Secure Sorting and Selection via Function Secret Sharing

Amit Agarwal
University of Illinois Urbana-Champaign
Urbana, USA
amita2@illinois.edu

Elette Boyle
NTT Research
Sunnyvale, USA
Reichman University
Herzliya, Israel
eboyle@alum.mit.edu

Nishanth Chandran
Microsoft Research
Bengaluru, India
nichandr@microsoft.com

Niv Gilboa
Ben Gurion University
Beersheba, Israel
gilboan@bgu.ac.il

Divya Gupta
Microsoft Research
Bengaluru, India
divya.gupta@microsoft.com

Yuval Ishai
Technion
Haifa, Israel
yuvali@cs.technion.ac.il

Mahimna Kelkar
Cornell University
New York, USA
mahimna@cs.cornell.edu

Yiping Ma
University of Pennsylvania
Philadelphia, USA
yipingma@seas.upenn.edu

ABSTRACT

We revisit the problem of concretely efficient secure computation of sorting and selection (e.g., maximum, median, or top-k) on secret-shared data, focusing on the case of security against a single semi-honest party. Previous solutions either have a high communication overhead or many rounds of interaction, even when allowing input-independent preprocessing.

We propose a suite of 2-party and 3-party offline-online protocols that exploit the efficient aggregation feature of *function secret sharing* to minimize the online communication and rounds. In particular, most of our protocols are optimal in terms of both online communication and online rounds up to small constant factors.

We compare the performance of our protocols with prior works for different input parameters (number of items, bit length of items, batch size) and system parameters (CPU cores, network) and obtain up to 14× improvement in online run time for sorting and selection under some settings.

CCS CONCEPTS

• **Security and privacy → Cryptography**.

KEYWORDS

Secure sorting; Secure selection; Secure multiparty computation; Function secret sharing

ACM Reference Format:
Amit Agarwal, Elette Boyle, Nishanth Chandran, Niv Gilboa, Divya Gupta, Yuval Ishai, Mahimna Kelkar, and Yiping Ma. 2024. Secure Sorting and Selection via Function Secret Sharing. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690359

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690359

1 INTRODUCTION

Sorting a list of n numbers (potentially with associated payloads) and selecting the k-th ranked element from such a list (e.g., maximum or median) are ubiquitous operations that serve as fundamental building blocks in numerous applications. In many of these use cases, sorting or selection need to be performed on sensitive data. This motivated a large body of work on efficient *secure computation* protocols for these algorithmic problems (see Section 1.2 for a partial survey). In the current work, we revisit this problem and present a suite of new protocols, based on *function secret sharing* [16], that offer unique efficiency features.

To enable a modular use of secure protocols as sub-protocols, it is typically assumed that both the input and output are secret-shared between two or more parties. Protocols are often divided into an input-independent *offline phase*, which can be carried out before the inputs are known, and an input-dependent *online phase*, which is facilitated by the offline phase. Here we consider the simplest setting for secure computation, in which security needs to hold against any single *semi-honest* party.[1] We consider both 3-party protocols and a useful special class of such protocols known as (2+1) protocols, where one of the parties acts as a "dealer" that only needs to send correlated randomness to the other two parties in the offline phase. For consistency of our protocol descriptions, we assume that the input is secret-shared between two of the parties in both the 2 + 1 and the 3-party setting. Furthermore, motivated by concrete efficiency, we restrict our attention to protocols that only make use of symmetric cryptography.

Even in the above liberal setting, all previous protocols for sorting or selection from the literature have an online phase which is far from optimal with respect to *either* communication complexity or

[1] While our main focus is on the semi-honest security model, most of our protocols can be enhanced to provide security against malicious parties, with a small overhead to the online phase, by making use of homomorphic MACs as in the SPDZ line of work [10, 23]. See the full version for more details.

round complexity. For example, protocols based on the classical garbled circuit (GC) technique [33] require only two online rounds, but their online communication is bigger than the input size by a security-parameter multiplicative factor (128 in practice). On the other hand, protocols based on secret sharing, including specialized protocols that mimic a radix-sort algorithm, have a high number of online rounds that grows with the number of values n or their bit-length ℓ. To give a concrete example, the online round complexity of a state-of-the-art protocol of Asharov et al. [7] is bigger than 10ℓ; while this is reasonable when running over a fast local network, it may lead to an unacceptably high latency when the parties are physically far from each other.

