Secure Computation of the k^{th}-Ranked Element in a Star Network

Anselme Tueno[1], Florian Kerschbaum[2], Stefan Katzenbeisser[3], Yordan Boev[1], and Mubashir Qureshi[1]

[1] SAP SE, Germany
[2] University of Waterloo, Canada
[3] University of Passau, Germany

Abstract. We consider the problem of securely computing the k^{th}-ranked element in a sequence of n private integers distributed among n parties. The k^{th}-ranked element (e.g., minimum, maximum, median) is of particular interest in collaborative benchmarking and auctions. Previous secure protocols for the k^{th}-ranked element require a communication channel between each pair of parties. A server model naturally fits with the client-server architecture of Internet applications in which clients are connected to the server and not to other clients. It simplifies secure computation by reducing the number of rounds and improves its performance and scalability. In this paper, we propose different approaches for privately computing the k^{th}-ranked element in the server model, using either garbled circuits or threshold homomorphic encryption. Our schemes have a constant number of rounds and can compute the k^{th}-ranked element within seconds for up to 50 clients in a WAN.

Keywords: k^{th}-Ranked Element, Garbled Circuit, Homomorphic Encryption

1 Introduction

Given n parties each holding a private integer, we consider the problem of securely computing the k^{th}-ranked element (KRE) of these n integers. This is a secure multiparty computation (SMC) where several parties wish to compute a function on their private input while revealing only the output of the computation. The computation of the k^{th}-ranked element is of particular interest in settings such as collaborative benchmarking and auctions, where the individual inputs are sensitive, yet the KRE is of mutual interest to all parties [1, 22].

Benchmarking. A key performance indicator (KPI) is a statistical quantity measuring the performance of a business process. Benchmarking is a management process where a company compares its KPI to the statistics of the same KPIs of a group of competitors from a peer group. A peer group is a group of similar companies, usually competitors, wanting to compare against each other.

Confidentiality. Confidentiality is of the utmost importance in benchmarking, since KPIs allow the inference of sensitive information. Companies are therefore

hesitant to share their business performance data due to the risk of losing a competitive advantage [22]. The confidentiality issue can be addressed using SMC [3, 16, 32], which guarantees that no party will learn more than the output of the protocol, i.e., the other parties' inputs remain confidential.

Communication Model. Generic SMC protocols [1, 3, 16, 32] can be used to keep KPIs confidential. They require a communication channel between each pair of parties. We will refer to this approach as the *standard model*. Protocols in this model do not scale easily to a large number of parties as they are highly interactive, resulting in high latency. Moreover, they are difficult to deploy as special arrangements are required between each pair of parties to establish a secure connection [10]. A promising approach for overcoming these limitations is to use the help of a small set of untrusted non-colluding servers. We will therefore refer to it as the *server model*. Relying on multiple non-colluding servers requires a different business model for providers of a privacy-preserving service [23]. We therefore use a communication model consisting of clients (with private inputs) and a server. In this model, the server provides no input and does not learn the output, but makes its computational resources available to the clients [20, 22]. There are communication channels only between each client and the server resulting in a centralized communication pattern, i.e., a star network. This model naturally fits with the client-server architecture of Internet applications and allows a service provider to play the server's role. It simplifies the secure protocol, and improves its performance and scalability [10, 20].

Symbol	Interpretation
μ	Bitlength of inputs
n	Number of clients
t	Secret sharing threshold, $t \leq n$
κ	Bitlength of asymmetric ciphertext
λ	Security parameter
$x_1, \ldots, x_n$	Client's inputs
$x_i^b = x_{i\mu} \ldots x_{i1}$	Bit representation of x_i with MSB $x_{i\mu}$
$\lvert y \rvert$	Bitlength of integer y, e.g., $\lvert x_i \rvert = \mu$
$[\![x_i]\!]$	x_i's ciphertext under public key pk
$[\![x_i]\!]_j$	x_i's ciphertext under public key pk_j
$[\![x_i^b]\!]$	Bitwise encryption $([\![x_{i\mu}]\!], \ldots, [\![x_{i1}]\!])$
$i \xleftarrow{\$} \mathbb{S}$	Choose a random element i in set $\mathbb{S}$
$\{i_1, \ldots, i_t\} \xleftarrow{\$} \mathbb{S}$	Choose t random distinct elements in $\mathbb{S}$
$\mathfrak{S}_n$	Set of all permutations of $\{1, \ldots, n\}$

(a) Notations.

	#Rounds	CR	FT
Kre-Ygc	4	$n-1 \mid 0$	0
Kre-Ahe	4	$t-1 \mid t$	$n-t$
Kre-She	2	$t-1 \mid t$	$n-t$
[1]	$O(\mu)$	$t-1 \mid$ n/a	$n-t$

(b) Schemes' properties: Column "#Rounds" is the number of rounds. Column "CR" refers to the number of parties that can collude - server excluded | server included - without breaking the privacy of non-colluding clients. Column "FT" refers to the number of parties that can fail without preventing the protocol to properly compute the intended functionality.

Table 1: Notations and Schemes' properties

Contribution. In summary, we propose different approaches for securely computing the k^{th}-ranked element (KRE) in a star network using garbled circuits (GC) or additive homomorphic encryption (AHE) or somewhat homomorphic encryption (SHE). Our schemes are secure against a semi-honest adversary:

– Our first scheme Kre-Ygc uses Yao's GC [2, 28] to compare clients' inputs.

	Kre-Ygc		Kre-Ahe	Kre-She	[1]
	sym.	asym.			
CC-C	$O(n\mu)$	$O(n)$	$O(n\mu)$	$O(\mu)$	$O(n\mu^2)$
CC-S	$O(n^2\mu)$	$O(n \log n)$	$O(n^2\mu)$	$O(n^2\mu \log \mu)$	n/a
BC-C	$O(n\mu\lambda)$	$O(n\kappa)$	$O(n\mu\kappa)$	$O((\mu+n)\kappa)$	$O(n\mu^2\lambda)$
BC-S	0	$O(n^2\kappa)$	$O(n^2\mu\kappa)$	$O(n\kappa)$	n/a

Table 2: Schemes' Complexity: Rows CC-C/S and BC-C/S denote the computation and communication (bit) complexity for each client and the server, respectively. The columns "sym." and "asym." denote symmetric and asymmetric operations in Kre-Ygc.

- Our second scheme Kre-Ahe is based on threshold AHE. We perform the comparison using the DGK protocol [12]. We also propose a modified variant of the Lin-Tzeng comparison protocol [27] that can be used instead of DGK, and that is faster at the cost of a small increase of the communication cost.
- The third scheme Kre-She is based on SHE and allows the server to non-interactively compute the KRE such that the clients only interact to jointly decrypt the result.

We compare the approaches in Tables 1b and 2 using the following measures:

- Rounds: In contrast to [1], our schemes have a constant number of rounds.
- Collusion-resistance: measures the number of parties that can collude without violating the privacy of the non-colluding ones.
- Fault-tolerance: measures the number of parties that can fail without preventing the protocol to properly compute the intended functionality.
- Complexity: This refers to the computation and communication complexity. A summary is illustrated in Table 2. We provide a detailed analysis in Appendix E.

Structure. The remainder of the paper is structured as follows. We begin by presenting related work in Section 2 and some preliminaries in Section 3. We present our security model in Section 4 and a technical overview in Section 5. The different approaches are presented in Sections 6 to 8. We discuss some implementation details and evaluation results in Section 9, before concluding our work in Section 10. We provide further details in the appendix.

2 Related Work

Our work is related to SMC. There are generic SMC protocols [13, 21] that can be used to compute the k^{th}-ranked element (KRE) of the union of n private datasets. Aggarwal et al. [1] introduced the first specialized protocol for the KRE. Their multiparty protocol performs a binary search in the input domain resulting in $O(\mu)$ comparisons and, hence, requiring $O(\mu)$ sequential rounds. Each round requires an SMC that performs two summations with complexity $O(n\mu)$ and two comparisons with complexity $O(\mu)$. As a result each client performs $O(n\mu^2)$ operations and sends $O(n\mu^2)$ bits. Our protocols perform $O(n^2)$ comparisons, that can be executed in parallel, and have either 4 or 2 rounds. We stress that all our $O(n^2)$ comparisons can be executed in parallel while the $O(\mu)$ comparisons of Aggarwal et al. must be executed sequentially one per round. In our 4-round schemes each client is involved in only $O(n)$ comparisons while each comparison in [1] involves all n clients. As a result, a client in our 4-round schemes has a complexity of $O(n)$ per round but must execute only 4 rounds, while a client in

[1] has a complexity of $O(n)$ per round as well but must execute up to μ rounds. In the 2-round scheme, all $O(n^2)$ comparisons are performed non-interactively by the server which in our model is allowed to be computationally more powerful. Our communication model allows to reduce the number of rounds from μ to 4 or 2. We note that $\mu = 32$ in our experiments for the 4-rounds schemes and $\mu = 16$ for the 2-round scheme. A summary of the complexity of our schemes is illustrated in Table 2.

The server model for SMC was introduced in [15], used in Kerschbaum [23], and cryptographic studied in [20]. The computation of the k^{th}-ranked element is also addressed in [4, 5] where the server is replaced by a blockchain.

3 Preliminaries

Garbled Circuit (GC). A GC [2, 14, 28, 33] can be used for secure 2-party computation. To evaluate a function f on input x_i, x_j, a garbling scheme $(F, e) \leftarrow Gb(1^\lambda, s, f)$ takes a security parameter λ, a random seed s, a Boolean encoding of f and outputs a GC F and an encoding string e that is used to derive garbled inputs $\bar{x}_i, \bar{x}_j$ from x_i, x_j, i.e. there is a function En such that $\bar{x}_i \leftarrow En(e, x_i)$ and $\bar{x}_j \leftarrow En(e, x_j)$. The garbling scheme is correct if $F(\bar{x}_i, \bar{x}_j) = f(x_i, x_j)$.

Homomorphic Encryption (HE). A HE consists of the usual algorithms for key generation $(pk, sk) \leftarrow KeyGen(\lambda)$, encryption $Enc(pk, m)$ (we denote $Enc(pk, m)$ by $[\![m]\!]$), decryption $Dec(sk, c)$. HE has an additional evaluation algorithm $Eval(pk, f, c_1, \ldots, c_n)$ that takes pk, an n-ary function f and ciphertexts $c_1, \ldots c_n$. It outputs a ciphertext c such that if $c_i = [\![m_i]\!]$ then it holds:

$$Dec(sk, Eval(ek, f, [\![m_1]\!], \ldots, [\![m_n]\!])) = Dec(sk, [\![f(m_1, \ldots, m_n)]\!]).$$

We require HE to be IND-CPA secure. If the scheme supports only addition, then it is *additively homomorphic*. Schemes such as [24, 30] are additively homomorphic and have the following properties:

- Add/Multiply: $\forall m_1, m_2, [\![m_1]\!] \cdot [\![m_2]\!] = [\![m_1 + m_2]\!]$, and $[\![m_1]\!]^{m_2} = [\![m_1 \cdot m_2]\!]$,
- Xor: $\forall a, b \in \{0,1\}, \textsc{Xor}([\![a]\!], b) = [\![a \oplus b]\!] = [\![1]\!]^b \cdot [\![a]\!]^{(-1)^b}$.

