots \| x_\ell)$ 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 $y_b^\ell$ of party $b \in \{0,1\}$ output by $\mathsf{IDPF.EvalPrefix}(b, k_b, x_1 \| \cdots \| x_\ell)$, we require that $y_0^\ell + y_1^\ell = f_{\alpha,\bar{\beta}}(x_1 \| \cdots \| x_\ell)$ holds at all times. It is assumed that the keys are generated according to $(k_0, k_1, \mathsf{pp}) \leftarrow \mathsf{IDPF.Gen}(1^\kappa, (\alpha, (\mathbb{G}_1, \beta_1), \cdots, (\mathbb{G}_n, \beta_n)))$.

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

$\mathsf{CondEval}$ is a cryptographic primitive that was initially introduced in [8], it inputs a Boolean secret sharing

$[s]^{\mathsf{B}}$, an arithmetic secret share $[x]^{\mathsf{A}}$, a function secret sharing of a function $f$ and a binary operator $\circ$, outputs the arithmetical secret sharing $[s \circ f(x)]^{\mathsf{A}}$, *i.e.,*

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

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 $[s]^{\mathsf{B}}$, an arithmetic secret share $[x]^{\mathsf{A}}$, a function $f$, and a binary operator $\wedge$, CondEval is realized by

$$\mathcal{K}_{\wedge} \leftarrow \mathsf{CondEval.Setup}(\wedge, 1^{\lambda}, \mathsf{pp}, f) \text{ and}$$
$$[s \wedge f(x)]^{\mathsf{B}} \leftarrow \mathsf{CondEval.Eval}(\wedge, \mathcal{K}_{\wedge}, [s]^{\mathsf{B}}, [x]^{\mathsf{A}})$$

respectively in the offline and online phase.

In the offline phase, a trusted third party takes the operation $\wedge$, a specific function $f$ can be secret shared, and public parameters pp to generate and distribute conditional evaluation key shares $\mathcal{K}_{\wedge}$ to two servers. These keys are used for secure computations in the online phase. In the online phase, upon the secret share $[s]^{\mathsf{B}}$ and the arithmetic share $[x]^{\mathsf{A}}$ are ready, two servers collaboratively perform CondEval.Eval in a single round and obtain $[s \wedge f(x)]^{\mathsf{B}}$. 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 initially 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: $X = \{11, 10, 01, 00, 10, 00, 10, 11, 10, 10\}$. 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

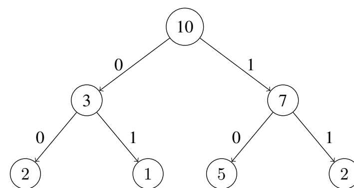



Figure 3: The prefix query tree of $X$.

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.,* , refering 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

$$((k_0, k_1), \mathsf{pp}) \leftarrow \mathsf{IDPF.Gen}(1^{\kappa}, \alpha, (\mathbb{G}_1, \mathbf{1}), \cdots, (\mathbb{G}_n, \mathbf{1}))$$

and we set the initial state to $\mathsf{st}_b^0 \leftarrow k_b$. Now when we evaluate the I-DPF keys for the first bit we have two cases:

- For $\alpha^1 \oplus 0$ we have:
  $(\mathsf{st}_b^1, [1]_b^{\mathsf{A}}) \leftarrow \mathsf{IDPF.EvalNext}(b, \mathsf{st}_b^0, \mathsf{pp}, \alpha^1 \oplus 0)$.
- For $\alpha^1 \oplus 1$ we have:
  $(\mathsf{st}_b^1, [0]_b^{\mathsf{A}}) \leftarrow \mathsf{IDPF.EvalNext}(b, \mathsf{st}_b^0, \mathsf{pp}, \alpha^1 \oplus 1)$.

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 $x^1$ 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 $[x_j]^{\mathsf{B}}, [\alpha_j]^{\mathsf{B}}$ and $[q]^{\mathsf{B}}$ and reveal the string $t^j = q \oplus x_j \oplus \alpha_j$ for each input $x_j$. These strings do not reveal anything about the inputs $x_j$ because of the way $\alpha_j$ 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 $q^i \equiv x_j^i$. 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 target 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 element, 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 $\mathcal{F}_{\mathsf{MAX}}$ or $\mathcal{F}_{\mathsf{KRE}}$. In [3], the servers learn every prefix count in the computation of $t$-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 $\mathcal{F}_{\mathsf{MAX}}$; for computing $\mathcal{F}_{\mathsf{KRE}}$, 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) $\mathcal{T}$, a data set $X$ of $m$ inputs and a domain size of $n$ for each $x_j \in X$. In the offline phase for each input $x_j \in X$, $\mathcal{T}$ does the following:

- It picks a random value $\alpha_j \leftarrow_{\$} \mathbb{Z}_{2^n}$ and calculates $[\alpha_j]^{\mathsf{B}}$.
- It runs $\mathsf{IDPF.Gen}(1^\kappa, \alpha_j, (\mathbb{G}_1, \mathbf{1}), \cdots, (\mathbb{G}_n, \mathbf{1}))$ for each $i \in [n]$ using the algorithm from [3]. This gives two keys $(k_{j,0}, k_{j,1})$ and a public parameter $pp$.
- It chooses a random binary string $q \in \{0,1\}^n$ and computes $[q_i]^{\mathsf{A}}, i \in [n]$. Then for each $b \in \{0,1\}$, $\mathcal{T}$ sends

$$K_b = \{(k_{1,b}, [\alpha_1]_b^{\mathsf{B}}), \cdots, (k_{m,b}, [\alpha_m]_b^{\mathsf{B}})\} \text{ and}$$
$$Q_b = \{[q_1]_b^{\mathsf{A}}, \cdots, [q_n]_b^{\mathsf{A}}\}$$

**Input:** Input bit $b \in \{0,1\}$ identifying server $\mathsf{S}_b$ and selecting inputs $(K_b, Q_b)$ from the offline phase and $[x_j]_b^{\mathsf{B}}$ where $j \in [m]$ from the online phase.
**Output:** $[c]^{\mathsf{B}}$ where $c = f(X)$.

1: **for** $i = 1$ to $n$ **do**
2: $\quad [q_i]^{\mathsf{B}} \leftarrow \mathcal{F}_{\mathsf{A2B}}([q_i]^{\mathsf{A}})$
3: **end for**
4: **for** $j = 1$ to $m$ **do**
5: $\quad \mathsf{st}_{j,b}^0 \leftarrow k_{j,b}$
6: $\quad t^j \leftarrow \mathcal{F}_{\mathsf{reveal}}([q \oplus x_j \oplus \alpha_j]^{\mathsf{B}})$
7: **end for**
8: **for** $i = 1$ to $n$ **do**
9: $\quad$ **for** $j = 1$ to $m$ **do**
10: $\quad\quad (\mathsf{st}_{j,b}^i, [\beta_i^j]_b^{\mathsf{A}}) \leftarrow \mathsf{IDPF.EvalNext}(b, \mathsf{st}_{j,b}^{i-1}, \mathsf{pp}, t_i^j)$
11: $\quad$ **end for**
12: $\quad [\mu]^{\mathsf{A}} \leftarrow \sum_{j=1}^m [\beta_i^j]^{\mathsf{A}}$
13: $\quad$ **if** $i < n$ **then**
14: $\quad\quad (\delta_i, *) \leftarrow \Pi_f(i, [\mu]^{\mathsf{A}}, [q_i]^{\mathsf{A}}, *)$
15: $\quad\quad [c_i]^{\mathsf{B}} \leftarrow \delta_i \oplus [q_i]^{\mathsf{B}}$
16: $\quad$ **else**
17: $\quad\quad [c_i]^{\mathsf{B}} \leftarrow \Pi_f(i, [\mu]^{\mathsf{A}}, [q_i]^{\mathsf{A}}, *)$
18: $\quad$ **end if**
19: $\quad$ **if** $i < n$ and $\delta_i = 1$ **then**
20: $\quad\quad$ **for** $j = 1$ to $m$ **do**
21: $\quad\quad\quad (\mathsf{st}_{j,b}^i, [\beta_i^j]_b^{\mathsf{A}}) \leftarrow \mathsf{IDPF.EvalNext}(b, \mathsf{st}_{j,b}^{i-1}, \mathsf{pp}, \neg t_i^j)$
22: $\quad\quad$ **end for**
23: $\quad$ **end if**
24: **end for**
25: Outputs $[c]^{\mathsf{B}} \leftarrow [c_1]^{\mathsf{B}} \| \cdots \| [c_n]^{\mathsf{B}}$.

Figure 4: Our bit-wise computation framework.

to server $\mathsf{S}_b$.