1.1 Overview of results and techniques

Motivated by the above limitation of previous techniques, in this work we explore a new approach that *simultaneously* offers near-optimal online communication and rounds (up to small multiplicative constants), with a moderate computational overhead. This gives rise to new protocols with unique efficiency features for sorting, selection, top-k, sorted top-k, and more. Our technical approach employs a novel combination of cryptographic and algorithmic techniques that can be instantiated in several ways. This yields a variety of protocols that offer different trade-offs between concrete round complexity, communication, and computation.

Protocol Abstraction. We divide the cleartext sorting (or selection) algorithms into two phases - *ranking* and *routing*. In the ranking phase, the *rank* of each element (namely, the number of smaller elements) is computed. In the routing phase, the inputs are re-ordered according to their rank (in the case of sorting) or the element(s) with the specific rank(s) are selected (in the case of selection). We then propose two distinct approaches to securely realize each phase, resulting in 4 different protocols summarized in Table 1.

1.1.1 Cryptographic Protocols with minimal rounds. We heavily leverage recent techniques for *function secret sharing* (FSS) [16], a lightweight tool for splitting simple but useful types of functions into succinct function shares. These function shares can be locally evaluated on any given input and produce *linear shares* of the function output. Linearity of the output shares enables *succinct aggregation*, a key feature of FSS that we exploit in our work: one can locally perform many (different) FSS evaluations on a set of inputs and then add up different subsets of the secret-shared outputs without any communication, except when reconstructing the aggregated outputs. In some cases, we will rely on FSS with redundant output shares that enable multiplication of secret-shared outputs before performing aggregation. It is important to note that even though some protocols require a super-linear (in the number of inputs n) number of *local* FSS evaluations, the communication required only depends on number of aggregated outputs that can be much smaller, e.g., linear (in n).

FSS-based Ranking. *Distributed comparison function* (DCF) [14, 17] is a special type of FSS scheme that supports comparison of secret-shared values. Combining the succinct aggregation feature of FSS with the secure comparison functionality provided by DCFs enables non-interactive and succinct secure computation of a secret-shared *ranking* of input values as follows: We use $\binom{n}{2}$ instances of DCF to compare all input pairs with each other in parallel, and separately aggregate the comparison results for each input entry (Section 3.1). We denote this protocol by $\binom{n}{2}$-CmpAgg. This approach can also be used to compute *stable* ranking, which break ties between identical values according to the order in which they appear in the input. For ranking n secret-shared inputs of bit-length ℓ in the 2 + 1 party setting, this approach requires an online communication cost of $2n\ell$ bits, 1 online round, online computation cost equivalent to $n^2\ell$ calls to a length-doubling PRG, and an offline communication of $\approx \lambda n^2\ell$ bits, where λ is the PRG seed length.

Our second protocol for secure ranking, denoted by mult-DPF (Section 3.2), relies on a recently introduced *multiplicative* variant of another instance of FSS called *distributed point function* (DPF) that supports non-interactive computation of degree-2 polynomials on the output shares [1, 19]. The advantage of this ranking technique is that its offline communication and online computational cost scale linearly (rather than quadratically) with n, without requiring additional interaction. The downside is that its online computational cost scales exponentially with the bit-length ℓ of the ranked values, and is therefore only practical when these values come from a small domain (but may have arbitrarily large associated payloads). Moreover, sorting in this small-domain regime can be efficiently bootstrapped into sorting in the large-domain regime via a radix sort approach [7]. Finally, another downside of this approach is the need for a third online party, making it unsuitable for the 2+1 setting. For ranking n secret-shared inputs of bit-length ℓ in the 3-party setting, this approach requires an online communication of $4n\ell$ bits, 1 online round, online computation cost equivalent to $3nL$ calls to a length-doubling PRG, where $L = 2^\ell$, and an offline communication of $\approx 36\lambda n\ell$ bits.