Threshold Homomorphic Encryption (THE) A THE [6] allows to share the private key to the parties using a threshold secret sharing scheme such that a subset of parties is required for decryption. Hence, instead of sk as above, the key generation outputs a set of shares $\mathbb{SK} = \{sks_1, \ldots, sks_n\}$ which are distributed to the clients. The decryption algorithm is replaced by the following algorithms:

- $\tilde{m}_i \leftarrow Decp(sks_i, c)$: The probabilistic partial decryption algorithm takes a ciphertext c and a share $sks_i \in \mathbb{SK}$ of the private key and outputs $\tilde{m}_i$.
- $m' \leftarrow Decf(\mathbb{M}_t)$: The deterministic final decryption algorithm takes a subset $\mathbb{M}_t = \{\tilde{m}_{j_1}, \ldots, \tilde{m}_{j_t}\} \subseteq \{\tilde{m}_1, \ldots, \tilde{m}_n\}$ of shares and outputs a message m'.

We refer to it as *threshold decryption*. It is correct if for all $\mathbb{M}_t = \{\tilde{m}_{j_1}, \ldots, \tilde{m}_{j_t}\}$ such that $|\mathbb{M}_t| \geq t$ and $\tilde{m}_{j_i} = Decp(sks_{j_i}, [\![m]\!])$, it holds $m = Decf(\mathbb{M}_t)$.

When used in a protocol, we denote by *combiner* the party which executes $Decf()$. It receives a set $\mathbb{M}_t = \{\tilde{m}_{j_1}, \ldots, \tilde{m}_{j_t}\}$ of partial decryption, runs $m' \leftarrow Decf(\mathbb{M}_t)$ and moves to the next step of the protocol specification.

4 Security Definition

This section provides definitions related to our model and security requirements. We start by defining the k^{th}-ranked element of a sequence of integers.

Definition 1. *Let $\mathbb{X} = \{x_1, ..., x_n\}$ be a set of n distinct integers and $\tilde{x}_1, \ldots, \tilde{x}_n$ be the corresponding sorted set, i.e., $\tilde{x}_1 < \ldots < \tilde{x}_n$, and $\mathbb{X} = \{\tilde{x}_1, \ldots, \tilde{x}_n\}$. The* rank *of an element $x_i \in \mathbb{X}$ is j, such that $x_i = \tilde{x}_j$. The k^{th}-ranked element* (KRE) *is the element $\tilde{x}_k$ with rank k.*

If the rank is $k = \left\lceil \frac{n}{2} \right\rceil$ then the element is called *median*. If $k = 1$ (resp. $k = n$) then the element is called *minimum* (resp. *maximum*).

Definition 2. *Let $C_1, \ldots, C_n$ be n clients each holding a private μ-bit integer $x_1, \ldots, x_n$ and S be a server which has no input. Our* ideal functionality $\mathcal{F}_{\text{KRE}}$ *receives $x_1, \ldots, x_n$ from the clients, computes the* KRE $\tilde{x}_k$ *and outputs $\tilde{x}_k$ to each client C_i. Moreover, $\mathcal{F}_{\text{KRE}}$ outputs a* leakage $\mathcal{L}_i$ *to each C_i and $\mathcal{L}_S$ to S.*

The leakage is specific to each protocol and contains information such as n, t, λ, κ, μ (see Table 1a). It can be inferred from the party's view. In case of collusion, additional leakage might include comparison results or the rank of some inputs.

Definition 3. *The* view *of the i-th party during an execution of the protocol on input $\vec{x} = (x_1, \ldots, x_n)$ is denoted by: $\mathsf{View}_i(\vec{x}) = \{x_i, r_i, m_{i1}, m_{i2}, \ldots\}$, where r_i represents the outcome of the i-th party's internal coin tosses, and m_{ij} represents the j-th message it has received.*

Since the server has no input, x_i in its view will be replaced by the empty string. We say that two distributions $\mathcal{D}_1$ and $\mathcal{D}_2$ are computationally indistinguishable (denoted $\mathcal{D}_1 \overset{c}{\equiv} \mathcal{D}_1$) if no probabilistic polynomial time (PPT) algorithm can distinguish them except with negligible probability. In this paper, we assume a semi-honest adversary. That is, parties follow the protocol, but the adversary tries to infer as much information as possible.

Definition 4. *Let $\mathcal{F}_{\text{KRE}} : (\{0,1\}^\mu)^n \mapsto \{0,1\}^\mu$ be the functionality that takes n μ-bit inputs $x_1, \ldots, x_n$ and returns their KRE. Let $I = \{i_1, \ldots, i_t\} \subset \{1, \ldots, n+1\}$ be a subset of indexes of corrupted parties (Server's input x_{n+1} is empty), $\vec{x} = (x_1, \ldots, x_n)$ and $\mathsf{View}_I(\vec{x}) = (I, \mathsf{View}_{i_1}(\vec{x}), \ldots, \mathsf{View}_{i_t}(\vec{x}))$. A protocol t-*privately computes $\mathcal{F}_{\text{KRE}}$ in the semi-honest model *if there exists a PPT simulator* SIM *such that: $\forall I, |I| = t$, $\mathcal{L}_I = \bigcup_{i \in I} \mathcal{L}_i$: $\text{SIM}(I, (x_{i_1}, \ldots, x_{i_t}), \mathcal{F}_{\text{KRE}}(\vec{x}), \mathcal{L}_I) \overset{c}{\equiv} \mathsf{View}_I(\vec{x})$.*

5 Technical Overview

In an initialization phase, clients generate and exchange cryptographic keys through the server, i.e., using the help of a non-colluding trusted third party. We stress that the initialization is run once and its complexity does not depend on the functionality $\mathcal{F}_{\text{KRE}}$. We therefore focus on the actual computations.

Definition 5. *Let $x_i, x_j, 1 \leq i, j, \leq n$, be integer inputs of C_i, C_j. Then the* comparison bit *b_{ij} of the pair (x_i, x_j) is defined as 1 if $x_i \geq x_j$ and 0 otherwise. The computation of $x_i \geq x_j$ is distributed and involves C_i, C_j, where they play different roles, e.g., generator and evaluator. Similar to the functional programming notation of an ordered pair, we use* head *and* tail *to denote C_i and C_j.*

Lemma 1. *Let $x_1, \ldots, x_n$ be n distinct integers, $r_1, \ldots, r_n \in \{1, \ldots, n\}$ their respective ranks and b_{ij} the comparison bit for (x_i, x_j). It holds $r_i = \sum_{j=1}^{n} b_{ij}$.*

To make sure that inputs are indeed distinct before the protocol, one can use the indexes of each C_i as differentiator [1]. Each C_i appends the $\log n$-bit string of i at the end of the bit string of x_i, resulting in a new input of length $\mu + \log n$. Note that, the extended input will be used only for input comparison, to avoid leaking the index of the winning client. For simplicity, we assume in the remainder of the paper, that the x_i's are all distinct μ-bit integers. Therefore, it is not necessary to compare all pairs (x_i, x_j), since $b_{ji} = 1 - b_{ij}$.

We would like to equally distribute the computation tasks among the clients. As example for $n = 3$, we need to compute only 3 (instead of 9) comparisons resulting in three *head* roles and three *tail* roles. Then we would like each of the three clients to play the role *head* as well as *tail* exactly once. We use Definition 6 and Lemma 2 to equally distribute the roles *head* and *tail* between clients.

Definition 6. *Let $\mathbb{X} = \{x_1, \ldots, x_n\}$ be a set of n integers. We define the predicate* Paired *as follows*

$$
\begin{aligned}
\textsc{Paired}(i,j) \coloneqq\ & (i \equiv 1 \pmod 2 \wedge i > j \wedge j \equiv 1 \pmod 2) \vee && \text{(1a)}\\
& (i \equiv 1 \pmod 2 \wedge i < j \wedge j \equiv 0 \pmod 2) \vee && \text{(1b)}\\
& (i \equiv 0 \pmod 2 \wedge i > j \wedge j \equiv 0 \pmod 2) \vee && \text{(1c)}\\
& (i \equiv 0 \pmod 2 \wedge i < j \wedge j \equiv 1 \pmod 2). && \text{(1d)}
\end{aligned}
$$

Lemma 2. *Let $\mathbb{X} = \{x_1, \ldots, x_n\}$ be a set of n integers and the predicate* Paired *be as above. Then comparing only pairs (x_i, x_j) such that* $\textsc{Paired}(i,j) = \mathit{true}$ *is enough to compute the rank of all elements in $\mathbb{X}$.*

For example, if $n = 3$, we compute b_{ij} only for (x_1, x_2), (x_2, x_3), (x_3, x_1). If $n = 4$, we compare only (x_1, x_2), (x_1, x_4), (x_2, x_3), (x_3, x_1), (x_3, x_4), (x_4, x_2).

The predicate Paired (Equation 1) is used in our schemes to reduce the number of comparisons and to equally distribute the computation task of the comparisons among the clients. As pointed out by an anonymous reviewer, Paired can be simplified as: $(i > j \wedge i \equiv j \pmod 2) \vee (i < j \wedge i \not\equiv j \pmod 2)$.

Let $\#head_i$ (resp. $\#tail_i$) denote the number of times $\textsc{Paired}(i,j) = \mathit{true}$ (resp. $\textsc{Paired}(j,i) = \mathit{true}$) holds. For example, if $n = 3$, we have $\#head_i = \#tail_i = 1$ for all clients. However, for $n = 4$, we have $\#head_1 = \#head_3 = 2$, $\#tail_1 = \#tail_3 = 1$, $\#head_2 = \#head_4 = 1$ and $\#tail_2 = \#tail_4 = 2$.

Lemma 3. *Let* $\mathbb{X} = \{x_1, \ldots, x_n\} \subset \mathbb{N}$ *and assume the predicate* PAIRED *is used to sort* $\mathbb{X}$. *If* n *is odd then:* $\#head_i = \#tail_i = \frac{n-1}{2}$. *If* n *is even then:*

$$\#head_i = \begin{cases} \frac{n}{2} & \text{if } i \text{ odd} \\ \frac{n}{2} - 1 & \text{if } i \text{ even} \end{cases} \quad \#tail_i = \begin{cases} \frac{n}{2} - 1 & \text{if } i \text{ odd} \\ \frac{n}{2} & \text{if } i \text{ even.} \end{cases}$$

6 Protocol KRE-YGC

KRE-YGC is based on GC and consists of an initialization and a main part. It does not tolerate collision with the server. An AHE ciphertext is denoted by $[\![\cdot]\!]$.