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

$$\{[x_1]_b^{\mathsf{B}}, \cdots, [x_m]_b^{\mathsf{B}}\}$$

to the server $\mathsf{S}_b$ 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 $\mathsf{S}_b$ gets $K_b$ and $Q_b$ for each $b \in \{0,1\}$. In the online phase server $\mathsf{S}_b$ gets some parts of $X$ from all clients using Boolean secret sharing. When all inputs are there, the servers compute a masked string $t^j = q \oplus x_j \oplus \alpha_j$ (of size $n$) for each $j \in m$. This is the input for the I-DPF evaluation functions. Then we use $c_i$ for the $i^{\text{th}}$ bit of $f(X)$ for each $i \in [n]$. To compute the new target bit $c_i$ the servers use arithmetical secret sharing to count the prefixes of the string $c_1 \| \cdots \| q_i$ over $X$ (*CountPrefix*), which is indicated by $[\mu]^{\mathsf{A}}$ in Fig. 4. Then $[\mu]^{\mathsf{A}}$ 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 $\Pi_{\mathsf{BitMax}}$

| $i$ | Extra Input | Extra Output |
|---|---|---|
| 1 | $[q_{i+1}]^{\mathsf{A}}$ | $[v_i]^{\mathsf{A}}, [w_i]^{\mathsf{A}}$ |
| $(1, n)$ | $[q_{i+1}]^{\mathsf{A}}, [v_{i-1}]^{\mathsf{A}}, [w_{i-1}]^{\mathsf{A}}$ | $[v_i]^{\mathsf{A}}, [w_i]^{\mathsf{A}}$ |
| $n$ | $[v_{i-1}]^{\mathsf{A}}, [w_{i-1}]^{\mathsf{A}}$ | − |

1) *CountPrefix*: For each $j \in [m]$, $\mathsf{S}_b$ does the following:
   - It runs $\mathsf{IDPF.EvalNext}(b, \mathsf{st}_{j,b}^{i-1}, \mathsf{pp}, t_i^j)$ to get a new I-DPF state $\mathsf{st}_{j,b}^i$.
   -It also gets $[\beta_i^j]_b^{\mathsf{A}}$, which is either 0 or 1.
   -It adds up all $[\beta_i^j]_b^{\mathsf{A}}$ to get $[\mu]^{\mathsf{A}}$ which indicates the amount of the prefix string $c_1||\cdots||c_{i-1}||q_i$ that appears in $X$.
2) *ComputeBit*: After obtaining $[\mu]^{\mathsf{A}}$, this framework feeds $[\mu]^{\mathsf{A}}$ along with $[q_i]^{\mathsf{A}}$ 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 $\Pi_{\mathsf{BitMax}}$and $\Pi_{\mathsf{BitKre}}$ 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 $\mathsf{IDPF.EvalNext}(b, \mathsf{st}_{j,b}^{i-1}, \mathsf{pp}, \neg t_i^j)$ again for each $j \in [m]$. They get a new state $\mathsf{st}_{j,b}^i$ for each input. These states count the prefixes of the string $c_1\|\cdots\|c_{i-1}\|c_i$ 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 $[c]^{\mathsf{B}}$. 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 $i^{\text{th}}$ bit of the maximum and the k-th ranked element of $X$. While for the sub-protocol that compute the $i^{\text{th}}$ bit of the minimum of $X$, we attach it in Appendix C.1.

### 4.1. Realizing $\Pi_{\mathsf{BitMax}}$

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

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

To use this interface it is required to provide at least $i$ and $[\mu]^{\mathsf{A}}$ as inputs. At least $\delta$ and $[c]^{\mathsf{B}}$ will be returned as outputs. There may be other inputs or outputs which are indicated by the symbol $*$. The exact interface of $\Pi_{\mathsf{BitMax}}$ 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 $[v]^{\mathsf{A}}$ and $[w]^{\mathsf{A}}$ to keep track of the progress. $[v]^{\mathsf{A}}$ is the number of values in $X$ that could be the maximum, and $[w]^{\mathsf{A}}$ is the product of $[v]^{\mathsf{A}}$ and the probability that the $i$-th bit of the maximum is 0, which we denote by $[q_{i+1}]^{\mathsf{A}}$). We start with $[v]^{\mathsf{A}} = \mathsf{SS.share}(m)$, where $m$ is the size of $X$. We update both $[v]^{\mathsf{A}}$ and $[w]^{\mathsf{A}}$ 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 $[\mu]^{\mathsf{A}}$ 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 $q_i$:

- $q_i = 1$: We get $[w]^{\mathsf{A}} = [v]^{\mathsf{A}} \cdot (1 - [q_i]^{\mathsf{A}}) = [0]^{\mathsf{A}}$. Upon invoking $\mathcal{F}_{\mathsf{NZ}}([\mu]^{\mathsf{A}} - [w]^{\mathsf{A}})$ we perform a nonzero check on $[\mu]^{\mathsf{A}}$. If the check passes, this signifies that at least one string in $X$ starts with the prefix $p\|1$.
- $q_i = 0$: We have $[w]^{\mathsf{A}} = [v]^{\mathsf{A}} \cdot (1 - [q_i]^{\mathsf{A}}) = [v]^{\mathsf{A}}$. When invoking $\mathcal{F}_{\mathsf{NZ}}([\mu]^{\mathsf{A}} - [w]^{\mathsf{A}})$ the nonzero check is performed on $[\mu]^{\mathsf{A}} - [v]^{\mathsf{A}}$. 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 $i^{\text{th}}$ 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 $[v]^{\mathsf{A}}$ and $[w]^{\mathsf{A}}$ are updated correctly. More specifically if $\delta = 0 (c_i \equiv q_i)$, then $[v]^{\mathsf{A}} = [\mu]^{\mathsf{A}}$ indicates the current count of maximum candidates. Conversely if $\delta = 1 (c_i \equiv 1 - q_i)$, then $[v]^{\mathsf{A}} = [v]^{\mathsf{A}} - [\mu]^{\mathsf{A}}$ 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 $i^{\text{th}}$ iteration.

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

**Functionality:** $(\delta_i, *) \leftarrow \Pi_{\mathsf{BitMax}}(i, [\mu]^{\mathsf{A}}, [q_i]^{\mathsf{A}}, *)$
**Input:** $[\mu]^{\mathsf{A}}$ and $[q_i]^{\mathsf{A}}$, possibly other inputs based on the value of $i$.
**Output:** A bit $\delta_i$, $[v]^{\mathsf{A}}$ and $[w]^{\mathsf{A}}$ if $i < n$; otherwise $[c_i]^{\mathsf{B}}$.

1: **if** $i = 1$ **then**
2: $\quad [v]^{\mathsf{A}} \leftarrow \mathsf{SS.share}(m)$
3: $\quad [w]^{\mathsf{A}} \leftarrow m(1 - [q_i]^{\mathsf{A}})$
4: **end if**
5: $[c_i]^{\mathsf{B}} \leftarrow \mathcal{F}_{\mathsf{NZ}}([\mu - w]^{\mathsf{A}})$
6: **if** $i < n$ **then**
7: $\quad [v_0]^{\mathsf{A}} \leftarrow [\mu]^{\mathsf{A}}$
8: $\quad [v_1]^{\mathsf{A}} \leftarrow [v]^{\mathsf{A}} - [\mu]^{\mathsf{A}}$
9: $\quad [w_0]^{\mathsf{A}} \leftarrow [v_0]^{\mathsf{A}}(1 - [q_{i+1}]^{\mathsf{A}})$
10: $\quad [w_1]^{\mathsf{A}} \leftarrow [v_1]^{\mathsf{A}}(1 - [q_{i+1}]^{\mathsf{A}})$
11: **end if**
12: **if** $i < n$ **then**
13: $\quad \delta_i \leftarrow \mathcal{F}_{\mathsf{reveal}}([c_i \oplus q_i]^{\mathsf{B}})$
14: $\quad [v]^{\mathsf{A}} \leftarrow [v_{\delta_i}]^{\mathsf{A}}$
15: $\quad [w]^{\mathsf{A}} \leftarrow [w_{\delta_i}]^{\mathsf{A}}$
16: $\quad$ Outputs $(\delta_i, [v]^{\mathsf{A}}, [w]^{\mathsf{A}})$.
17: **else**
18: $\quad$ Outputs $[c_i]^{\mathsf{B}}$.
19: **end if**

#### 4.1.1. Rounds Optimization on Protocol 1.

In Protocol 1, assuming $\mathcal{F}_{\mathsf{NZ}}$ 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 4. However, for each $i \in [1, \cdots, n-1]$, we observe that it's possible to further reduce the overall communication rounds by combining the $\mathcal{F}_{\mathsf{reveal}}$ operation from $i^{\text{th}}$ iteration with the $\mathcal{F}_{\mathsf{NZ}}$ from the next $(i+1)^{\text{th}}$ 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 $[X]^{\mathsf{B}}$ before invoking Protocol 1. Specifically, instead of limiting ourselves to computing the prefix count of $q_1$ over $[X]^{\mathsf{B}}$, we expand our computation to include the prefix counts of $q_1$, $\neg q_1$, $q_1 \| q_2$, $\neg q_1 \| q_2$ respectively, thus obtain $[\mu]^{\mathsf{A}}, [\bar{\mu}]^{\mathsf{A}}, [\mu_0]^{\mathsf{A}}, [\mu_1]^{\mathsf{A}}$ 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 $\mathcal{F}_{\mathsf{reveal}}$ 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:

$$[c^0]^{\mathsf{B}} \leftarrow \mathcal{F}_{\mathsf{NZ}}([\mu_0]^{\mathsf{A}} - [w_0]^{\mathsf{A}}) \text{ and}$$
$$[c^1]^{\mathsf{B}} \leftarrow \mathcal{F}_{\mathsf{NZ}}([\mu_1]^{\mathsf{A}} - [w_1]^{\mathsf{A}}),$$

- They also prepare a list of candidate terms for the third comparison bit computation using $\mathcal{F}_{\mathsf{NZ}}$, which include:

$$[w_{00}]^{\mathsf{A}} \leftarrow (1 - [q_3]^{\mathsf{A}})[\mu_0]^{\mathsf{A}},$$
$$[w_{01}]^{\mathsf{A}} \leftarrow (1 - [q_3]^{\mathsf{A}})[\mu_1]^{\mathsf{A}},$$
$$[w_{10}]^{\mathsf{A}} \leftarrow (1 - [q_3]^{\mathsf{A}})([\mu - \mu_0]^{\mathsf{A}}),$$
$$[w_{11}]^{\mathsf{A}} \leftarrow (1 - [q_3]^{\mathsf{A}})([\bar{\mu} - \mu_1]^{\mathsf{A}}).$$

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 $\neg q_1 \| q_2$, servers do one more bit prefix count over $\neg q_1 \| q_2 \| q_3$, and from the I-DPF states by evaluating $\neg q_1$, servers do two more bits prefix count over $\neg q_1 \| \neg q_2 \| q_3$.

By following the established naming convention, we denote the obtained prefix count results over $c_1 \| q_2$, $c_1 \| \neg q_2$, $c_1 \| q_2 \| q_3$, and $c_1 \| \neg q_2 \| q_3$ as $[\mu]^{\mathsf{A}}$, $[\bar{\mu}]^{\mathsf{A}}$, $[\mu_0]^{\mathsf{A}}$, and $[\mu_1]^{\mathsf{A}}$ respectively.