FSS-based Routing. Following the *ranking* step, where we compute the secret-shared rank of each input entry, we need to perform a secure *routing* step where the secret inputs are reordered according to their rank (for sorting) or the input with required rank is copied to the output (for selection). This should be done while hiding the ranks of individual inputs and the routing pattern. A traditional approach for achieving this is by invoking a *secret-shared shuffle* [21] subprotocol, which applies a random secret permutation to the secret-shared inputs *even before the ranking phase*. If this is done, it is safe to reveal the (stable) ranks of the shuffled inputs. Given the ranks, the routing is trivial and can be done in the clear (for secret-shared values). We present this approach in Section 4.1. Additionally, we present an alternative new method of *DPF-based routing*, which improves over the online round complexity of the above shuffle-based approach, in Section 4.2. When combined with a 1-round ranking approach described earlier, the shuffling and DPF based routing approaches lead to a sorting/selection protocol with 3 and 2 online rounds respectively.

Secure Sorting and Selection. The costs of four sorting and selection protocols obtained by combining our two ranking ($\binom{n}{2}$-CmpAgg and mult-DPF) and two routing techniques (Shuffle and DPF) are summarized in Table 1. We compare these with the classical approach of using a garbled circuit (GC) in offline-online paradigm, which has similar online round complexity but a much

Protocol			Parties	Online			Offline	Ref
ID	Routing	Ranking		Communication (bits)	Rounds	Computation (PRG calls)	Communication (bits)	
Garbled circuit			$2+1$	$\geq \lambda n\ell$	2	$> \frac{3}{8} n\ell \log^2 n$	$> 1.5\lambda n\ell \log^2 n$	[8, 29, 31]
I	Shuffle	$\binom{n}{2}$-CmpAgg	$2+1$	$4n\ell + 2n\log n$	3	$n^2\ell$	$\lambda n^2\ell$	Thm. 4.1
II	Shuffle	mult-DPF	3	$8n\ell + 4n\log n$	3	$3nL$	$36\lambda n\ell$	Thm. 4.1 full ver.
III	DPF	$\binom{n}{2}$-CmpAgg	$2+1$	$4n\ell + 2n\log n$	2	$n^2\ell$	$\lambda n^2\ell$	Thm. 4.2
IV	DPF	mult-DPF	3	$8n\ell + 4n\log n$	2	$\begin{cases} 3nL + n\log n & : \text{selection} \\ 3nL + 2n^2\ell/\lambda & : \text{sorting} \end{cases}$	$36\lambda n\ell + 2\lambda n\log n$	Thm. 4.2 full ver.
FSS gate			$O(1)$	$n\ell$	1	Barrier	Barrier	Thm. 4.3

Table 1: Summary of protocols for secure sorting and selection minimizing online rounds. The protocols map an additive secret-sharing of n inputs from $[L]$, with bit-length $\ell = \lceil \log_2 L \rceil$, to an additive secret-sharing of either n outputs (for sorting) or one output (for selection). The sharing is over a group $\mathbb{G} = \mathbb{Z}_L$. The protocols are secure against a semi-honest adversary corrupting a single party. The first row refers to the baseline of using garbled circuits, the last row refers to a barrier for obtaining efficient protocols with a single online round, and all other rows refer to our new FSS-based protocols. The "Routing" and "Ranking" columns refer to different technical approaches taken by our protocols. In the "Parties" column, $2+1$ refers to 2-party protocols that use offline correlated randomness provided by the dealer party (the dealer can be eliminated at an additional cost). The online (resp., offline) columns refers to total costs across all parties after (resp., before) inputs are known. In the $2+1$ case, offline communication refers to correlated randomness from the dealer party. All the entries refer to both selection and sorting except where indicated otherwise. We report computation cost as number of calls to a length-doubling PRG with a λ-bit seed. A call in which only half of the PRG output is used is counted as $\frac{1}{2}$ call (as this is true for AES-based PRG). Also, for Garbled Circuit, we count an invocation of a Random Oracle (RO) as half a PRG call. When using a standard AES-based PRG, the number of (fixed-key) AES calls needed is twice the number of calls to length doubling PRG. All costs ignore low-order additive terms and assume $\log n \ll \lambda$. All protocols can handle an extra payload of p bits, with a typical cost comparable to or better than replacing ℓ by $(\ell + p)$. See the full version for details.

higher online communication cost[2]. Each of the new protocols in the table has some competitive advantages over all other protocols (including prior ones) in some setting of parameters.