6.1 KRE-YGC Initialization

The initialization consists of public key distribution and Diffie-Hellman (DH) key agreement. Each client C_i sends its public key pk_i of an AHE to the server S. Then S distributes the pk_i to all C_i. In our implementation, we use Paillier's scheme [30] , but any AHE scheme such as [24] will work. Then each pair (C_i, C_j) of clients runs DH key exchange through the server to generate a common secret key $ck_{ij} = ck_{ji}$. The key ck_{ij} is used by C_i and C_j to seed the pseudorandom number generator of the garbling scheme that is used to generate a comparison GC for x_i and x_j, i.e. $Gb(1^\lambda, ck_{ij}, f_>)$, where $f_>$ is a Boolean comparison circuit. For comparison GC, we will use the schemes of Kolesnikov et al. [25, 26].

6.2 KRE-YGC Main Protocol

Protocol 1 is a 4-round protocol in which we use GC to compare inputs and to reveal a blinded comparison bit to the server. Then we use AHE to unblind the comparison bits, compute the ranks and the KRE without revealing anything to the parties. Let $f_>$ be defined as: $f_>((a_i, x_i), (a_j, x_j)) = a_i \oplus a_j \oplus b_{ij}$, where $a_i, a_j \in \{0,1\}$, i.e., $f_>$ computes $b_{ij} = [x_i > x_j]$ and blinds the bits b_{ij} with a_i, a_j.

Comparing Inputs. For each pair (x_i, x_j), if $\text{PAIRED}(i,j) = true$ the parties do the following:

- C_i chooses a masking bit $a_i^{ij} \xleftarrow{\$} \{0,1\}$ and extends its input to (a_i^{ij}, x_i). Then using the common key ck_{ij}, it computes $(F_>^{ij}, e) \leftarrow Gb(1^\lambda, ck_{ij}, f_>)$ and $(\bar{a}_i^{ij}, \bar{x}_i^{ij}) \leftarrow En(e, (a_i^{ij}, x_i))$, and sends $F_>^{ij}$, $(\bar{a}_i^{ij}, \bar{x}_i^{ij})$ to S.
- C_j chooses a masking bit $a_j^{ij} \xleftarrow{\$} \{0,1\}$ and extends its input x_j to (a_j^{ij}, x_j). Then using the common key $ck_{ji} = ck_{ij}$, it computes $(F_>^{ij}, e) \leftarrow Gb(1^\lambda, ck_{ji}, f_>)$ and $(\bar{a}_j^{ij}, \bar{x}_j^{ij}) \leftarrow En(e, (a_j^{ij}, x_j))$, and sends only $(\bar{a}_j^{ij}, \bar{x}_j^{ij})$ to S.
- We have $b'_{ij} \leftarrow F_>^{ij}((\bar{a}_i^{ij}, \bar{x}_i^{ij}), (\bar{a}_j^{ij}, \bar{x}_j^{ij})) = a_i^{ij} \oplus a_j^{ij} \oplus b_{ij}$ (i.e. b_{ij} is hidden to S). The server then evaluates all GCs (Steps 1 to 5).

Unblinding Comparison Bits. Using AHE, the parties unblind each $b'_{ij} = a_i^{ij} \oplus a_j^{ij} \oplus b_{ij}$, where a_i^{ij} is known to C_i and a_j^{ij} is known to C_j, without learning anything. As a result $[\![b_{ij}]\!]_i$ and $[\![b_{ij}]\!]_j$ are revealed to S encrypted under pk_i and pk_j. This is illustrated in Steps 6 to 16 and works as follows:

– S sends b'_{ij} to C_i and C_j. They reply with $[\![a_j^{ij} \oplus b_{ij}]\!]_i$ and $[\![a_i^{ij} \oplus b_{ij}]\!]_j$.
– S forwards $[\![a_i^{ij} \oplus b_{ij}]\!]_j$, $[\![a_j^{ij} \oplus b_{ij}]\!]_i$ to C_i, C_j. They reply with $[\![b_{ij}]\!]_j$, $[\![b_{ij}]\!]_i$.
– S sets $[\![b_{ji}]\!]_j = [\![1 - b_{ij}]\!]_j$.

Computing the Rank. The computation of the rank is done at the server by homomorphically adding comparison bits. Hence for each i, the server computes $[\![r_i]\!]_i = [\![\sum_{j=1}^{n} b_{ij}]\!]_i$. Then, it chooses a random number α_i and computes $[\![\beta_i]\!]_i = [\![(r_i - k) \cdot \alpha_i]\!]_i$ (Steps 17 to 19). The ciphertext $[\![\beta_i]\!]_i$ encrypts 0 if $r_i = k$ (i.e., x_i is the k^{th}-ranked element) otherwise it encrypts a random plaintext.

Computing the KRE's Ciphertext. Each client C_i receives $[\![\beta_i]\!]_i$ encrypted under its public key pk_i and decrypts it. Then if $\beta_i = 0$, C_i sets $m_i = x_i$ otherwise $m_i = 0$. Finally, C_i encrypts m_i under each client's public key and sends $[\![m_i]\!]_1, \ldots, [\![m_i]\!]_n$ to the server (Steps 20 to 22).

Revealing the KRE's Ciphertext. In the final steps (Steps 23 to 24), the server adds all $[\![m_j]\!]_i$ encrypted under pk_i and reveals $[\![\sum_{j=1}^{n} m_j]\!]_i$ to C_i.

Kre-Ygc protocol correctly computes the KRE. The proof follows from the correctness of the GC protocol, Lemmas 1 and 2 and the correctness of the AHE scheme. Kre-Ygc is not fault-tolerant and a collusion with the server reveals all inputs to the adversary. In the next section, we address this using threshold HE. We stress that using threshold HE in Kre-Ygc is not enough as the server has all GCs and each client can decode all GCs involving its input.

Protocol 1: Kre-Ygc Protocol

1: **for** $i := 1, j := i + 1$ **to** n **do**
2: **if** Paired(i, j) **then**
3: $C_i \to S$: $F_{>}^{ij}, (\bar{a}_i^{ij}, \bar{x}_i^{ij})$
4: $C_j \to S$: $(\bar{a}_j^{ij}, \bar{x}_j^{ij})$
5: S: **let** $b'_{ij} \leftarrow F_{>}^{ij}(\bar{x}_i^{ij}, \bar{x}_j^{ij})$
6: **for** $i := 1, j := i + 1$ **to** n **do**
7: **if** Paired(i, j) **then**
8: $S \to C_i$: $b'_{ij} = a_i^{ij} \oplus a_j^{ij} \oplus b_{ij}$
9: $S \to C_j$: $b'_{ij} = a_i^{ij} \oplus a_j^{ij} \oplus b_{ij}$
10: $C_i \to S$: $[\![a_j^{ij} \oplus b_{ij}]\!]_i$
11: $C_j \to S$: $[\![a_i^{ij} \oplus b_{ij}]\!]_j$
12: $S \to C_i$: $[\![a_i^{ij} \oplus b_{ij}]\!]_j$
13: $S \to C_j$: $[\![a_j^{ij} \oplus b_{ij}]\!]_i$
14: $C_i \to S$: $[\![b_{ij}]\!]_j$
15: $C_j \to S$: $[\![b_{ij}]\!]_i$
16: S: **let** $[\![b_{ji}]\!]_j \leftarrow [\![1 - b_{ij}]\!]_j$
17: **for** $i := 1$ **to** n **do**
18: $S : [\![r_i]\!]_i \leftarrow [\![\sum_{j=1}^{n} b_{ij}]\!]_i$ ▷ $b_{ii} = 1$
19: $S \to C_i$: $[\![\beta_i]\!]_i \leftarrow [\![(r_i - k) \cdot \alpha_i]\!]_i$, for a random α_i
20: **for** $i := 1$ **to** n **do**
21: C_i: $m_i := \begin{cases} x_i & \text{if } \beta_i = 0 \\ 0 & \text{if } \beta_i \neq 0 \end{cases}$
22: $C_i \to S$: $[\![m_i]\!]_1, \ldots, [\![m_i]\!]_n$
23: **for** $i := 1$ **to** n **do**
24: $S \to C_i$: $[\![\sum_{j=1}^{n} m_j]\!]_i$

7 Protocol Kre-Ahe

In this section, we describe Kre-Ahe (Protocol 4) which instantiates the comparison with DGK [12]. We start by describing the initialization.

7.1 Kre-Ahe Initialization

We assume threshold key generation. Hence, there is a public/private key pair (pk, sk) for an AHE, where sk is split in n shares $sks_1, \ldots, sks_n$ such that

client C_i gets share sks_i and at least t shares are required to reconstruct sk. Additionally, each C_i has its own AHE key pair (pk_i, sk_i) and publishes pk_i. We denote by $[\![x_i]\!], [\![x_i]\!]_j$ encryptions of x_i under pk, pk_j respectively (Table 1a).

7.2 DGK Comparison Protocol

We briefly reviewing DGK [12]. To determine whether $x_i \leq x_j$ or $x_i > x_j$, one computes for each $1 \leq u \leq \mu$ the following numbers z_u: $z_u = s + x_{iu} - x_{ju} + 3\sum_{v=u+1}^{\mu}(x_{iv} \oplus x_{jv})$. Let (pk_i, sk_i) be the key pair of C_i. Client C_i will be called *Generator* and C_j *Evaluator*. Privately evaluating $x_i \geq x_j$ works as follows:

- C_i sends $[\![x_{i\mu}]\!]_i, \ldots, [\![x_{i1}]\!]_i$ (encrypted under pk_i) to client C_j.
- C_j chooses a random bit δ_{ji}, sets $s = 1 - 2 \cdot \delta_{ji}$, computes $[\![z_u]\!]_i$ as defined above, sends $([\![z_\mu]\!]_i, \ldots, [\![z_1]\!]_i)$ to C_i in a random order and outputs δ_{ji}.
- If one $[\![z_u]\!]_i$ decrypts to 0 then C_i sets $\delta_{ij} = 1$ else $\delta_{ij} = 0$. C_i outputs δ_{ij}.