In the third round, the servers reveal the second masked comparison bit $[c_{\delta_1} \oplus q_2]^{\mathsf{B}}$, 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 $(n-1)^{\text{th}}$ round, however by the last $n^{\text{th}}$ iteration, servers need only reveal $\delta_{n-1}$ and compute two candidate comparison bits. Without any further communication, this gives us $[c_{\delta_{n-1}}]^{\mathsf{B}}$, 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 $\mathcal{F}_{\mathsf{MAX}}$, which we refer to as $\Pi_{\mathsf{Max}}$. The communication complexity of $\Pi_{\mathsf{Max}}$ involves exactly $(m+1)n + 10n\kappa - 11\kappa$ bits, distributed over $n+1$ rounds, incurring a computational cost of $3mn$ invocations of $\mathsf{IDPF.EvalNext}$ 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 visa. 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 $\Pi_{\mathsf{BitKre}}$

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

$$(\delta_i, *) \leftarrow \Pi_{\mathsf{BitKre}}(i, [\mu]^{\mathsf{A}}, [q_i]^{\mathsf{A}}, [k]^{\mathsf{A}}, *).$$

This interface requires at least inputs of $i$ and $[\mu]^{\mathsf{A}}$ 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:

TABLE 4: Detailed operations of the rounds reduction method on $\Pi_{\mathsf{BitMax}}$

| 1st Round | 2nd Round | 3rd Round | $\cdots$ | $n^{\text{th}}$ Round |
|---|---|---|---|---|
| $[c_1]^{\mathsf{B}} \leftarrow \mathcal{F}_{\mathsf{NZ}}([\mu - w]^{\mathsf{A}})$ | $\delta_1 \leftarrow \mathcal{F}_{\mathsf{reveal}}([c_1 \oplus q_1]^{\mathsf{B}})$ | $\delta_2 \leftarrow \mathcal{F}_{\mathsf{reveal}}([c_{\delta_1} \oplus q_2]^{\mathsf{B}})$ | | $\delta_{n-1} \leftarrow \mathcal{F}_{\mathsf{reveal}}([c_{\delta_{n-2}} \oplus q_{n-1}]^{\mathsf{B}})$ |
| $[w_0]^{\mathsf{A}} \leftarrow [v_0]^{\mathsf{A}}[1-q_2]^{\mathsf{A}}$ | $[c^0]^{\mathsf{B}} \leftarrow \mathcal{F}_{\mathsf{NZ}}([\mu_0 - w_0]^{\mathsf{A}})$ | $[c^0]^{\mathsf{B}} \leftarrow \mathcal{F}_{\mathsf{NZ}}([\mu_0 - w_{0\delta_1}]^{\mathsf{A}})$ | | $[c^0]^{\mathsf{B}} \leftarrow \mathcal{F}_{\mathsf{NZ}}([\mu_0 - w_{0\delta_{n-2}}]^{\mathsf{A}})$ |
| $[w_1]^{\mathsf{A}} \leftarrow [v_1]^{\mathsf{A}}1[1-q_2]^{\mathsf{A}}$ | $[c^1]^{\mathsf{B}} \leftarrow \mathcal{F}_{\mathsf{NZ}}([\mu_1 - w_1]^{\mathsf{A}})$ | $[c^1]^{\mathsf{B}} \leftarrow \mathcal{F}_{\mathsf{NZ}}([\mu_1 - w_{1\delta_1}]^{\mathsf{A}})$ | | $[c^1]^{\mathsf{B}} \leftarrow \mathcal{F}_{\mathsf{NZ}}([\mu_1 - w_{1\delta_{n-2}}]^{\mathsf{A}})$ |
| – | $[w_{00}]^{\mathsf{A}} \leftarrow [1-q_3]^{\mathsf{A}}[\mu_0]^{\mathsf{A}}$ | $[w_{00}]^{\mathsf{A}} \leftarrow (1-[q_4]^{\mathsf{A}})[\mu_0]^{\mathsf{A}}$ | $\cdots$ | - |
| – | $[w_{01}]^{\mathsf{A}} \leftarrow [1-q_3]^{\mathsf{A}}[\mu_1]^{\mathsf{A}}$ | $[w_{01}]^{\mathsf{A}} \leftarrow [1-q_4]^{\mathsf{A}}[\mu_1]^{\mathsf{A}}$ | | - |
| – | $[w_{10}]^{\mathsf{A}} \leftarrow [1-q_3]^{\mathsf{A}}[\mu-\mu_0]^{\mathsf{A}}$ | $[w_{10}]^{\mathsf{A}} \leftarrow [1-q_4]^{\mathsf{A}}[\mu-\mu_0]^{\mathsf{A}}$ | | - |
| – | $[w_{11}]^{\mathsf{A}} \leftarrow [1-q_3]^{\mathsf{A}}[\bar{\mu}-\mu_1]^{\mathsf{A}}$ | $[w_{11}]^{\mathsf{A}} \leftarrow [1-q_4]^{\mathsf{A}}[\bar{\mu}-\mu_1]^{\mathsf{A}}$ | | - |

TABLE 5: Extra input/output of $\Pi_{\mathsf{BitKre}}$

| $i$ | Extra Input | Extra Output |
|---|---|---|
| 1 | $[q_{i+1}]^{\mathsf{A}}$ | $[v_i]^{\mathsf{A}}, [w_i]^{\mathsf{A}}$ |
| $(1, n)$ | $[q_{i+1}]^{\mathsf{A}}, [v_{i-1}]^{\mathsf{A}}, [w_{i-1}]^{\mathsf{A}}$ | $[v_i]^{\mathsf{A}}, [w_i]^{\mathsf{A}}$ |
| $n$ | $[v_{i-1}]^{\mathsf{A}}, [w_{i-1}]^{\mathsf{A}}$ | – |

1) To compute the correct bit $c$ of the target value using a secure comparison between $k$ and $\mu$.
2) To accurately update the internal variables $v$ and $k$ for use in the next iteration.

To achieve the first objective, a straightforward method involves two steps and two rounds of communication:

$$[r]^{\mathsf{A}} \leftarrow [q_i]^{\mathsf{A}} \cdot [\mu]^{\mathsf{A}} + (1 - [q_i]^{\mathsf{A}}) \cdot [v - \mu]^{\mathsf{A}},$$
$$[c_i]^{\mathsf{B}} \leftarrow \mathcal{F}_{\mathsf{LessEqualThan}}\big([k]^{\mathsf{A}}, [r]^{\mathsf{A}}\big).$$

Firstly $[r]^{\mathsf{A}}$ is computed based on the value of $[q_i]^{\mathsf{A}}$ indicating $\mathsf{IncPrefixCount}(p\|1)$. This costs one round of communication. Secondly the function $\mathcal{F}_{\mathsf{LessEqualThan}}$ is used to compare $[k]^{\mathsf{A}}$ and $[r]^{\mathsf{A}}$ and assign the result to $[c]^{\mathsf{B}}$. This costs a second round of communication.

Instead of using this straightforward method, we achieve the first objective in Protocol 2 using only one round of communication by computing:

$$[r_0]^{\mathsf{A}} \leftarrow [v]^{\mathsf{A}} - [\mu]^{\mathsf{A}}, [r_1]^{\mathsf{A}} \leftarrow [\mu]^{\mathsf{A}},$$
$$[c^0]^{\mathsf{B}} \leftarrow \mathsf{CondEval}\big([q_i]^{\mathsf{B}}, \wedge, f_{\leq}([k]^{\mathsf{A}}, [r_1]^{\mathsf{A}})\big),$$
$$[c^1]^{\mathsf{B}} \leftarrow \mathsf{CondEval}\big([1-q_i]^{\mathsf{B}}, \wedge, f_{\leq}([k]^{\mathsf{A}}, [r_0]^{\mathsf{A}})\big),$$
$$[c_i]^{\mathsf{B}} \leftarrow [c^0]^{\mathsf{B}} \oplus [c^1]^{\mathsf{B}}.$$

This ensures secure and correct derivation of the target bit $c$ in an efficient way. The key innovation is simultaneously comparing $[k]^{\mathsf{A}}$ with both $[r_0]^{\mathsf{A}}$ and $[r_1]^{\mathsf{A}}$, enabling two evaluations to occur within a single round of communication. The use of the primitive $\mathsf{CondEval}$ ensures that one of $c_0$ or $c_1$ will be zero, allowing for the oblivious and secure determination of $[c_i]^{\mathsf{B}}$ through the computation $[c_i]^{\mathsf{B}} = [c^0]^{\mathsf{B}} \oplus [c^1]^{\mathsf{B}}$.