Protocol I is attractive when sorting or selecting from a small number of items (say, $n \leq 50$), especially when performing a batch of many such small sorting tasks. This may arise in many applications, ranging from computing MaxPool in DNNs to sorting the results of a private search or a random sample of database entries that satisfy a given criteria. We also show how to improve upon the quadratic scaling with n by allowing a higher (constant) number of rounds (Section 5). For instance, in case of sorting, online computation can be reduced to $O(n^{1+1/r})$ if we allow for $2r+1$ online rounds. In case of selection, online computation can be reduced to $O(n)$ if we allow for 9 online rounds. Protocol II is attractive when sorting values from a small domain size L (say, $\ell = \log L \leq 15$). This is directly motivated by many applications (e.g., sorting database records by height or age) and can be used as a building block for generalized radix sort [7].

Protocols III and IV use DPF-based routing and reduce the round complexity attained with Shuffle-based routing in Protocols I and II from 3 to 2. However, the DPF-based routing internally makes use of a multiplicative masking technique, which limits its applicability to secret sharing over *fields* (as opposed to shuffle based routing which can work over groups). Protocol IV has an additional advantage that its offline cost scales linearly with n. In the case of selection, its online computation also scales linearly with n. On the downside, it only applies in the 3-party setting, compared to the $2+1$ setting of Protocol III, and its online communication is slightly higher.

[2]Using oblivious transfer, the GC-based protocol can be implemented in the plain two-party setting (without a dealer) with similar costs. This is also the case for our protocols in the small-domain case, where one can use concretely efficient distributed FSS key generation protocols [25]. For our large-domain protocols, the cost of distributing the dealer is substantially higher, and we leave optimizing it to future work.

1.1.2 *A Barrier.*

The best online round complexity achieved by our protocols is 2 rounds. We complement this by providing evidence against the possibility of efficient protocols for secure sorting or selection with a single online round. We show a barrier by relating sorting/selection to a well-known open problem about FSS for conjunctions. This suggests that our 2-round solutions may be the best one could hope for without using either stronger cryptographic assumptions (such as LWE [24]), exponential preprocessing (via one-time truth tables [22, 28]), or a large number of parties [9].

1.1.3 *Reducing Computation via Compare-aggregate algorithms.*

We explore the benefits of relaxing the requirement of minimal rounds to a higher constant rounds for reducing the online computation (and offline communication) of comparison-based secure sorting/selection protocols from quadratic (in n) to sub-quadratic by building on the succinct aggregation feature of FSS. For this, we use an adaptation of a comparison-based sorting and selection model due to Valiant to optimize the computational cost of DCF-based ranking, $\binom{n}{2}$-CmpAgg. In Valiant's model, an algorithm proceeds in rounds, where each round selects pairs of elements that should be compared. The decision regarding which elements should be compared in each round can depend on the results of comparisons in previous rounds. This makes Valiant's model strictly more liberal than the traditional comparison network model [2], giving

rise to significantly simpler algorithms with better concrete efficiency [3, 13, 18].

To exploit the succinct aggregation feature of DCFs, we need to modify Valiant's model into what we call the "compare-aggregate" model. Here we represent the comparisons made in each round by a graph, where only the rank of each node relative to its neighbors in the graph is made public and can be used in the next rounds. (To avoid information leakage about the relative ranking of the original inputs, the protocols we obtain from compare-aggregate algorithms need to start with a secure shuffle.) Intuitively, when using a graph H to specify the pairs of DCF-based comparisons, the number of edges in H corresponds to offline communication and online computation, and the number of nodes with nonzero degree corresponds to online communication. We observe that many (but not all) existing sorting and selection algorithms in Valiant's model can be naturally adapted to our more restrictive compare-aggregate model. We further note that unlike the previous case, where we aimed to minimize the number of online rounds (Table 1), there is a big difference in complexity of sorting and selection. Indeed, while sorting inherently requires a total of $\Omega(n \log n)$ comparisons, selection can be done with $O(n)$ comparisons even when insisting on a constant number of rounds.