In our server model, clients run the protocol through the server such that after the computation the server learns $[\![\delta_{ij} \oplus \delta_{ji}]\!]$ encrypted under pk. That is, C_j sends $[\![z_{i\mu}]\!]_i, \ldots, [\![z_{i1}]\!]_i, [\![\delta_{ji}]\!]$ to C_i via the server, where each $[\![z_{iu}]\!]_i$ is encrypted under pk_i and $[\![\delta_{ji}]\!]$ is encrypted under pk. Client C_i computes the shared bit δ_{ij} and sends back $[\![\delta_{ij} \oplus \delta_{ji}]\!]$ to the server. In DGK, clients C_i and C_j perform respectively $O(\mu)$ and $O(6\mu)$ asymmetric operations. We will denote a call to the DGK comparison between C_i, C_j as DgkCompare(i, j).

7.3 Modified Lin-Tzeng Comparison Protocol

We now describe our modified version of the Lin-Tzeng comparison protocol [27], which can be used instead of DGK. It is faster at the cost of sending μ more ciphertexts for each comparison. The main idea of Lin and Tzeng's scheme is to reduce the greater-than comparison to the set intersection of prefixes.

Input Encoding. Let $\text{Int}(y_\eta \cdots y_1) = y$ be a function that takes a bit string of length η and parses it into the η–bit integer $y = \sum_{l=1}^{\eta} y_l \cdot 2^{l-1}$. The *0-encoding* $V^0_{x_i}$ and *1-encoding* $V^1_{x_i}$ of an integer input x_i are the following vectors: $V^0_{x_i} = (v_{i\mu}, \cdots, v_{i1}), V^1_{x_i} = (u_{i\mu}, \cdots, u_{i1})$, such that for each $l, (1 \leq l \leq \mu)$

$$v_{il} = \begin{cases} \text{Int}(x_{i\mu}x_{i\mu-1}\cdots x_{il'}1) & \text{if } x_{il} = 0 \\ r^{(0)}_{il} & \text{if } x_{il} = 1 \end{cases} \quad u_{il} = \begin{cases} \text{Int}(x_{i\mu}x_{i\mu-1}\cdots x_{il}) & \text{if } x_{il} = 1 \\ r^{(1)}_{il} & \text{if } x_{il} = 0, \end{cases}$$

where $l' = l + 1$, $r^{(0)}_{il}$, $r^{(1)}_{il}$ are random numbers of a fixed bitlength $\nu > \mu$ (e.g. $2^\mu \leq r^{(0)}_{il}, r^{(1)}_{il} < 2^{\mu+1}$) with $LSB(r^{(0)}_{il}) = 0$ and $LSB(r^{(1)}_{il}) = 1$ (LSB is the least significant bit). If the Int function is used the compute the element at position l, then we call it a *proper encoded element* otherwise we call it a *random encoded element*. Note that a random encoded element $r^{(1)}_{il}$ at position l in the 1-encoding of x_i is chosen such that it is guaranteed to be different to a proper or random encoded element at position l in the 0-encoding of x_j, and vice versa. Hence, it is enough if $r^{(1)}_{il}$ and $r^{(0)}_{il}$ are just one or two bits longer than any possible

proper encoded element at position l. Also note that the bitstring $x_{i\mu}x_{i\mu-1}\cdots x_{il}$ is interpreted by the function INT as the bitstring $y_{\mu-l+1}\cdots y_1$ with length $\mu-l+1$ where $y_1 = x_{il}, y_2 = x_{i(l+1)}, \ldots, y_{\mu-l+1} = x_{i\mu}$. If we see $V^0_{x_i}, V^1_{x_j}$ as sets, then $x_i > x_j$ iff they have exactly one common element.

Lemma 4. *Let x_i and x_j be two integers, then $x_i > x_j$ iff $V = V^1_{x_i} - V^0_{x_j}$ has a unique position with 0.*

The Protocol. Let $[\![V^0_{x_i}]\!]_i = ([\![v_{i\mu}]\!]_i, \ldots, [\![v_{i1}]\!]_i)$, $[\![V^1_{x_i}]\!]_i = ([\![u_{i\mu}]\!]_i, \ldots, [\![u_{i1}]\!]_i)$ denote encryption of $V^0_{x_i}$ and $V^1_{x_i}$. Let $[\![V^1_{x_i} - V^0_{x_j}]\!]_i = ([\![u_{i\mu} - v_{j\mu}]\!]_i, \ldots, [\![u_{i1} - v_{j1}]\!]_i)$. Client C_i sends $[\![V^0_{x_i}]\!]_i, [\![V^1_{x_i}]\!]_i$ to C_j via the server. Client C_j randomly chooses between evaluating either $[\![V^1_{x_i} - V^0_{x_j}]\!]_i$ or $[\![V^1_{x_j} - V^0_{x_i}]\!]_i$ and sets $\delta_{ji} \leftarrow 0$ or $\delta_{ji} \leftarrow 1$ accordingly. Then it randomizes each ciphertext and sends them back to C_i in a random order. If one of these ciphertexts decrypts to 0, C_i sets $\delta_{ij} = 1$ else $\delta_{ij} = 0$. This is clearly faster than DGK at the cost of increasing the communication (μ more ciphertexts are sent to C_j). Due to place constraint, we discuss in Appendix A the difference of our modification to the original protocol [27].

7.4 KRE-AHE Main Protocol

KRE-AHE is a 4-round protocol in which inputs are compared interactively using DGK. The comparison bits are encrypted under pk and revealed to the server which then computes the ranks of the x_i's and triggers a threshold decryption.
Uploading Ciphertext. Each C_i sends $[\![x_i]\!]$ (encrypted under pk) and $[\![x^b_i]\!]_i = ([\![x_{i\mu}]\!]_i, \ldots, [\![x_{i1}]\!]_i)$ (encrypted under its own public key pk_i) to the server. This is illustrated in Step 2 of protocol 4. The server then initializes a matrix $G = [g_{11}, \ldots, g_{nn}]$, where $g_{ii} = [\![1]\!]$ and $g_{ij} (i \neq j)$ will be computed using DGK as $g_{ij} = [\![b_{ij}]\!]$ if PAIRED(i, j) is true, and an array $X = [[\![x_1]\!], \ldots, [\![x_n]\!]]$ (Step 3).
Comparing Inputs. In this step, pairs of clients run DGKCOMPARE with the server as explained above. If (i, j) satisfies the predicate PAIRED, then C_i runs DGK as generator and C_j is the evaluator. After the computation, C_i and C_j get shares δ_{ij} and δ_{ji} of the comparison bit and the server gets $[\![b_{ij}]\!] = [\![\delta_{ij} \oplus \delta_{ji}]\!]$ which is encrypted under pk (the server cannot decrypt $[\![b_{ij}]\!]$).
Computing the KRE's Ciphertext. After all admissible comparisons have been computed (and the result stored in the matrix G), the server uses Algorithm 2 to compute the rank of each input x_i, i.e., $[\![r_i]\!] = [\![\sum_{j=1}^{n} b_{ij}]\!]$. Now, the server has the encrypted ranks $[\![r_1]\!], \ldots, [\![r_n]\!]$, where exactly one $[\![r_i]\!]$ encrypts k. Since we are looking for the element whose rank is k, the server then computes $y_i = ([\![r_i]\!] \cdot [\![k]\!]^{-1})^{\alpha_i} \cdot [\![x_i]\!] = [\![(r_i - k)\alpha_i + x_i]\!]$ for all i, where α_i is a number chosen randomly in the plaintext space. Therefore, for the ciphertext $[\![r_i]\!]$ encrypting k, y_i is equal to $[\![x_i]\!]$. Otherwise y_i encrypts a random plaintext.
Decrypting the KRE's Ciphertext. In Step 12, the server distributes the result $Y = [y_1, \ldots, y_n]$ of Algorithm 2 to the clients for threshold decryption. For that, the array Y is passed as $n \times 1$ matrix to Algorithm 3.

Lemma 5 shows that the ciphertexts generated from Algorithm 3 allow to correctly decrypt $Y = [y_1, \ldots, y_n]$. The first part shows that each C_i receives a

Algorithm 2: Computing the KRE's ciphertext in Kre-Ahe

```
1: function ComputeKreAhe(G, X, k)
2:   parse G as [g_11, ..., g_nn]
3:   parse X as [[x_1], ..., [x_n]]
4:   for i := 1 to n do
5:     [r_i] ← g_ii
6:     for j := 1 to n (j ≠ i) do
7:       if Paired(i, j) then
8:         [r_i] ← [r_i] · g_ij
9:       else
10:        [r_i] ← [r_i] · [1] · g_ji^-1
11:  for i := 1 to n do
12:    y_i ← ([r_i] · [k]^-1)^α_i · [x_i]
13:  return [y_1, ..., y_n]
```

Algorithm 3: Decryption Request in Kre-Ahe

```
1: function DecReq(Y, i, t, π)
2:   parse Y as [y_1, ..., y_n]
3:   let Z^(i) = [z_1^(i), ..., z_t^(i)]
4:   for j := 1 to t do
5:     u ← i − t + j mod n
6:     if u ≤ 0 then
7:       u ← u + n                    ▷ 1 ≤ u ≤ n
8:     I^(i) = I^(i) ∪ {u}
9:     v ← π(u)
10:    z_j^(i) = y'_v
11:  return (Z^(i), I^(i))
```

subset of t elements of Y. The second part shows that each y_i is distributed to exactly t different C_i which allows a correct threshold decryption of each row.

Lemma 5. *Let $\mathbb{X} = \{x_1, \ldots, x_n\}$ be a set of n elements, $\mathbb{X}_i = \{x_{i-t+1}, \ldots, x_i\}$, $1 \le i \le n$, where the indexes in $\mathbb{X}_i$ are computed modulo n, and $t \le n$. Then:*
- *Each subset $\mathbb{X}_i$ contains exactly t elements of $\mathbb{X}$ and*
- *Each $x \in \mathbb{X}$ is in exactly t subsets $\mathbb{X}_i$.*

In Step 16, the server S receives partial decryptions from the clients, forwards them to the corresponding combiner (Step 18). Each combiner C_j performs a final decryption (Step 21) resulting in a message $\tilde{x}_j$ whose bitlength is less or equal to μ if it is the KRE. Combiner C_j then sets $m^{(j)} = \tilde{x}_j$ if $|\tilde{x}_j| \le \mu$, otherwise $m^{(j)} = 0$ (Step 22). Then $m^{(j)}$ is encrypted with the public key of all clients and send to S (Step 23). Finally, the server reveals the KRE to each C_i (Step 25).

Kre-Ahe correctly computes the KRE. This follows from the correctness of DGK [12], Lemmas 1 and 5 and the correctness of AHE. Kre-Ahe evaluates comparisons interactively but requires threshold decryption for $O(n)$ elements. In Kre-Ahe, we can evaluate either the comparison or the rank at the server, but not both. In the next scheme, we compute the KRE's ciphertext non-interactively at the server. Clients are only required for the threshold decryption.

8 Protocol Kre-She

This section describes Kre-She based on SHE. Hence, $[\![x]\!]$ now represents an SHE ciphertext of the plaintext x. The initialization and threshold decryption are similar to Kre-Ahe.