To achieve the second objective let $[l]^{\mathsf{A}}$ and $[r]^{\mathsf{A}}$ denote the left and right branches of the current prefix query tree result. Then a straightforward way to update $[v]^{\mathsf{A}}$ and $[k]^{\mathsf{A}}$ would be to compute the following:

$$[k]^{\mathsf{A}} \leftarrow [c_i]^{\mathsf{A}} \cdot [k]^{\mathsf{A}} + (1 - [c_i]^{\mathsf{A}}) \cdot [k - r]^{\mathsf{A}},$$
$$[v]^{\mathsf{A}} \leftarrow [c_i]^{\mathsf{A}} \cdot [r]^{\mathsf{A}} + (1 - [c_i]^{\mathsf{A}}) \cdot [l]^{\mathsf{A}}.$$

In a typical scenario, converting from $[c_i]^{\mathsf{B}}$ to $[c_i]^{\mathsf{A}}$ and then executing the above operations would take two rounds. However Protocol 2 introduces an optimization. Given that our secure computation framework mandates the revelation of a bit $\delta_i = c_i \oplus q_i$, we can do both the revelation of $\delta_i$ and the computation of

$$[t_1]^{\mathsf{A}} = [q_i]^{\mathsf{A}} \cdot [l]^{\mathsf{A}},$$
$$[t_2]^{\mathsf{A}} = [q_i]^{\mathsf{A}} \cdot [r]^{\mathsf{A}}$$

in one round. We can express $c_i$ as $\delta_i \oplus q_i = \delta_i + q_i - 2\delta \cdot q_i$. Substituting this expanded form of $c_i$ and $[t_1]^{\mathsf{A}}, [t_2]^{\mathsf{A}}$ into the previous operations, we get the following equations for updating $[k]^{\mathsf{A}}$ and $[v]^{\mathsf{A}}$:

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

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

**Protocol 2** Bit-wise k-th ranked element computation

**Functionality:** $(\delta_i, [k]^{\mathsf{A}}, [v]^{\mathsf{A}}) \leftarrow \mathsf{BitKre}(i, [\mu]^{\mathsf{A}}, [q_i]^{\mathsf{A}}, [k]^{\mathsf{A}}, *)$
**Input:** An index $i \in [n]$, $[\mu]^{\mathsf{A}}$, $[b]^{\mathsf{A}}$ and the target ranking $[k]^{\mathsf{A}}$, it also inputs $[v]^{\mathsf{A}}$ if $i > 1$.
**Output:** A bit $\delta_i$, and the updated $[v]^{\mathsf{A}}, [k]^{\mathsf{A}}$; otherwise $[c_i]^{\mathsf{B}}$.

1: **if** $i = 1$ **then**
2: $\quad [v]^{\mathsf{A}} \leftarrow \mathcal{F}_{\mathsf{share}}(m)$
3: **end if**
4: $[q_i]^{\mathsf{B}} \leftarrow \mathcal{F}_{\mathsf{A2B}}([q_i]^{\mathsf{A}})$
5: $[r_0]^{\mathsf{A}} \leftarrow [v]^{\mathsf{A}} - [\mu]^{\mathsf{A}}$
6: $[r_1]^{\mathsf{A}} \leftarrow [\mu]^{\mathsf{A}}$
7: $[c^0]^{\mathsf{B}} \leftarrow \mathsf{CondEval}\big([q_i]^{\mathsf{B}}, \wedge, f_{\leq}([k]^{\mathsf{A}}, [r_1]^{\mathsf{A}})\big)$
8: $[c^1]^{\mathsf{B}} \leftarrow \mathsf{CondEval}\big([1-q_i]^{\mathsf{B}}, \wedge, f_{\leq}([k]^{\mathsf{A}}, [r_0]^{\mathsf{A}})\big)$
9: $[c_i]^{\mathsf{B}} \leftarrow [c^0]^{\mathsf{B}} \oplus [c^1]^{\mathsf{B}}$
10: **if** $i < n$ **then**
11: $\quad [r]^{\mathsf{A}} \leftarrow [q_i]^{\mathsf{A}} \cdot [r_1]^{\mathsf{A}} + [1-q_i]^{\mathsf{A}} \cdot [r_0]^{\mathsf{A}}$
12: $\quad [l]^{\mathsf{A}} \leftarrow [v]^{\mathsf{A}} - [r]^{\mathsf{A}}$
13: $\quad [t_1]^{\mathsf{A}} = [q_i]^{\mathsf{A}} \cdot [l]^{\mathsf{A}}$
14: $\quad [t_2]^{\mathsf{A}} = [q_i]^{\mathsf{A}} \cdot [r]^{\mathsf{A}}$
15: $\quad \delta_i \leftarrow \mathcal{F}_{\mathsf{reveal}}([c_i]^{\mathsf{B}} \oplus [q_i]^{\mathsf{B}})$
16: $\quad [k]^{\mathsf{A}} \leftarrow [k]^{\mathsf{A}} + (\delta_i - 1)[r]^{\mathsf{A}} + (1 - 2\delta_i)[t_2]^{\mathsf{A}}$
17: $\quad [v]^{\mathsf{A}} \leftarrow (1 - \delta_i)[l]^{\mathsf{A}} + (2\delta_i - 1)[t_1]^{\mathsf{A}} + \delta_i[r]^{\mathsf{A}} + (1 - 2\delta_i)[t_2]^{\mathsf{A}}$
18: $\quad$ Ouputs $(\delta_i, [k]^{\mathsf{A}}, [v]^{\mathsf{A}})$.
19: **else**
20: $\quad$ Outputs $[c_i]^{\mathsf{B}}$.
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

for computing $\mathcal{F}_{\mathsf{KRE}}$, which we refer to as $\Pi_{\mathsf{BitKre}}$. In Protocol 2 the first round corresponds to the operations in lines 7,8,11, while the second round corresponds to the computation of $[t_1]^{\mathsf{A}}, [t_2]^{\mathsf{A}}$ in lines 13,14,15. Thus, $\Pi_{\mathsf{BitKre}}$ 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 $\mathcal{O}(n)$, which is independent on the size of the secret input set $m$. This is different with comparison-based approaches, where amount of the 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 iteration. We denote by $\omega$ the batch size and assume $n$ is a multiple of $\omega$. Thus, if we compute the target string batch-wisely, with a lower number of iterations $d = n/\omega$ required, we could reduce the communication complexity from $\mathcal{O}(n)$ to $\mathcal{O}(d)$, potentially reducing communication rounds required in the end.

We denote by $[\vec{v}]^{\mathsf{A}}$ a resulting prefix count vector of a prefix list

$$p\|(\psi \oplus \mathcal{F}_{\mathsf{BDC}}(2^{\omega}-1,\omega)), \cdots, p\|(\psi \oplus \mathcal{F}_{\mathsf{BDC}}(0,\omega))$$

over $X$ for $\psi = q_{(i-1)\omega+1..i\omega}$, where $\psi$ is the $i^{\text{th}}$ batch of the random string $q$, thus, $[\vec{v}]^{\mathsf{A}}$ indicates an *unordered* prefix count vector. However, for meaningful computation with these values, we need somehow **convert this unordered vector $[\vec{v}]^{\mathsf{A}}$ to an ordered vector $[\vec{v'}]^{\mathsf{A}}$**, by ordered vector we mean a resulting vector over an ordered prefix list

$$p\|(\mathcal{F}_{\mathsf{BDC}}(2^{\omega}-1,\omega)), \cdots, p\|(\mathcal{F}_{\mathsf{BDC}}(0,\omega)).$$

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 $[M_i]^{\mathsf{A}}$, which implies that the servers cannot infer any information about $q$ from them. With $[M_i]^{\mathsf{A}}$ ready in the $i^{\text{th}}(i \in [d])$ 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 $[\vec{v}]^{\mathsf{A}}$ to $[\vec{v'}]^{\mathsf{A}}$. Specifically, this is done by computing

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

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 $[\vec{v}]^{\mathsf{A}}$ for the unordered prefix list, covered in lines 10-17.
   b) Invocation of the function $\Pi_f$ with $[\vec{v}]^{\mathsf{A}}$ and $[M_i]^{\mathsf{A}}$ 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 $c_1\|\cdots\|c_{i\omega}$, based on the value of $\delta_i$.
3) **Output Generation**: Output result bits $[c]^{\mathsf{B}}$ in line 26.

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 $\Pi_{\mathsf{BatchKre}}$

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

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

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

Integrating protocol 3 within the computational framework in Fig. 5 results in our finalized protocol for computing $\mathcal{F}_{\mathsf{KRE}}$, which we refer to as $\Pi_{\mathsf{BatchKre}}$. Note that it requires four rounds in Protocol 3, the first round involves converting $[\vec{v}]^{\mathsf{B}}$ to $[\vec{v'}]^{\mathsf{B}}$, the second round is to perform $\mathcal{F}_{\mathsf{LessEqualThan}}$ check. The third round computes $[b_t]^{\mathsf{B}}$ for $t \in [\tau]$, which aids in determining the target batch and in the final round $\sigma$ is revealed. Thus, $\Pi_{\mathsf{BatchKre}}$ involves exactly $(m+1)n + \frac{n\kappa}{\omega}\big(\tau + 2\tau^2 + 2(\tau-1) + 4(\frac{n}{\omega}-1)\big)$ bits, distributed over $1 + (4n)/\omega$ rounds, incurring a computational cost of $(2^{\omega}nm)/\omega$ invocations of IDPF.EvalNext on average on each server.

**Input:** Input bit $b \in \{0,1\}$ identifying server $\mathsf{S}_b$ and selecting inputs $(K_b, Q_b, [M]_b^{\mathsf{A}})$ from the offline phase and $[x_j]_b^{\mathsf{B}}$ where $j \in [m]$ from the online phase.
**Output:** $[c]^{\mathsf{B}}$ where $c = f(X)$.
1: $\tau \leftarrow 2^\omega, d \leftarrow n/\omega$
2: **for** $j = 1$ to $m$ **do**
3: $\quad [q_i]^{\mathsf{B}} \leftarrow \mathcal{F}_{\mathsf{A2B}}([q_i]^{\mathsf{A}})$
4: **end for**
5: **for** $j = 1$ to $m$ **do**
6: $\quad \mathsf{st}_{j,b}^0 \leftarrow k_{j,b}$
7: $\quad t^j \leftarrow \mathcal{F}_{\mathsf{reveal}}([q \oplus x_j \oplus \alpha_j]^{\mathsf{B}})$
8: **end for**
9: **for** $i = 1$ to $d$ **do**
10: $\quad$ **for** $t = 1$ to $\tau$ **do**
11: $\quad\quad \eta \leftarrow \mathcal{F}_{\mathsf{BDC}}(t-1, \omega) \oplus t^j[((i-1)\omega+1)..i\omega]$
12: $\quad\quad$ **for** $j = 1$ to $m$ **do**
13: $\quad\quad\quad (\hat{\mathsf{st}}_{j,b}^t, [\beta_t^j]_b^{\mathsf{A}}) \leftarrow \mathsf{IDPF.EvalNext}(b, \mathsf{st}_{j,b}^{i-1}, \mathsf{pp}, \eta)$
14: $\quad\quad$ **end for**
15: $\quad\quad [\mu_t]^{\mathsf{A}} \leftarrow \sum_{j=1}^m [\beta_t^j]^{\mathsf{A}}$
16: $\quad$ **end for**
17: $\quad [\vec{v}]^{\mathsf{A}} \leftarrow \{[\mu_1]^{\mathsf{A}}, \cdots, [\mu_\tau]^{\mathsf{A}}\}$
18: $\quad (\delta_i, *) \leftarrow \Pi_f(i, [\vec{v}]^{\mathsf{A}}, [M_i]^{\mathsf{A}}, *)$
19: $\quad \mathsf{idx} \leftarrow \mathcal{F}_{\mathsf{BC}}(\delta_i)$
20: $\quad$ **if** $i < n$ **then**
21: $\quad\quad$ **for** $j = 1$ to $m$ **do**
22: $\quad\quad\quad \mathsf{st}_{j,b}^{i\omega} \leftarrow \hat{\mathsf{st}}_{j,b}^{\mathsf{idx}}$
23: $\quad\quad$ **end for**
24: $\quad$ **end if**
25: **end for**
26: Outputs $[c]^{\mathsf{B}} \leftarrow (\delta_1 \| \cdots \| \delta_d) \oplus [q]^{\mathsf{B}}$.