8.1 SHE Routines

Protocol Kre-She is based on the BGV scheme [9] as implemented in HElib [17] and requires binary plaintext space and Smart-Vercauteren ciphertext packing

Protocol 4: KRE-AHE Protocol

1: **for** $i \coloneqq 1$ **to** n **do**
2: $C_i \to S$: $[\![x_i]\!], [\![x_i^b]\!]_i$
3: S : **let** $G = [g_{11}, \ldots, g_{nn}]$
S : **let** $X = [[\![x_1]\!], \ldots, [\![x_n]\!]]$
4: **for** $i \coloneqq 1, j \coloneqq i+1$ **to** n **do**
5: **if** PAIRED(i,j) **then**
6: C_i, C_j, S: $g_{ij} \leftarrow$ DGKCOMPARE(i,j)
7: S: $Y \leftarrow$ COMPUTEKREAHE(G, X, k)
8: S: **let** $\pi \xleftarrow{\$} \mathfrak{S}_n$ be a permutation
9: S: **parse** Y **as** $[y_1, \ldots, y_n]$
10: S: **let** $Y' = [y_{\pi(1)}, \ldots, y_{\pi(n)}]$
11: **for** $i \coloneqq 1$ **to** n **do**
12: $S \to C_i$: $Q^{(i)} \leftarrow$ DECREQ(Y', i, t, π)
13: C_i: **parse** $Q^{(i)}$ as $(Z^{(i)}, I^{(i)})$
parse $I^{(i)}$ as $[j_1, \ldots, j_t]$
parse $Z^{(i)}$ as $[z_{j_1}^{(i)}, \ldots, z_{j_t}^{(i)}]$
14: **for** $i \coloneqq 1$ **to** n **do**
15: **for each** j **in** $I^{(i)}$ **do**
16: $C_i \to S$: $h_j^{(i)} \leftarrow [\![Decp(sks_i, z_j^{(i)})]\!]_j$
17: **for** $j \coloneqq 1$ **to** n **do**
18: $S \to C_j$: $(h_j^{(i_1)}, \ldots, h_j^{(i_t)})$
19: **for** $j \coloneqq 1$ **to** n **do**
20: C_j: $d_u = Dec(sk_j, h_j^{(i_u)}), u = 1, \ldots, t$
21: C_j: $\tilde{x}_j \leftarrow Decf(d_1, \ldots, d_t)$
22: C_j: $m^{(j)} \coloneqq \begin{cases} \tilde{x}_j & \text{if } |\tilde{x}_{j'}| \le \mu \\ 0 & \text{if } |\tilde{x}_{j'}| > \mu \end{cases}$
23: $C_j \to S$: $[\![m^{(j)}]\!]_1, \ldots, [\![m^{(j)}]\!]_n$
24: **for** $i \coloneqq 1$ **to** n **do**
25: $S \to C_i$: $[\![\sum_{j=1}^n m^{(j)}]\!]_i$
26: C_i: $Dec(sk_i, [\![\sum_{j=1}^n m^{(j)}]\!]_i)$

(SVCP) technique [31]. Using SVCP, a ciphertext consists of a fixed number m of slots encrypting bits, i.e. $[\![\cdot|\cdot|\ldots|\cdot]\!]$. The encryption of a bit b replicates b to all slots, i.e., $[\![b]\!] = [\![b|b|\ldots|b]\!]$. However, one can pack the bits of x_i^b in one ciphertext and will denote it by $[\![\vec{x}_i]\!] = [\![x_{i\mu}|\ldots|x_{i1}|0|\ldots|0]\!]$.

Each C_i sends $[\![x_i^b]\!], [\![\vec{x}_i]\!]$ to S as input to Algorithm 5 which uses built-in routines to compute the KRE. We denote addition and multiplication routines by the operators $\oplus$ and $\odot$. Then addition of packed ciphertexts is defined as component-wise addition mod 2: $[\![b_{i1}|\ldots|b_{im}]\!] \oplus [\![b_{j1}|\ldots|b_{jm}]\!] = [\![b_{i1} \oplus b_{j1}|\ldots|b_{im} \oplus b_{jm}]\!]$. The multiplication is defined similarly: $[\![b_{i1}|\ldots|b_{im}]\!] \odot [\![b_{j1}|\ldots|b_{jm}]\!] = [\![b_{i1} \odot b_{j1}|\ldots|b_{im} \odot b_{jm}]\!]$, where $b_{iu} \odot b_{ju}$ is a multiplication mod 2.

Let x_i, x_j be two integers, $b_{ij} = [x_i > x_j]$ and $b_{ji} = [x_j > x_i]$, the routine SHECMP takes $[\![x_i^b]\!], [\![x_j^b]\!]$, compares x_i and x_j and returns $[\![b_{ij}]\!], [\![b_{ji}]\!]$. Note that if the inputs to SHECMP encrypt the same value, then the routine outputs two ciphertexts of 0. The comparison circuit has depth $\log(\mu - 1) + 1$ and requires $O(\mu \log \mu)$ homomorphic multiplications [11].

Let $b_{i1}, \ldots, b_{in}$ be n bits such that $r_i = \sum_{j=1}^n b_{ij}$ and let $r_i^b = r_{i \log n}, \ldots, r_{i1}$ be the bit representation of r_i. The routine SHEFADDER implements a full adder on $[\![b_{i1}]\!], \ldots, [\![b_{in}]\!]$ and returns $[\![r_i^b]\!] = ([\![r_{i \log n}]\!], \ldots, [\![r_{i1}]\!])$.

There is no built-in routine for equality check in HElib. We implemented it using SHECMP and addition. Let x_i and x_j be two μ-bit integers. We use SHEEQUAL to denote the equality check routine and implement SHEEQUAL$([\![x_i^{\mathsf{b}}]\!], [\![x_j^{\mathsf{b}}]\!])$ by first computing $([\![b_i']\!], [\![b_i'']\!]) =$ SHECMP$([\![x_i^{\mathsf{b}}]\!], [\![x_j^{\mathsf{b}}]\!])$ and then $[\![\beta_i]\!] = [\![b_i']\!] \oplus [\![b_i'']\!] \oplus [\![1]\!]$. This results in $\beta_i = 1$ if $x_i = x_j$ and $\beta_i = 0$ otherwise.

8.2 KRE-SHE Main Protocol

In KRE-SHE, the server S receives encrypted inputs from clients. For each client's integer x_i, the encrypted input consists of:

– an encryption $[\![x_i^{\mathsf{b}}]\!] = ([\![x_{i\mu}]\!], \ldots, [\![x_{i1}]\!])$ of the bit representation and
– an encryption $[\![\vec{x}_i]\!] = [\![x_{i\mu}|\ldots|x_{i1}|0|\ldots|0]\!]$ of the packed bit representation.

Then the server runs Algorithm 5 which uses SheCmp to pairwise compare the inputs resulting in encrypted comparison bits $[\![b_{ij}]\!]$. Then SheFadder is used to compute the rank of each input by adding comparison bits. The result is an encrypted bit representation $[\![r_i^{\mathsf{b}}]\!]$ of the ranks. Using the encrypted bit representations $[\![k^{\mathsf{b}}]\!]$, $[\![r_i^{\mathsf{b}}]\!]$ of k and each rank, SheEqual checks the equality and returns an encrypted bit $[\![\beta_i]\!]$. Recall that because of SVCP the encryption of a bit β_i is automatically replicated in all slots, i.e., $[\![\beta_i]\!] = [\![\beta_i|\beta_i|\ldots|\beta_i]\!]$, such that evaluating $[\![\vec{y}_i]\!] \leftarrow [\![\vec{x}_i]\!] \odot [\![\beta_i]\!]$, $1 \le i \le n$, and $[\![\vec{y}_1]\!] \oplus \ldots \oplus [\![\vec{y}_n]\!]$ returns the KRE's ciphertext. Correctness and security follow from Lemma 1, correctness and security of SHE. The leakage is $\mathcal{L}_S = \mathcal{L}_i = \{n, t, \kappa, \lambda, \mu\}$.

Algorithm 5: Computing the KRE's Ciphertext in Kre-She

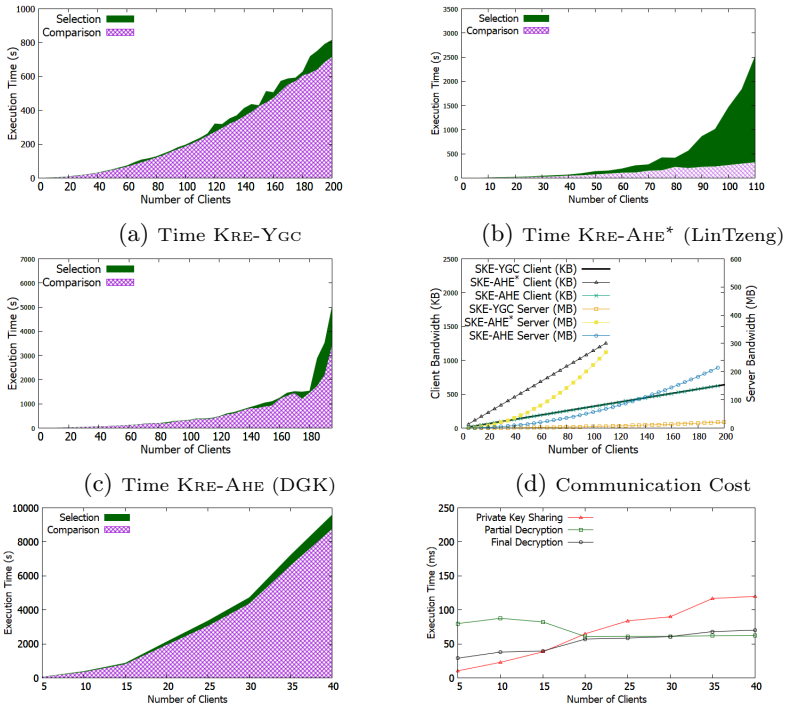
```
1: function ComputeKreShe(X, Z, c)
2:   parse X as [[x_1^b], ..., [x_n^b]]
     parse Z as [[x⃗_1], ..., [x⃗_n]]
     parse c as [k^b]
3:   for i := 1 to n do
4:     [b_ii] ← [1]
5:     for j := i + 1 to n do
6:       ([b_ij], [b_ji]) ← SheCmp([x_i^b], [x_j^b])
7:   for i := 1 to n do
8:     [r_i^b] ← SheFadder([b_i1], ..., [b_in])
9:   for i := 1 to n do
10:    [β_i] ← SheEqual([r_i^b], [k^b])
11:  for i := 1 to n do
12:    [y⃗_i] ← [x⃗_i] ⊙ [β_i]
13:  return [y⃗_1] ⊕ ... ⊕ [y⃗_n]
```

9 Evaluation

This section presents our evaluation results. We implemented Kre-Ygc and Kre-Ahe as client-server Java applications while using SCAPI [14]. As Kre-She mostly consists of the homomorphic evaluation by the server, we implemented Algorithm 5 and n-out-of-n threshold decryption using HElib [17, 18].