Figure 5: Our batch-wise computation framework.

## 6. Experimental Evaluation

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

**Experiment Setting**. Except for $\Pi_{\mathsf{ICMax}}$ from MP-SPDZ [17] that is realized in c++, all other protocols (including basis and 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 $\mathcal{F}_{\mathsf{MAX}}$

**SOTA implementations for** $\mathcal{F}_{\mathsf{MAX}}$. As mentioned in the related work, to compute the maximum from a secrete set of size $m$, state-of-the-art solutions employ a secure comparison protocol internally, with which they perform pair-wise comparisons in $\mathcal{O}(\log m)$ 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 $\Pi_{\mathsf{ScMax1}}$, $\Pi_{\mathsf{ScMax2}}$ and $\Pi_{\mathsf{ICMax}}$. Different with our protocol $\Pi_{\mathsf{Max}}$ that inputs/outputs Boolean secret sharing, $\Pi_{\mathsf{ScMax1}}$, $\Pi_{\mathsf{ScMax2}}$ and $\Pi_{\mathsf{ICMax}}$ input the arithmetical secret sharing $[X]^{\mathsf{A}}$ of set $X$ and output the arithmetical sharing of the maximum of $X$. Most of the source codes for $\Pi_{\mathsf{ScMax1}}$ and $\Pi_{\mathsf{ScMax2}}$ 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 $\Pi_{\mathsf{ScMax1}}$ and $\Pi_{\mathsf{ScMax2}}$. For the complete implementation of $\Pi_{\mathsf{ICMax}}$, 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 $\Pi_{\mathsf{ScMax1}}$ and $\Pi_{\mathsf{ScMax2}}$.

We compared three existing solutions, $\Pi_{\mathsf{ScMax1}}$, $\Pi_{\mathsf{ScMax2}}$ and $\Pi_{\mathsf{ICMax}}$, with our protocol $\Pi_{\mathsf{Max}}$. 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 $5 \times 10^6$. For a fair comparison, in our implementation we have all protocols operate on a 32 bit ring except for $\Pi_{\mathsf{ScMax2}}$, 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 $\Pi_{\mathsf{ScMax2}}$ we use the default setting of MP-SPDZ [17] where $\lambda = 40$. Additionally, for the optimal performance in $\Pi_{\mathsf{ScMax1}}$ 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 $\kappa$ 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, $\Pi_{\mathsf{Max}}$, and the naive protocol, $\Pi_{\mathsf{ICMax}}$—with 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 $\Pi_{\mathsf{Max}}$ and $\Pi_{\mathsf{ICMax}}$ in terms of the time required for all the key

1. https://github.com/nann-cheng/FSS-KRE

TABLE 6: Comparison of offline/online costs for protocols $\Pi_{\mathsf{ScMax1}}$, $\Pi_{\mathsf{ScMax2}}$, $\Pi_{\mathsf{ICMax}}$, and $\Pi_{\mathsf{Max}}$.

| Phase | Measurements | Ref. | $10^3$ | $10^4$ | $10^5$ | $10^6$ | $2 \times 10^6$ | $5 \times 10^6$ |
|---|---|---|---|---|---|---|---|---|
| Offline | KeyGen Time (s) | $\Pi_{\mathsf{ICMax}}$ [5] | 0.008 | 0.093 | 0.864 | 8.465 | 16.61 | 37.08 |
| | | Our $\Pi_{\mathsf{Max}}$ | **0.007** | **0.064** | **0.611** | **5.573** | **12.122** | **24.65** |
| | Key size/Server (MB) | $\Pi_{\mathsf{ICMax}}$ [5] | **0.73** | 7.34 | 73.42 | 734.32 | 1468.65 | 3671.64 |
| | | Our $\Pi_{\mathsf{Max}}$ | 0.74 | **7.07** | **70.41** | **703.8** | **1407.6** | **3519.09** |
| Online | Rounds | $\Pi_{\mathsf{ScMax1}}$ [12] | 80 | 112 | 136 | – | – | – |
| | | $\Pi_{\mathsf{ScMax2}}$ [6] | 80 | 112 | 136 | – | – | – |
| | | $\Pi_{\mathsf{ICMax}}$ [5] | **20** | **28** | 34 | 40 | 42 | 46 |
| | | Our $\Pi_{\mathsf{Max}}$ | 32 | 32 | **32** | **32** | **32** | **32** |
| | Run Time (s) | $\Pi_{\mathsf{ScMax1}}$ [12] | 6.42 | 9.03 | 11.68 | – | – | – |
| | | $\Pi_{\mathsf{ScMax2}}$ [6] | 6.44 | 9.20 | 11.85 | – | – | – |
| | | $\Pi_{\mathsf{ICMax}}$ [5] | **1.66** | **2.35** | **3.34** | 9.80 | 17.35 | 40.44 |
| | | Our $\Pi_{\mathsf{Max}}$ | 2.60 | 2.83 | 3.60 | **8.45** | **14.43** | **33.40** |
| | Commu. Volume/server (MB) | $\Pi_{\mathsf{ScMax1}}$ [12] | 0.03 | 0.29 | 2.96 | – | – | – |
| | | $\Pi_{\mathsf{ScMax2}}$ [6] | 0.062 | 0.61 | 6.18 | – | – | – |
| | | $\Pi_{\mathsf{ICMax}}$ [5] | 0.011 | 0.114 | 1.144 | 11.444 | 22.88 | 57.22 |
| | | Our $\Pi_{\mathsf{Max}}$ | **0.005** | **0.038** | **0.370** | **3.696** | **7.392** | **18.47** |

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 $\Pi_{\mathsf{Max}}$, these are I-DPFs, while in $\Pi_{\mathsf{ICMax}}$, 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, $\Pi_{\mathsf{Max}}$ exhibits slightly better than $\Pi_{\mathsf{ICMax}}$ 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 $\Pi_{\mathsf{ICMax}}$, 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 $\lambda + n(\lambda + 2 + \ell) + 2\ell$ bits. Consequently, $\Pi_{\mathsf{Max}}$ requires $2\ell$ bits less for each function secret sharing key compared to that in $\Pi_{\mathsf{ICMax}}$.
- 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 $G : \{0,1\}^\lambda \rightarrow \{0,1\}^{2(2\lambda+1)}$ and three invocations of a pseudo-random group element converter $\mathsf{Convert}_{\mathbb{G}}$, 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 $\mathsf{Convert}_{\mathbb{G}}$ 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 $\Pi_{\mathsf{ScMax1}}$, $\Pi_{\mathsf{ScMax2}}$, $\Pi_{\mathsf{ICMax}}$, and $\Pi_{\mathsf{Max}}$. In Table 6, we record the actual communication rounds/volume and total online runtime in our benchmarks of all four protocols with varying input size $m$. For protocol $\Pi_{\mathsf{ScMax1}}$ and $\Pi_{\mathsf{ScMax2}}$, 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 $\Pi_{\mathsf{Max}}$ requires the least communication volume, roughly a third of that required in $\Pi_{\mathsf{ICMax}}$ and one-eighth of that required in $\Pi_{\mathsf{ScMax1}}$, this is consistent with our analysis presented in the related work. Additionally, in terms of concrete communication rounds required, our protocol $\Pi_{\mathsf{Max}}$ requires $n + 1 = 32$ rounds regardless of the value of $m$, while both $\Pi_{\mathsf{ScMax1}}$ and $\Pi_{\mathsf{ScMax1}}$ need $8\lceil\log(m)\rceil$ rounds and $\Pi_{\mathsf{ICMax}}$ 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, $\Pi_{\mathsf{ScMax1}}$ and $\Pi_{\mathsf{ScMax2}}$ were found to perform the least effectively, necessitating a higher volume of communication and significantly more rounds compared to the others. Interestingly, when comparing $\Pi_{\mathsf{ScMax1}}$ with $\Pi_{\mathsf{ScMax2}}$, 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 $\Pi_{\mathsf{ICMax}}$ to $\Pi_{\mathsf{Max}}$, 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 difference in terms of computation overhead within protocol $\Pi_{\mathsf{ICMax}}$ and $\Pi_{\mathsf{Max}}$. The evaluation algorithm of IC used in $\Pi_{\mathsf{ICMax}}$, requires $2n$ invocations of a pseudo-random generator $G : \{0,1\}^\lambda \rightarrow \{0,1\}^{2(2\lambda+1)}$, $2n$ invocations of a pseudo-random group element converter $\mathsf{Convert}_{\mathbb{G}}$ plus other arithmetical operation cost denoted as $T(2n)$, in contrast to $3n$ invocations of a *lighter* pseudo-random generator $G : \{0,1\}^\lambda \rightarrow \{0,1\}^{2\lambda+2}$, $3n$ invocation of a pseudo-random group element converter $\mathsf{Convert}_{\mathbb{G}}$ and other arithmetical operation cost denoted as $T(3n)$ in our evaluation of I-DPF used in $\Pi_{\mathsf{Max}}$. Considering both a pseudo-random generator $G$ and a pseudo-random group

TABLE 7: Comparison of offline/online costs for protocols $\Pi_{\mathsf{NaiveKre}}$, $\Pi_{\mathsf{BitKre}}$, and $\Pi_{\mathsf{BatchKre}}(\omega = 3)$.

| Phase | Measurements | Ref. | 10 | 50 | $10^3$ | $10^4$ | $10^5$ | $10^6$ |
|---|---|---|---|---|---|---|---|---|
| Offline | KeyGen Time (s) | $\Pi_{\mathsf{NaiveKre}}$ [5] | 0.001 | 0.009 | – | – | – | – |
| | | Our $\Pi_{\mathsf{BitKre}}$ | 0.001 | 0.002 | 0.006 | **0.048** | **0.477** | **4.595** |
| | | Our $\Pi_{\mathsf{BatchKre}}$ | 0.001 | 0.002 | 0.006 | 0.053 | 0.494 | 4.644 |
| | Key Size/Server (MB) | $\Pi_{\mathsf{NaiveKre}}$ [5] | **0.039** | 0.930 | – | – | – | – |
| | | Our $\Pi_{\mathsf{BitKre}}$ | 0.159 | 0.261 | 0.834 | 6.970 | 68.339 | 682.02 |
| | | Our $\Pi_{\mathsf{BatchKre}}$ | 0.083 | **0.110** | **0.758** | **6.895** | **68.264** | **681.95** |
| Online | Rounds | $\Pi_{\mathsf{NaiveKre}}$ [5] | 73 | 765 | – | – | – | – |
| | | Our $\Pi_{\mathsf{BitKre}}$ | 61 | 61 | 61 | 61 | 61 | 61 |
| | | Our $\Pi_{\mathsf{BatchKre}}$ | **51** | **51** | **51** | **51** | **51** | **51** |
| | Run Time (s) | $\Pi_{\mathsf{NaiveKre}}$ [5] | 5.84 | 61.732 | – | – | – | – |
| | | Our $\Pi_{\mathsf{BitKre}}$ | 4.93 | 4.93 | 5.00 | 5.32 | 6.63 | **12.61** |
| | | Our $\Pi_{\mathsf{BatchKre}}$ | **4.15** | **4.16** | **4.18** | **4.37** | **5.75** | 20.52 |
| | Commu. Volume/server (MB) | $\Pi_{\mathsf{NaiveKre}}$ [5] | **0.0007** | 0.014 | – | – | – | – |
| | | Our $\Pi_{\mathsf{BitKre}}$ | 0.038 | 0.038 | 0.041 | 0.073 | 0.395 | 3.614 |
| | | Our $\Pi_{\mathsf{BatchKre}}$ | 0.006 | **0.006** | **0.009** | **0.042** | **0.363** | **3.582** |

element converter $\mathsf{Convert}_{\mathbb{G}}$ realized with AES expansion, the overall expansion comparison within the corresponding evaluation algorithm in $\Pi_{\mathsf{ICMax}}$ and $\Pi_{\mathsf{Max}}$ are $8n\lambda + 2n(\ell+2)$, $6n\lambda + 3n(\ell+2)$. Thus, when comparing the cost of $\Pi_{\mathsf{ICMax}}$ to $\Pi_{\mathsf{Max}}$, 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 $m \in \{10^3, 10^4, 10^5\}$, we see from Tab. 6 that $\Pi_{\mathsf{ICMax}}$ performs better online efficiency than $\Pi_{\mathsf{Max}}$, 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 $\Pi_{\mathsf{Max}}$ than $\Pi_{\mathsf{ICMax}}$, which cannot only be explained by less communication rounds used, *e.g.,* when $m = 5 \times 10^6$, $\Pi_{\mathsf{Max}}$ requires almost seven seconds less than that in $\Pi_{\mathsf{ICMax}}$ in terms of online run-time, while meanwhile the communication rounds difference between them two contributes to only $14 \times 80\mathtt{ms} = 1.12\mathtt{s}$ saving in theory.

## 6.2. Evaluation for $\mathcal{F}_{\mathsf{KRE}}$

**SOTA implementations for $\mathcal{F}_{\mathsf{KRE}}$.** Given the absence of efficient solutions for $\mathcal{F}_{\mathsf{KRE}}$, we constructed $\Pi_{\mathsf{NaiveKre}}$, a straightforward comparison-based solution for computing the $k$-th ranked element from a secret shared set $X$. It takes the arithmetical secret sharing $[k]^{\mathsf{A}}$ 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 $X$.

$\Pi_{\mathsf{NaiveKre}}$ begins by performing a full sorting over $[X]^{\mathsf{A}}$ to obtain a vector $[V]^{\mathsf{A}}$ in $\mathcal{O}(m \log m)$ iterations. Subsequently, it identifies the $k$-th ranked element by computing the inner product between $[V]^{\mathsf{A}}$ and another vector $[I]^{\mathsf{A}}$. Here, $[I]^{\mathsf{A}}$ is of size $m$, where the $i^{\text{th}}$ ($i \in [m]$) element is the zero-check result over $i - [k]^{\mathsf{A}}$. Notably, the input index $[k]^{\mathsf{A}}$ 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 $\Pi_{\mathsf{NaiveKre}}$, we use the integer interval containment function secret sharing as the underlying secure comparison implementation, as the same as in $\Pi_{\mathsf{ICMax}}$.

In Table 7, we evaluate the performance of our protocols $\Pi_{\mathsf{BitKre}}$ and $\Pi_{\mathsf{BatchKre}}$ against $\Pi_{\mathsf{NaiveKre}}$. Note that we set the batch size $\omega = 3$ in the benchmark of $\Pi_{\mathsf{BatchKre}}$. 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 $\Pi_{\mathsf{NaiveKre}}$, 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, $\Pi_{\mathsf{BitKre}}$ and $\Pi_{\mathsf{BatchKre}}$ outperform $\Pi_{\mathsf{NaiveKre}}$ 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 $\Pi_{\mathsf{NaiveKre}}$. 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 $\mathcal{F}_{\mathsf{Kre}}$, 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


[1] Surya Addanki, Kevin Garbe, Eli Jaffe, Rafail Ostrovsky, and Antigoni Polychroniadou. Prio+: Privacy preserving aggregate statistics via boolean shares. *Cryptology ePrint Archive*, 2021.

[2] Gagan Aggarwal, Nina Mishra, and Benny Pinkas. Secure computation of the median (and other elements of specified ranks). *Journal of Cryptology*, 23(3):373–401, July 2010.

[3] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Lightweight techniques for private heavy hitters. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 762–776. IEEE, 2021.

[4] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 1292–1303. ACM Press, October 2016.

[5] Elette Boyle, Niv Gilboa, and Yuval Ishai. Secure computation with preprocessing via function secret sharing. In *Theory of Cryptography: 17th International Conference, TCC 2019, Nuremberg, Germany, December 1–5, 2019, Proceedings, Part I 17*, pages 341–371. Springer, 2019.

[6] Octavian Catrina and Sebastiaan De Hoogh. Improved primitives for secure multiparty integer computation. In *Security and Cryptography for Networks: 7th International Conference, SCN 2010, Amalfi, Italy, September 13-15, 2010. Proceedings 7*, pages 182–199. Springer, 2010.

[7] Gowri R Chandran, Carmit Hazay, Robin Hundt, and Thomas Schneider. Comparison-based mpc in star topology (full version). *Cryptology ePrint Archive*, 2022.

[8] Nan Cheng, Melek Önen, Aikaterini Mitrokotsa, Oubaïda Chouchane, Massimiliano Todisco, and Alberto Ibarrondo. Privacy-preserving cosine similarity computation with malicious security applied to biometric authentication. Cryptology ePrint Archive, Paper 2023/1684, 2023. https://eprint.iacr.org/2023/1684.

[9] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *14th USENIX symposium on networked systems design and implementation (NSDI 17)*, pages 259–282, 2017.

[10] Geoffroy Couteau. New protocols for secure equality test and comparison. In Bart Preneel and Frederik Vercauteren, editors, *ACNS 18*, volume 10892 of *LNCS*, pages 303–320. Springer, Heidelberg, July 2018.

[11] Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. Multiparty computation from threshold homomorphic encryption. In Birgit Pfitzmann, editor, *EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 280–299. Springer, Heidelberg, May 2001.

[12] Ivan Damgård, Daniel Escudero, Tore Frederiksen, Marcel Keller, Peter Scholl, and Nikolaj Volgushev. New primitives for actively-secure mpc over rings with applications to private machine learning. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1102–1120. IEEE, 2019.

[13] Ivan Damgård, Martin Geisler, and Mikkel Krøigaard. Efficient and secure comparison for on-line auctions. In Josef Pieprzyk, Hossein Ghodosi, and Ed Dawson, editors, *ACISP 07*, volume 4586 of *LNCS*, pages 416–430. Springer, Heidelberg, July 2007.

[14] Juan A. Garay, Berry Schoenmakers, and José Villegas. Practical and secure solutions for integer comparison. In Tatsuaki Okamoto and Xiaoyun Wang, editors, *PKC 2007*, volume 4450 of *LNCS*, pages 330–342. Springer, Heidelberg, April 2007.

[15] Peng Hu, Yongli Wang, Bei Gong, Yongjian Wang, Yanchao Li, Ruxin Zhao, Hao Li, and Bo Li. A secure and lightweight privacy-preserving data aggregation scheme for internet of vehicles. *Peer-to-Peer Networking and Applications*, 13:1002–1013, 2020.

[16] Swanand Kadhe, Nived Rajaraman, O Ozan Koyluoglu, and Kannan Ramchandran. Fastsecagg: Scalable secure aggregation for privacy-preserving federated learning. *arXiv preprint arXiv:2009.11248*, 2020.

[17] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1575–1590. ACM Press, November 2020.