Experimental Setup. For Kre-Ygc and Kre-Ahe, we experimented using for the server a machine with a 6-core Intel(R) Xeon(R) E-2176M CPU @ 2.70GHz and 32GB of RAM, and for the clients two machines with each two Intel(R) Xeon(R) CPU E7-4880 v2 @ 2.50GHz. The client machines were equipped with 8GB and 4GB of RAM, and were connected to the server via WAN. Windows 10 Enterprise was installed on all three machines. For each experiment, about 3/5 of the clients were run on the machine with 8GB RAM while about 2/5 were run on the machine with 4GB RAM. Since the main computation of Kre-She is done on the server, we evaluate only Algorithm 5 on a Laptop with Intel(R) Core(TM) i5-7300U CPU @ 2.60GHz running 16.04.1-Ubuntu.

Results. We evaluated Kre-Ygc, Kre-Ahe at security level $\lambda = 128$, bitlength $\mu = 32$ and (minimal) threshold $t = 2$ for threshold decryption. We instantiated Kre-Ahe with Elliptic Curve ElGamal using curve `secp256r1`. We implemented ElGamal using CRT-based technique of Hu et al. [19] and pre-computation of the



(a) Time KRE-YGC (b) Time KRE-AHE* (LinTzeng)

(c) Time KRE-AHE (DGK) (d) Communication Cost

(e) Server Time KRE-SHE (f) Threshold Decryption KRE-SHE

Fig. 6: Results for KRE-YGC, KRE-AHE, KRE-SHE

logarithm table [7] for fast threshold decryption [8]. Figure 8 shows our results which are summarized in Table 3 for $n = 100$.

KRE-YGC is the most efficient in both computation and communication and takes 197 seconds to each client to compute the KRE of 100 clients in a WAN setting. The communication is 0.31 MB for each client and 5.42 MB for the server. However, KRE-YGC is neither collusion-resistant nor fault-tolerant.

KRE-AHE is the second most efficient and is collusion-resistant and fault-tolerant. In KRE-AHE, the comparisons can be evaluated non-interactively using LinTzeng or interactively using both LinTzeng and DGK. The non-interactive variant (denoted by KRE-AHE*) requires $O(n^2)$ threshold decryptions. It computation cost is illustrated in Figure 6b. The interactive one, whose cost is illustrated in Figure 6c, requires only $O(n)$ threshold decryptions. In Table 3, we also illustrate the costs when $t = 1$ (i.e., each C_i knows sk) for both KRE-AHE and KRE-AHE* and when $t = n$ (i.e., all C_i run the decryption) for KRE-AHE.

Table 3: Performance Comparison for 100 clients: C-Bits (resp. S-Bits) denotes the number of bits sent by each client (resp. the server). t is the number of clients for the treshold decryption.

	KRE-YGC	KRE-AHE			KRE-AHE*	
t	n/a	1	2	100	1	2
Time (s)	197.00	353.00	336.00	441.00	1024.00	1749.00
C-Bits (MB)	0.31	0.30	0.30	0.32	0.56	1.11
S-Bits (MB)	5.42	56.07	56.12	60.56	111.37	222.67

		n								
		10	11	12	13	14	15	16	17	18
[1]	time (s)	2.09	3.37	3.88	6.26	6.30	13.50	14.48	21.69	23.38
	B-C (MB)	13.50	18.21	20.03	25.69	27.83	50.13	53.71	64.97	69.03
Kre-Ygc	time (s)	1.20	1.31	1.59	2.02	2.34	2.43	3.02	3.31	3.76
	B-C (KB)	30.62	33.24	37.02	39.64	43.43	46.05	49.83	52.46	56.23
	B-S (KB)	68.55	81.36	95.24	110.22	126.27	143.40	161.62	180.92	201.28
Kre-Ahe	time (s)	3.45	3.96	4.74	4.84	5.31	5.71	5.98	6.70	6.86
	B-C (KB)	28.41	34.66	35.22	41.47	42.02	48.27	48.83	55.08	55.63
	B-S (KB)	575.15	701.25	840.15	991.21	1155.15	1331.14	1520.10	1721.06	1935.05

Table 4: Performance Comparison to [1]: Rows B-C/S is the communication for each client/server.

We evaluated Algorithm 5 of Kre-She at security level at least 110. The result is illustrated in Figure 6e for inputs with bitlength $\mu = 16$. The computation is dominated by the inputs' comparison and takes less than one hour for 25 clients. We also evaluated in Figure 6f the performance of the threshold decryption with a n-out-of-n secret sharing. For up to 40 clients threshold decryption costs less than 0.15 second. Kre-She is practically less efficient than all other schemes, but has the best asymptotic complexity.

As a result Kre-Ygc is suitable if the server is non-colluding and clients cannot fail. If collusion and failure are an issue, then either Kre-Ahe or Kre-She is suitable. Kre-She has the best asymptotic complexity, but, requires more efficient SHE.

Comparison to [1]. We implemented the semi-honest scheme of Aggarwal et al. [1] using MP-SPDZ [29] which is the state-of-the-art framework for secret sharing based multiparty computation. We evaluated Kre-Ygc, Kre-Ahe and [1] on a machine with a 6-core Intel(R) Xeon(R) E-2176M CPU @ 2.70GHz and 32GB of RAM. The input bitlength is 32. For evaluating [1], we used MP-SPDZ's option for semi-honest Shamir. A summary of the evaluation in Table 4 shows that our schemes scale better for increasing values of n.

10 Conclusion

In this paper, we considered the problem of computing the KRE (with applications to benchmarking) of n clients' private inputs. We proposed and compare different approaches based on garbled circuits or threshold HE. The computation is supported by the server which coordinates the protocol and undertakes as much computations as possible. The server is oblivious, and does not learn the input of the clients. We also implemented and evaluated our schemes.

Acknowledgments

We thank the anonymous reviewers for their valuable comments, and Andreas Fischer and Jonas Böhler for helpful contribution to some implementations.

References

1. Aggarwal, G., Mishra, N., Pinkas, B.: Secure computation of the k th-ranked element. In: EUROCRYPT. pp. 40–55 (2004)
2. Bellare, M., Hoang, V.T., Rogaway, P.: Foundations of garbled circuits. In: CCS. pp. 784–796. CCS '12 (2012)
3. Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for non-cryptographic fault-tolerant distributed computation. In: STOC. pp. 1–10. ACM, New York, NY, USA (1988)
4. Blass, E., Kerschbaum, F.: Strain: A secure auction for blockchains. In: ESORICS (1). vol. 11098, pp. 87–110. Springer (2018)
5. Blass, E., Kerschbaum, F.: Secure computation of the k^{th}-ranked integer on blockchains. IACR Cryptology ePrint Archive **2019**, 276 (2019)
6. Boneh, D., Gennaro, R., Goldfeder, S., Jain, A., Kim, S., Rasmussen, P.M.R., Sahai, A.: Threshold cryptosystems from threshold fully homomorphic encryption. In: CRYPTO. pp. 565–596 (2018)
7. Boneh, D., Goh, E.J., Nissim, K.: Evaluating 2-dnf formulas on ciphertexts. In: TCC '05. pp. 325–341. Springer-Verlag, Berlin, Heidelberg (2005)
8. Boneh, D., Shoup, V.: A graduate course in applied cryptography. https://crypto.stanford.edu/ dabo/cryptobook/ (2017)
9. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: Fully homomorphic encryption without bootstrapping. ECCC **18**, 111 (2011)
10. Catrina, O., Kerschbaum, F.: Fostering the uptake of secure multiparty computation in e-commerce. In: PARES '08. pp. 693–700. ARES '08 (2008)
11. Cheon, J.H., Kim, M., Lauter, K.E.: Homomorphic computation of edit distance. In: FC. pp. 194–212 (2015)
12. Damgård, I., Geisler, M., Krøigaard, M.: Efficient and secure comparison for on-line auctions. In: ACISP. pp. 416–430 (2007)
13. Damgård, I., Keller, M., Larraia, E., Pastro, V., Scholl, P., Smart, N.P.: Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In: ESORICS '13. pp. 1–18 (2013)
14. Ejgenberg, Y., Farbstein, M., Levy, M., Lindell, Y.: SCAPI: the secure computation application programming interface. IACR Cryptology ePrint Archive (2012)
15. Feige, U., Killian, J., Naor, M.: A minimal model for secure computation (extended abstract). In: STOC '94. pp. 554–563. STOC '94 (1994)
16. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game. In: STOC. pp. 218–229. ACM, New York, NY, USA (1987)
17. Halevi, S., Shoup, V.: Algorithms in helib. In: CRYPTO (1). Lecture Notes in Computer Science, vol. 8616, pp. 554–571. Springer (2014)
18. Helib. https://github.com/homenc/HElib (2019)
19. Hu, Y., Martin, W., Sunar, B.: Enhanced flexibility for homomorphic encryption schemes via crt. In: ACNS (2012)
20. Kamara, S., Mohassel, P., Raykova, M.: Outsourcing multi-party computation. IACR Cryptology ePrint Archive **2011**, 272 (2011)
21. Keller, M., Orsini, E., Scholl, P.: Mascot: Faster malicious arithmetic secure computation with oblivious transfer. In: CCS '16. pp. 830–842 (2016)
22. Kerschbaum, F.: Building a privacy-preserving benchmarking enterprise system. Enterprise IS **2**(4), 421–441 (2008)
23. Kerschbaum, F.: Adapting privacy-preserving computation to the service provider model. In: CSE. pp. 34–41 (2009)

24. Koblitz, N.: Elliptic curve cryptosystems. Mathematics of Computation **48**(177), 203–209 (Jan 1987)
25. Kolesnikov, V., Sadeghi, A., Schneider, T.: Improved garbled circuit building blocks and applications to auctions and computing minima. In: CANS. pp. 1–20 (2009)
26. Kolesnikov, V., Schneider, T.: Improved garbled circuit: Free XOR gates and applications. In: ICALP. pp. 486–498 (2008)
27. Lin, H., Tzeng, W.: An efficient solution to the millionaires' problem based on homomorphic encryption. In: ACNS. pp. 456–466 (2005)
28. Lindell, Y., Pinkas, B.: A proof of security of yao's protocol for two-party computation. J. Cryptol. **22**(2), 161–188 (Apr 2009)
29. Multi-protocol spdz. https://github.com/data61/MP-SPDZ (2019)
30. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: EUROCRYPT'99. pp. 223–238. Springer-Verlag (1999)
31. Smart, N.P., Vercauteren, F.: Fully homomorphic simd operations. Des. Codes Cryptography **71**(1), 57–81 (Apr 2014)
32. Yao, A.C.: Protocols for secure computations. In: SFCS '82. pp. 160–164. SFCS '82, IEEE Computer Society, Washington, DC, USA (1982)
33. Zahur, S., Rosulek, M., Evans, D.: Two halves make a whole - reducing data transfer in garbled circuits using half gates. In: EUROCRYPT '15. pp. 220–250 (2015)