[18] Bruno Rossi, Stanislav Chren, Barbora Buhnova, and Tomas Pitner. Anomaly detection in smart grid data: An experience report. In *2016 ieee international conference on systems, man, and cybernetics (smc)*, pages 002313–002318. IEEE, 2016.

[19] Anselme Tueno, Florian Kerschbaum, Stefan Katzenbeisser, Yordan Boev, and Mubashir Qureshi. Secure computation of the $k^{\text{th}}$-ranked element in a star network. In Joseph Bonneau and Nadia Heninger, editors, *FC 2020*, volume 12059 of *LNCS*, pages 386–403. Springer, Heidelberg, February 2020.

[20] Jing Wang, Libing Wu, Sherali Zeadally, Muhammad Khurram Khan, and Debiao He. Privacy-preserving data aggregation against malicious data mining attack for iot-enabled smart grid. *ACM Transactions on Sensor Networks (TOSN)*, 17(3):1–25, 2021.

[21] Yuan Zhang, Qingjun Chen, and Sheng Zhong. Efficient and privacy-preserving min and $k$ th min computations in mobile sensing systems. *IEEE Transactions on Dependable and Secure Computing*, 14(1):9–21, 2015.

# A. Security Analysis

We consider a static corruption model where a Honest-but-Curious adversary $\mathcal{A}$ chooses one of the two computing parties $\mathsf{S}_0, \mathsf{S}_1$ before the execution of the computations.

In the subsequent discussion, we focus exclusively on a formal security proof of our protocol $\Pi_{\mathsf{Max}}$. The security analysis for other protocols $\Pi_{\mathsf{Kre1}}, \Pi_{\mathsf{Kre2}}$ follow a similar structure, and we omit it here.

## A.1. Security of $\Pi_{\mathsf{Max}}$

In the following we define an ideal functionality $\mathcal{F}_{\mathsf{Max}}$ that interacts with $\mathsf{S}_0, \mathsf{S}_1$, and the adversary $\mathcal{A}$ is parameterized by a public known function $f_{\mathsf{Max}}(X)$.

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

Also, we define another ideal functionality $\mathcal{F}_{\mathsf{BitMax}}$ works almost the same as $\mathcal{F}_{\mathsf{Max}}$, differently it inputs an additional index $i \in [n]$ and outputs $[y_i]_b^{\mathsf{B}}$ instead of $[y]_b^{\mathsf{B}}$ to $\mathsf{S}_b$ for $b \in \{0, 1\}$.

**Theorem 1.** *There is a PPT algorithm simulator* $\mathsf{Sim_b}$ *that realizes ideal functionality* $\mathcal{F}_{\mathsf{Setup}}$*, which inputs only* $(1^\kappa, (n, \mathbb{G}_1, \cdots, \mathbb{G}_n))$ *and outputs* $(K_b^*, Q_b^*)$*, such that the output is computationally indistinguishable with 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 $K_b^*$ that are computationally indistinguishable with the real world output $K_b$. Also, $Q_b$ is perfectly indistinguishable with $Q_b^*$ when the simulator $\mathsf{Sim_b}$ chooses $Q_b^* \leftarrow_\$ \mathbb{G}^n$. □

**Theorem 2.** *In the* $(\mathcal{F}_{\mathsf{Setup}}, \mathcal{F}_{\mathsf{NZ}})$*-hybrid model, there is a PPT simulator* $\mathsf{Sim_b}$ *that* $\forall X \in \mathbb{Z}_{2^n}^m$ *and function* $f_{\mathsf{BitMax}}(X, i) : \{0,1\}^{n \times m} \rightarrow \{0, 1\}$*,* $\mathsf{S_b}$ *realizes ideal functionality* $\mathcal{F}_{\mathsf{BitMax}}$*, such that its output is computationally indistinguishable with the real execution of protocol* $\Pi_{\mathsf{BitMax}}$*.*

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

- $\mathsf{S_b}$ uniformly generates $\delta_i^* \leftarrow_\$ \{0, 1\}$, which is perfectly indistinguishable with $\mathsf{S}_b$'s view $\delta_i$ in line 13 of Fig. 4.

- As for all $j \in [m], i \in [n]$ that, $\mathsf{S_b}$ 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 $\Pi_{\mathsf{Max}}$.

For all other secret share within the real protocol execution, $\mathsf{S_b}$ generates random group elements which are perfectly indistinguishable to what in the real protocol execution. □

**Definition 1.** *(Security) There is a PPT simulator* $\mathsf{Sim_b}$ *such that* $\forall X \in \mathbb{Z}_{2^n}^m$ *and function* $f_{\mathsf{Max}}(X) : \{0,1\}^{n\times m} \to \{0,1\}^n$, $\mathsf{S_b}$ *realizes the ideal functionality* $\mathcal{F}_{\mathsf{Max}}$*, such that its behavior is computationally indistinguishable from a real world execution of protocol* $\Pi_{\mathsf{Max}}$ *in the presence of a semi-honest adversary* $\mathcal{A}$.

**Theorem 3.** *In the* $(\mathcal{F}_{\mathsf{Setup}}, \mathcal{F}_{\mathsf{BitMax}})$*-hybrid model, protocol* $\Pi_{\mathsf{Max}}$ *securely realize the functionality* $\mathcal{F}_{\mathsf{Max}}$.

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

- $\mathsf{Sim_b}$ randomly select $t^j \leftarrow_\$ \{0,1\}^n$ simulating $\mathsf{S}_b$'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 $\Pi_{\mathsf{Max}}$, it holds that the view of $\mathcal{A}$ in our simulation is identical to that in the real execution of protocol $\Pi_{\mathsf{Max}}$;
- $\mathsf{Sim_b}$ passes $\sigma_i^*$ to $\mathcal{A}$ simulating $\mathsf{S}_b$'s view in line 13 of Fig. 4.

□

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

## B. Full construction of $\Pi_{\mathsf{BatchKre}}$

In Protocol 3, we present the full construction of $\Pi_{\mathsf{BatchKre}}$ that securely computes the $i^{th}$ batch of the KRE of $X$. Here, the servers start by computing the ordered vector $[\vec{v'}]^{\mathsf{A}}$ using the conversion matrix $[M]^{\mathsf{A}}$ and the initial vector $[\vec{v}]^{\mathsf{A}}$. Subsequently, $\forall t \in [\tau]$, the secret sharing $[v_t^*]^{\mathsf{A}}$ is computed as $\sum_{\ell=1}^{t}[v'_\ell]^{\mathsf{A}}$, a $\mathcal{F}_{\mathsf{LessEqualThan}}$ comparison is executed on each $[v_t^*]^{\mathsf{A}}$ producing a comparison result $[c_t]^{\mathsf{A}}$. Denote $\vec{v^*}$ as $\{v_1^*, \cdots, v_\tau^*\}$ and $\vec{c}$ as $\{c_1, \cdots, c_\tau\}$. Since the elements in $\vec{v^*}$ are monotonically increasing, the vector $\vec{c}$ initially contains zeros. Starting from a specific index in $\vec{c}$ , all subsequent elements will have a value of one. Let $\xi$ be the index of the first nonzero element in $\vec{c}$. This marks target batch containing the corresponding prefix of the $k^{\text{th}}$ ranked element of set $X$. To identify $\xi$ the following conditional equation is utilized for each $t$ in $[\tau]$:

$$\begin{cases} [b_t]^{\mathsf{A}} \leftarrow [c_t]^{\mathsf{A}} \text{ if } t = 1, \\ [b_t]^{\mathsf{A}} \leftarrow [c_t]^{\mathsf{A}} \cdot (1 - [c_{t-1}]^{\mathsf{A}}) \text{ if } t > 1. \end{cases}$$

FUNCTIONALITY $\mathcal{F}_{\mathsf{ConvMatrix}}(q)$:
**Input:** A binary string $q$ of length $\omega$.
**Output:** A $\tau \times \tau$ matrix $M$.

```
1: Set M as an τ × τ matrix initiated as all zeros.
2: Q ← {q1 q2 ··· qω, ··· , (1 − q1)(1 − q2) ··· (1 − qω)}
3: for i = 1 to τ do
4:     η ← F_BDC(τ − i, ω)
5:     for j = 1 to τ do
6:         ξ ← F_BDC(τ − j, ω)
7:         s ← {0}^ω
8:         for k = 1 to ω do
9:             if η_k = 1 then
10:                s_k ← ξ_k
11:            else
12:                s_k ← ¬ξ_k
13:            end if
14:        end for
15:        M[i][j] ← Q_{τ−F_BC(s)}
16:    end for
17: end for
18: Outputs M.
```

Figure 6: Conversion matrix generation.

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

$$[\delta_i]^{\mathsf{B}} \leftarrow [\delta_i]^{\mathsf{B}} \oplus (s_i \wedge \mathcal{F}_{\mathsf{A2B}}([b_t]^{\mathsf{A}})), \text{ for } i \in [\omega].$$

And this also enables us to update $k$ with

$$[k]^{\mathsf{A}} \leftarrow [b_1]^{\mathsf{A}} \cdot [k]^{\mathsf{A}} + \sum_{t=2}^{\tau}[b_t]^{\mathsf{A}} \cdot ([k]^{\mathsf{A}} - [v_{t-1}^*]^{\mathsf{A}}).$$

## 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 $[a]^{\mathsf{B}}$ is equal to the maximum of a set $[X]^{\mathsf{B}}$, 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 $[w]^{\mathsf{A}}$. Concerning the latter, there are two instances where we prepare $[w]^{\mathsf{A}}$. First, during initialization when $i = 1$, we compute $[w]^{\mathsf{A}} = m \cdot [b_0]^{\mathsf{A}}$. Second, for $i \in (1, n)$, we use following computations instead:

$$[w_0]^{\mathsf{A}} \leftarrow [v_0]^{\mathsf{A}} \cdot [b_1]^{\mathsf{A}},$$
$$[w_1]^{\mathsf{A}} \leftarrow [v_1]^{\mathsf{A}} \cdot [b_1]^{\mathsf{A}}.$$

These adjustments ensure the extraction of the $i^{\text{th}}$ bit of the minimum value of $X$, as we outline below.

**Protocol 3** The batch-wise k-th ranked element protocol

**Functionality:** $(\delta, *) \leftarrow \Pi_{\mathsf{BatchKre}}(i, [\vec{v}]^{\mathsf{A}}, [M]^{\mathsf{A}}, [q]^{\mathsf{B}}, [k]^{\mathsf{A}}, *)$
**Input:** A vector $\vec{v}$ of length $\tau$, a binary string $q$ of length $\omega$ and $[k]^{\mathsf{A}}$.
**Output:** A binary string $\sigma$ of length $\omega$, and an updated $[k]^{\mathsf{A}}$ if $i < d$.

1: $[\vec{v'}]^{\mathsf{A}} \leftarrow [\vec{v}]^{\mathsf{A}} \times [M]^{\mathsf{A}}$ ▷ Compute ordered prefix query results.
2: **for** $t = 1$ to $\tau$ **do**
3: $[v_t^*]^{\mathsf{A}} \leftarrow \sum_{\ell=1}^{t} [v'_\ell]^{\mathsf{A}}$
4: $[c_t]^{\mathsf{A}} \leftarrow \mathcal{F}_{\mathsf{LessEqualThan}}([k]^{\mathsf{A}}, [v_t^*]^{\mathsf{A}})$ ▷ Compute ordered comparison bits.
5: **end for**
6: $[\sigma]^{\mathsf{B}} \leftarrow [q]^{\mathsf{B}}$
7: **for** $t = 1$ to $\tau$ **do**
8: $s \leftarrow \mathcal{F}_{\mathsf{BDC}}(\tau - t, \omega)$
9: **if** $t = 1$ **then**
10: $[b_t]^{\mathsf{A}} \leftarrow [c_t]^{\mathsf{A}}$
11: **else**
12: $[b_t]^{\mathsf{A}} \leftarrow [c_t]^{\mathsf{A}} \cdot (1 - [c_{t-1}]^{\mathsf{A}})$
13: **end if**
14: **for** $i = 1$ to $\omega$ **do**
15: $[\sigma_i]^{\mathsf{B}} \leftarrow [\sigma_i]^{\mathsf{B}} \oplus (s_i \wedge \mathcal{F}_{\mathsf{A2B}}([b_t]^{\mathsf{A}}))$ ▷ Extract target batch.
16: **end for**
17: **end for**
18: $\sigma \leftarrow \mathcal{F}_{\mathsf{reveal}}([\sigma]^{\mathsf{B}})$
19: **if** $i < d$ **then**
20: $[k]^{\mathsf{A}} \leftarrow [b_1]^{\mathsf{A}} \cdot [k]^{\mathsf{A}} + \sum_{t=2}^{\tau} [b_t]^{\mathsf{A}} \cdot ([k]^{\mathsf{A}} - [v_{t-1}^*]^{\mathsf{A}})$ ▷ Update $[k]^{\mathsf{A}}$ for use in the next iteration.
21: Outputs $(\sigma, [k]^{\mathsf{A}})$.
22: **else**
23: Outputs $\sigma$.
24: **end if**

- If $q_i = 1$, We get $[w]^{\mathsf{A}} = [v]^{\mathsf{A}} \cdot [q_i]^{\mathsf{A}} = [v]^{\mathsf{A}}$. Upon invoking $\mathcal{F}_{\mathsf{ZeroCheck}}([\mu]^{\mathsf{A}} - [w]^{\mathsf{A}})$, we perform a zero check on $[\mu]^{\mathsf{A}} - [v]^{\mathsf{A}}$. If this check passes, it implies that all strings in $X$ start with the prefix $p\|1$.
- Otherwise, We have $[w]^{\mathsf{A}} = [v]^{\mathsf{A}} \cdot [q_i]^{\mathsf{A}} = [0]^{\mathsf{A}}$. When invoking $\mathcal{F}_{\mathsf{ZeroCheck}}([\mu]^{\mathsf{A}} - [w]^{\mathsf{A}})$, the zero check is performed on $[\mu]^{\mathsf{A}}$. If this check passes, it means that all remaining candidates for the minimum start with the prefix $p\|1$.

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 $\Pi_{\mathsf{MaxVry}}$

To verify whether a Boolean secret-shared value $[a]^{\mathsf{B}}$ is equal to the maximum of a set $[X]^{\mathsf{B}}$, a straightforward method might involve using protocol 1 to first calculate the maximum $[c]^{\mathsf{B}}$ and then check for equality between $[c]^{\mathsf{B}}$ and $[a]^{\mathsf{B}}$. Yet, this direct approach leads to communication rounds 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 $\Pi_{\mathsf{MaxVry}}$, which verifies the maximum value within a mere three rounds of communication. Protocol $\Pi_{\mathsf{MaxVry}}$ inputs $[X]^{\mathsf{B}}$ and $[a]^{\mathsf{B}}$, and outputs $[c]^{\mathsf{B}}$, which indicates if $a$ is the maximal value in $X$. The protocol achieves this by computing $[c]^{\mathsf{B}} = [c_0]^{\mathsf{B}} \wedge [c_1]^{\mathsf{B}}$, 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 $\Pi_{\mathsf{MaxVry}}$ that perform this maximum verification in just three communication rounds. The protocol inputs $[X]^{\mathsf{B}}$ and $[a]^{\mathsf{B}}$, and outputs $[c]^{\mathsf{B}}$, where $c$ indicates if $a$ is the maximum element in $X$. In essence, we compute $[c]^{\mathsf{B}} = [c_0]^{\mathsf{B}} \wedge [c_1]^{\mathsf{B}}$ 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:** $[y]^{\mathsf{B}} \leftarrow \Pi_{\mathsf{MaxVry}}([X]^{\mathsf{B}}, \mathcal{K}, [a]^{\mathsf{B}})$
**Input:** For each $b \in \{0,1\}$, $\mathsf{S}_b$ inputs $K_b$ in the offline phase; $[a]^{\mathsf{B}}$ where $a \in \mathbb{Z}_{2^n}$ and $[x_j]_b^{\mathsf{B}} (j \in [m])$ in the online phase.
**Output:** A Boolean Secret Sharing $[c]^{\mathsf{B}}$ where

$$c = \begin{cases} 1 & \text{If } a = \mathcal{F}_{\mathsf{Max}}(X), \\ 0 & \text{Otherwise.} \end{cases}$$

1: **for** $j = 1$ to $m$ **do**
2: **for** $b = 0$ to $1$ **do**
3: $\mathsf{st}_{j,b}^0 \leftarrow k_{j,b}$
4: **end for**
5: $t^j \leftarrow \mathcal{F}_{\mathsf{reveal}}([a \oplus x_j \oplus \alpha_j]^{\mathsf{B}})$
6: **end for**
7: $[w]^{\mathsf{A}} \leftarrow \mathcal{F}_{\mathsf{Share}}(0)$
8: $[\beta]^{\mathsf{A}} \leftarrow \mathcal{F}_{\mathsf{Share}}(0)$
9: **for** $i = 1$ to $n$ **do**
10: $[a_i]^{\mathsf{A}} \leftarrow \mathcal{F}_{\mathsf{B2A}}([a_i]^{\mathsf{B}})$
11: **for** $j = 1$ to $m$ **do**
12: **for** $b = 0$ to $1$ **do**
13: $(\mathsf{st}_{j,b}^{i,0}, [\beta_{i,0}^j]_b^{\mathsf{A}}) \leftarrow \mathsf{IDPF.EvalNext}(b, \mathsf{st}_{j,b}^{i-1,0}, \mathsf{pp}, t_i^j)$
14: $v_i^j \leftarrow 1 \oplus \neg t_i^j$
15: $(\mathsf{st}_{j,b}^{i,1}, [\beta_{i,1}^j]_b^{\mathsf{A}}) \leftarrow \mathsf{IDPF.EvalNext}(b, \mathsf{st}_{j,b}^{i-1,0}, \mathsf{pp}, v_i^j)$
16: **end for**
17: **end for**
18: $[\mu_i]^{\mathsf{A}} \leftarrow \sum_{j=1}^{m} [\beta_{i,1}^j]^{\mathsf{A}}$
19: **end for**
20: $[\beta]^{\mathsf{A}} \leftarrow \sum_{j=1}^{m} [\beta_{n,0}^j]^{\mathsf{A}}$
21: $[c_0]^{\mathsf{B}} \leftarrow \mathcal{F}_{\mathsf{NZ}}([\beta]^{\mathsf{A}})$
22: $[w]^{\mathsf{A}} \leftarrow \sum_{i=1}^{n} \left((1 - [a_i]^{\mathsf{A}}) \cdot [\mu_i]^{\mathsf{A}}\right)$
23: $[c]^{\mathsf{B}} \leftarrow \mathsf{CondEval}\left([c_0]^{\mathsf{B}}, \wedge, f_{=0}([w]^{\mathsf{A}})\right)$
24: Outputs $[c]^{\mathsf{B}}$.

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

i. In line 10, servers run $\mathcal{F}_{\mathsf{B2A}}$ on each bit of $[a]^{\mathsf{B}}$ and obtain $[a_i]^{\mathsf{A}}$, which are used later for calculating $[w]^{\mathsf{A}}$.
ii. In lines 21 and 22, where servers compute $[c_0]^{\mathsf{B}}$ and $[w]^{\mathsf{A}}$.
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 $[w]^{\mathsf{A}}$ should be adjusted to:

$$[w]^{\mathsf{A}} \leftarrow [w]^{\mathsf{A}} + [a_i]^{\mathsf{A}} \cdot [\mu_i]^{\mathsf{A}}$$

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 $i^{\text{th}}$ target bit is identified by comparing $\mathsf{PrefixCount}(p\|1)$ and $\mathsf{PrefixCount}(p\|0)$, where $p$ denotes the determined target prefix.