A Difference of our Modification to the original LinTzeng protocol

We discuss in this section the difference of our modification to the original protocol of Lin and Tzeng [27]. In contrast to the original scheme [27], we note the following differences. Firstly, we use additively HE instead of multiplicative. It suits best with our setting and can be implemented using ElGamal on elliptic curve with better performance and smaller ciphertexts. Although decryption requires solving the discrete logarithm, this is not necessary since we are looking for ciphertexts encrypting 0. Secondly, instead of relying on a collision-free hash function as [27], we use the Int function which is simpler to implement and more efficient as it produces smaller values. Thirdly, we choose the random encoded elements as explained above and encrypt them, while the original protocol uses ciphertexts chosen randomly in the ciphertext space. Fourthly, in the original protocol, the evaluator has access to x_j in plaintext and does not need to choose random encoded elements. By encoding as explained in our modified version, we can encrypt both encodings and delegate the evaluation to a third party which is not allowed to have access to the inputs in plaintext. In fact, we can executes all $O(n^2)$ comparisons non-interactively at the server, but this requires threshold decryption for $O(n^2)$ elements. That is, computing the comparisons interactively turned out to be more efficient.

B Decryption of Additive ElGamal

In our evaluation of Kre-Ahe, we use additive ElGamal which requires solving the discrete logarithm. First, we stress that in the comparison protocols (both

DGK and LinTzeng), computing the discrete logarithm is not necessary as the generator is looking for a ciphertext of zero. Recall that an additive ElGamal ciphertext for a plaintext m looks like (g^r, g^{sr+m}), where r is random number, g is the generator of the group, and s is the private key. Hence, checking if the ciphertext is encrypting 0, is done by computing g^{rs} from the first component and then checking if the result equals the second component.

The problem of computing the discrete logarithm in the decryption of additive ElGamal is similar to the somewhat encryption scheme of Boneh et al. [7] (BGN) and several pairing-based schemes. As stated in [7], one can limit the message space to short messages and pre-compute a logarithm table such that the decryption occurs in constant time. To handle larger messages, we use CRT ElGamal [19], which decomposes a large message in smaller messages according to the chinese remainder theorem (CRT) and encrypts the smaller messages. We briefly describe it and refer to [19] for more details.

Let $sk = s$ be the private key and g be the generator of the group. Choose l small moduli $d_i \in \mathbb{Z}^+$ such that $gdc(d_i, d_j) = 1$ for $i \neq j$. Then the plaintext space is $\mathcal{M} = \{0, \ldots, N\}$ where $N < d = \Pi_{i=1}^{l} d_i$. The public key is $(g, h = g^s, \langle d_1, \ldots, d_l \rangle)$.

To encrypt a message $m \in \mathcal{M}$, compute $\langle (g^{r_1}, h^{r_1} g^{m_1}), \ldots, (g^{r_l}, h^{r_l} g^{m_l}) \rangle$, where $m_i = m \bmod d_i$ and r_i is a random number.

To decrypt a ciphertext $\langle (u_1, v_1), \ldots, (u_l, v_l) \rangle$, compute

$$CRT^{-1}(\langle \log_g(v_1 u_1^{-s}), \ldots, \log_g(v_l u_l^{-s}) \rangle),$$

where $CRT^{-1}(\langle x_1, \ldots, x_l \rangle) = \sum_{i=1}^{l} x_i \frac{d}{d_i} (\frac{d^{-1}}{d_i} \bmod d_i) \mod d$, and $\log_g(\cdot)$ denotes the logarithm w.r.t g.

Given several ciphertexts, it is straightforward to compute the ciphertext encrypting the sum of the corresponding plaintexts. This is done by multiplying the given ciphertexts componentwise.

C Proof of Lemmas

C.1 Proof of Lemma 1

Proof. Since r_i is the rank of x_i, x_i is by definition larger or equal to r_i elements in $\{x_1, \ldots, x_n\}$. This means that r_i values among $b_{i1}, \ldots, b_{in}$ are 1 and the remaining $n - r_i$ values are 0. It follows that $\sum_{j=1}^{n} b_{ij} = r_i$.

C.2 Proof of Lemma 2

Proof. Let $\mathbb{P} = \{(x_i, x_j) : x_i, x_j \in \mathbb{X} \wedge i \neq j\}$, $\mathbb{P}_1 = \{(x_i, x_j) : x_i, x_j \in \mathbb{X} \wedge \textsc{Paired}(i, j) = true\}$, $\mathbb{P}_2 = \{(x_i, x_j) : x_i, x_j \in \mathbb{X} \wedge Q(i, j) = true\}$, where $Q(i, j)$ is defined as follows:

$$\begin{aligned} Q(i,j) :=\ & (i \equiv 1 \pmod 2 \wedge i < j \wedge j \equiv 1 \pmod 2) \vee && \text{(2a)} \\ & (i \equiv 0 \pmod 2 \wedge i > j \wedge j \equiv 1 \pmod 2) \vee && \text{(2b)} \\ & (i \equiv 0 \pmod 2 \wedge i < j \wedge j \equiv 0 \pmod 2) \vee && \text{(2c)} \\ & (i \equiv 1 \pmod 2 \wedge i > j \wedge j \equiv 0 \pmod 2). && \text{(2d)} \end{aligned}$$

Clearly, $\mathbb{P}$ contains the maximum number of comparisons required to compute the rank of every $x_i \in \mathbb{X}$. Now it suffices to show that:

1. $\mathbb{P}_1$ and $\mathbb{P}_2$ form a partition of $\mathbb{P}$
2. $\forall\ (x_i, x_j) \in \mathbb{P} : (x_i, x_j) \in \mathbb{P}_1 \Leftrightarrow (x_j, x_i) \in \mathbb{P}_2$

$\mathbb{P}_1$ and $\mathbb{P}_2$ are clearly subsets of $\mathbb{P}$. For each $(x_i, x_j) \in \mathbb{P}$, (i, j) satisfies exactly one of the conditions (1a), ..., (1d), (2a), ..., (2d), hence $\mathbb{P} \subseteq \mathbb{P}_1 \cup \mathbb{P}_2$. Moreover, for each $(x_i, x_j) \in \mathbb{P}$, either $\textsc{Paired}(i, j) = \mathit{true}$ or $Q(i, j) = \mathit{true}$. It follows that $\mathbb{P}_1 \cap \mathbb{P}_2 = \varnothing$ which concludes the proof of claim 1. To prove claim 2, it suffices to see that, (i, j) satisfies condition (1a) if and only if (j, i) satisfies condition (2a). The same holds for (1b) and (2b), (1c) and (2c), (1d) and (2d).

C.3 Proof of Lemma 3

Proof. This is actually a corollary of the proof of Lemma 2. It follows from the fact that $(x_i, x_j) \in \mathbb{P}_1 \Leftrightarrow (x_j, x_i) \in \mathbb{P}_2$ and any x_i is involved in $n-1$ comparisons (since we need $b_{i1}, \ldots, b_{in}$ to compute $r_i = \sum_{j=1}^{n} b_{ij}$, where we have $b_{ii} = 1$ without comparison). This proves the case when n is odd. If n is even then the odd case applies for $n' = n - 1$. Then for each $i \in \{1, \ldots, n'\}$, we have $\textsc{Paired}(i, n) = \mathit{true}$ if i is odd (condition 1b) and $\textsc{Paired}(n, i) = \mathit{true}$ if i is even (condition 1c).

C.4 Proof of Lemma 4

Proof. If $V = V^1_{x_i} - V^0_{x_j}$ has a unique 0 at a position $l, (1 \le l \le \mu)$ then u_{il} and v_{il} have bit representation $y_{\mu-l+1} \cdots y_1$, where for each $h, \mu - l + 1 \ge h \ge 2$, $y_h = x_{ig} = x_{jg}$ with $g = l + h - 1$, and $y_1 = x_{il} = 1$ and $x_{jl} = 0$. It follows that $x_i > x_j$.
If $x_i > x_j$ then there exists a position l such that for each $h, \mu \ge h \ge l+1$, $x_{ih} = x_{jh}$ and $x_{il} = 1$ and $x_{jl} = 0$. This implies $u_{il} = v_{il}$.

For $h, \mu \ge h \ge l + 1$, either u_{ih} bit string is a prefix of x_i while v_{jh} is random, or u_{ih} is random while v_{jh} bit string is a prefix of x_j. From the choice of $r^{(0)}_{ih}$, $r^{(1)}_{ih}$, we have $u_{ih} \neq v_{ih}$.

For $h, l - 1 \ge h \ge 1$ there are three cases: u_{ih} and v_{ih} (as bit string) are both prefixes of x_i and x_j, only one of them is prefix, both are random. For the first case the difference of the bits at position l and for the other cases the choice of $r^{(0)}_{ih}$ imply that $u_{ih} \neq v_{ih}$.

C.5 Proof of Lemma 5

Proof. It is clear from the definition that $\mathbb{X}_i \subseteq \mathbb{X}$ for all i and since $i-(i-t+1)+1 = t$, $\mathbb{X}_i$ has exactly t elements. Let x_i be in $\mathbb{X}$, then from the definition, x_i is element of only the subsets $\mathbb{X}_i, \mathbb{X}_{i+1}, \ldots, \mathbb{X}_{i+t-1}$, where indexes of the $\mathbb{X}_i$ are computed mod n. Again, it holds $(i + t - 1) - i + 1 = t$.

D Security Proofs

Let the inherent leakage be $\mathcal{L} = \{k, n, t, \kappa, \lambda, \mu\}$, i.e., protocol's parameters.

Theorem 1. *If the server S is non-colluding and the AHE scheme is IND-CPA secure, then* Kre-Ygc *1-privately computes $\mathcal{F}_{\text{KRE}}$ in the semi-honest model with leakage $\mathcal{L}_S = \mathcal{L}_i = \mathcal{L}$. Hence, there are simulators SIM_{C_i} for each C_i and SIM_S for S such that:*

$$\text{SIM}_S(\varnothing, \mathcal{L}_S) \stackrel{c}{\equiv} \mathsf{View}_{\mathsf{S}}(x_1, \ldots, x_n) \text{ and}$$

$$\text{SIM}_{C_i}(x_i, \mathcal{F}_{\text{KRE}}(x_1, \ldots, x_n), \mathcal{L}_i) \stackrel{c}{\equiv} \mathsf{View}_{\mathsf{C_i}}(x_1, \ldots, x_n).$$

Proof (Sketch). The leakage is clear as parties see only random strings. $\mathsf{View}_{\mathsf{C_i}}$ consists of:

$$(F_{>}^{ij}, (\bar{a}_i^{ij}, \bar{x}_i^{ij}), b'_{ij}, [\![a_j^{ij} \oplus b_{ij}]\!]_i, [\![a_i^{ij} \oplus b_{ij}]\!]_j, [\![b_{ij}]\!]_j)_{1 \le j \le n (i \ne j)}, [\![\beta_i]\!]_i, \beta_i.$$

For each $m \in \mathsf{View}_{\mathsf{C_i}}$, SIM_{C_i} chooses random bit strings of length $|m|$. The view of the server consists of:

$$\langle F_{>}^{ij}, (\bar{a}_i^{ij}, \bar{x}_i^{ij}), (\bar{a}_j^{ij}, \bar{x}_j^{ij}), b'_{ij}, [\![a_j^{ij} \oplus b_{ij}]\!]_i, [\![a_i^{ij} \oplus b_{ij}]\!]_j, [\![a_i^{ij} \oplus b_{ij}]\!]_j, [\![a_j^{ij} \oplus b_{ij}]\!]_i,$$
$$[\![b_{ij}]\!]_j, [\![b_{ij}]\!]_i \rangle_{\text{Paired}(i,j)=true}, \langle [\![r_i]\!]_i, [\![\beta_i]\!]_i \rangle_{1 \le i \le n}, \langle [\![m_i]\!]_j \rangle_{1 \le i,j \le n}.$$

For each $m \in \mathsf{View}_{\mathsf{S}}$, SIM_S chooses random bit strings of length $|m|$.

Theorem 2. *Let $t \in \mathbb{N}$ and $\tau < t$. If the server S is non-colluding and the AHE scheme is IND-CPA secure, then* Kre-Ahe *τ-privately compute $\mathcal{F}_{\text{KRE}}$ in the semi-honest model with leakage $\mathcal{L}_S = \mathcal{L}_i = \mathcal{L}$. Hence, let $I = \{i_1, \ldots, i_\tau\}$, $\mathcal{L}_I = \bigcup_{i \in I} \mathcal{L}_i$, there exists a simulator SIM_I such that:*

$$\text{SIM}_I((x_{i_1}, \ldots, x_{i_\tau}), \mathcal{F}_{\text{KRE}}(x_1, \ldots, x_n), \mathcal{L}_I) \stackrel{c}{\equiv} \mathsf{View}_{\mathsf{I}}(x_1, \ldots, x_n).$$

Proof (Sketch). The leakage is clear as parties see only random strings (IND-CPA ciphertexts, random shares or random bits).

In Kre-Ahe , all messages can be simulated by choosing random bit strings of the corresponding length. However, the simulation of Step 18 must be coherent with Step 22. Each client receives random shares in Step 18, runs the final decryption $Decf(.)$ in Step 22 and learns a random message. To simulate Steps 18 and 22, the simulator chooses t random values for Step 18 such that running $Decf(.)$ returns the random message simulated in Step 22.

For example, if the underlying AHE is ECC ElGamal (ECE), then a ciphertext has the form $c = (\alpha_1, \alpha_2) = (r \cdot P, m \cdot P + r \cdot pk)$. For each ECE ciphertext $c = (\alpha_1, \alpha_2) = (r \cdot P, m \cdot P + r \cdot pk)$ that must be final decrypted in Step 22, C_j gets α_2 and t partial decryption results $\alpha_{11}, \ldots, \alpha_{1t}$ of α_1 in Step 18. To simulate this, the simulator chooses a random message m and a random $\tilde{\alpha}_2$. Then it computes $\tilde{\alpha}_1 = \tilde{\alpha}_2 - m \cdot P$ and generates random $\tilde{\alpha}_{11}, \ldots, \tilde{\alpha}_{1t}$ such that $\sum_{i=1}^{t} \tilde{\alpha}_{1i} = \tilde{\alpha}_1$ in $\mathbb{G}$.

Theorem 3. *Let $t \in \mathbb{N}$ and $\tau < t$. If the server S is non-colluding and the SHE scheme is IND-CPA secure, then* Kre-She *τ-privately computes $\mathcal{F}_{\text{KRE}}$ in the semi-honest model with leakage $\mathcal{L}_S = \mathcal{L}_i = \mathcal{L}$. Hence, let $I = \{i_1, \ldots, i_\tau\}$ denote the indexes of corrupt clients, $\mathcal{L}_I = \bigcup_{i \in I} \mathcal{L}_i$ denote their joint leakages and $\mathsf{View}_{\mathsf{I}}(x_1, \ldots, x_n)$ denote their joint views, there exists a simulator SIM_I such that:*

$$\text{SIM}_I((x_{i_1}, \ldots, x_{i_\tau}), \mathcal{F}_{\text{KRE}}(x_1, \ldots, x_n), \mathcal{L}_I) \stackrel{c}{\equiv} \mathsf{View}_{\mathsf{I}}(x_1, \ldots, x_n).$$

Proof (Sketch). The leakage is clear as parties see only random strings (IND-CPA ciphertexts or partial decryption results). The security is also straightforward as the computation is almost completely done by the server alone and encrypted under an IND-CPA encryption. Moreover, the partial decryption reveals only partial result to each decryptor.

Recall that our adversary is semi-honest. In Kre-Ygc, a server collusion reveals all inputs to the adversary. In Kre-Ahe, a server collusion only increase the leakage as long as the number of corrupted clients is smaller than t. For example, the adversary can learn the rank of corrupted clients. In Kre-She, the KRE is homomorphically computed by the server such that the clients are only required for the decryption of one ciphertext encrypting the KRE. Moreover, the ciphertexts are encrypted using the threshold public key. As a result, assuming semi-honest adversary and a collusion set containing less than t clients, a server collusion leaks no more information than $k, n, t, \kappa, \lambda, \mu$.

E Complexity analysis

In this section, we discuss the complexity of our schemes. We will use κ and λ as length of asymmetric ciphertext and symmetric security parameter.

E.1 Kre-Ygc Protocol

A GC for the comparison of two μ-bit integers consists of μ AND-gates resulting in 4μ symmetric ciphertexts [25, 26]. It can be reduced by a factor of 2 using the *halfGate* optimization [33] at the cost of performing two cheap symmetric operations (instead of one) during GC evaluation.

We do the analysis for the case where n is odd (the even case is similar). From Lemma 3, each client generates $(n-1)/2$ GCs resulting in $(n-1)\mu$ symmetric operations. The computation of encrypted comparison bits (Steps 6 to 16) and the computation of the KRE's ciphertext require $O(n)$ asymmetric operations to each client. Finally, each client has to decrypt one ciphertext in Step 23. As a result, the computation complexity of each client is therefore $O((n-1)\mu)$ symmetric and $O(2n+1)$ asymmetric operations. In communication, this results in $n\kappa$ bits for the asymmetric ciphertexts, $2\mu\lambda(n-1)/2$ bits for the GCs and $\mu\lambda(n-1)/2$ for the garbled inputs and $n\kappa$ bits for handling the server's leakage. In total each client sends $2n\kappa + \frac{3\mu\lambda(n-1)}{2}$.

The server evaluates $n(n-1)/2$ GCs each consisting of 2μ symmetric ciphertexts. Computing the rank (Steps 17 to 19) requires $O(n \log n + n)$ operations to the server. Finally, the server evaluates $\log n + n$ asymmetric operations to compute the KRE ciphertext for each client (Steps 23 to 24). The total computation complexity of the server is $O(n(n-1)\mu)$ symmetric and $O((n+1)\log n + 2n)$. In communication, the server sends $n(n-1)$ asymmetric ciphertexts in Steps 6 to 16, n asymmetric ciphertexts in Steps 17 to 19 and n asymmetric ciphertexts in Steps 23 to 24. This results in a total of $(n^2+n)\kappa$ bits.

E.2 Kre-Ahe Protocol

Since Kre-Ahe also requires the predicate Paired as Kre-Ygc, we do the analysis for the case where n is odd (the even case is similar).

Each client performs $O(\mu+1)$ operations in Step 2, $O(\frac{7\mu(n-1)}{2})$ operations in Step 6, $O(t)$ operations in Step 16 and $O(\log t)$ in Step 21, $O(n)$ operations in Step 23 and $O(1)$ operations in Step 26. This results in a total of $O(\mu + \frac{7\mu(n-1)}{2} + t + \log t + n + 1)$ asymmetric operations.

Each client sends $(\mu+1)\kappa$ bits in Step 2, $\frac{\kappa(n-1)}{2}$ bits (when the client is head) and $\frac{(\mu+1)\kappa(n-1)}{2}$ (when the client is tail) in Step 6, $t\kappa$ bits in Step 16 and $n\kappa$ bits in Step 23. This results in a total of $(\mu\frac{(n+1)}{2} + 2n + t)\kappa$ bits for each client.

The cryptographic operations of the server happen in ComputeKreAhe (Algorithm 2) that is called in Step 7 of Protocol 4. The server performs $O(n^2+n)$ asymmetric operations.

The server sends $\frac{(\mu\kappa+(\mu+1)\kappa)n(n-1)}{2}$ bits in Step 6, $nt\kappa$ bits in Steps 12 and 18, $n\kappa$ bits in Step 25. This results in a total of $(\frac{(2\mu+1)n(n-1)}{2} + 2nt + n)\kappa$ bits for the server.

E.3 Kre-She Protocol

Each client has $O(\mu)$ computation cost ($\mu+1$ encryptions and eventually one partial decryption and a communication cost of $(\mu+n+1)\kappa$ bits.

The cryptographic operations of the server happen in ComputeKreShe (Algorithm 5). The SHE comparison circuit has depth $\log(\mu-1)+1$ and requires $O(\mu \log \mu)$ homomorphic multiplications [11]. For all comparisons the server performs, therefore, $O(n^2 \mu \log \mu)$ multiplication. In Step 10 of Algorithm 5, the computation of $[\![\prod_{j=1,j\neq k}^{n}(r_i - j)]\!]$ has depth $\log n$ and requires $O(n \log n)$ homomorphic multiplications. Step 12 of Algorithm 5 adds an additional circuit depth and requires $O(n)$ homomorphic multiplications. As a result, Algorithm 5 has a total depth of $\log(\mu-1) + \log n + 2$ and requires $O(n^2\mu \log \mu + n \log n + n)$ homomorphic multiplications. The server sends $(t+nt)\kappa$ bits in the threshold decryption.