Using this compare-aggregate adaptation of known algorithms, we can reduce the quadratic computational cost of the comparison-based secure sorting/selection protocols (Protocol I and III from Table 1) at the price of requiring a higher (but still constant) number of rounds. See the full version for a summary. For instance, the comparison based secure sorting Protocol I (resp. selection Protocol III) from Table 1 requires $n^2\ell$ PRG calls in the online phase and an online communication of $4n\ell+2n\log n$ over 3 (resp. 2) online rounds. On the other hand, by leveraging compare-aggregate algorithms, we provide a sorting protocol that requires $2rn^{1+1/r}\ell$ PRG calls in the online phase and communicates $2n\ell(r+1)+2rn\log n$ bits over $2r+1$ rounds, for any constant r. We also provide a protocol for computing maximum (resp. median) that requires $2n\ell$ (resp. $\approx 15.6n\ell$) length-doubling PRG calls and $4n\ell + 2n\log n$ (resp. $6n\ell + 4n\log n$) bits over 9 rounds (details in the full version)

1.2 Related work

As mentioned earlier, this work considers standard 3-party protocols and a useful special class of such protocols known as (2+1) protocols, where one of the parties acts as a "dealer" that only needs to send correlated randomness to the other two parties in the offline phase and then remains completely silent during the online phase. While there are quite a few prior works on secure sorting (see [5, 7, 11, 34] and references therein), with varying tradeoffs, no prior work achieves asymptotically optimal online rounds (i.e. constant rounds) and online communication (i.e. linear in input size) *simultaneously* even in the more liberal 3-party model. Hence, for our baseline, the most relevant and competitive approach from the literature is the classic Garbled Circuit solution [3] which achieves optimal online rounds (exactly 2) with suboptimal online communication (i.e. input size times a computational security parameter) in the (2+1) party setting. We note that [34] proposes constant online round sorting protocols based on counting/bead sort and all-pairwise comparison. However, the limitation of counting/bead sort approach is that it doesn't extend to payloads (whereas all our protocols and baselines do) and the online communication scales *exponentially* with the bit-length ℓ. The limitation of all-pairwise comparison based approach is that the online communication scales *quadratically* with the the number of items n. As another baseline, we compare with [7] which performs 3 party radix sort with close to asymptotically optimal online communication (quasilinear in the input size) but a high number of rounds (linear in the length parameter ℓ). Other alternative approaches, such as those based on oblivious version of quicksort [5, 27] and GMW style batcher sort [8], neither have the constant online round feature of the Garbled Circuit solution nor the low communication feature of the secure radix sort protocol [7]. Hence, we restrict our comparison to the Garbled Circuit approach and secure radix sort approach [7] as their costs metrics represent the state of the art in online rounds or online communication for secure sorting and selection.

Functionality: $\mathcal{F}_{\text{sort}}$

Public Parameters: Number of elements n, input domain size L, abelian group $\mathbb{G}$ where $|\mathbb{G}| \geq L$, linear ordering $<: \mathbb{G} \times \mathbb{G} \to \{0,1\}$ where the L smallest elements in $\mathbb{G}$ are denoted by $[L] = \{0, \ldots, L-1\}$, payload group $\mathbb{H}$.
Inputs: Party P_b holds $\mathbf{x_b} \in \mathbb{G}^n$, $\mathbf{y_b} \in \mathbb{H}^n$, for $b \in \{0,1\}$ s.t. $\mathbf{x_0} + \mathbf{x_1} \in [L]^n$.
(1) Reconstruct $\mathbf{x} = \mathbf{x_0} + \mathbf{x_1}$ and $\mathbf{y} = \mathbf{y_0} + \mathbf{y_1}$.
(2) Sort $\mathbf{x}, \mathbf{y}$, using $\mathbf{x}$ as the keys and $\mathbf{y}$ as the corresponding payloads, to obtain $\mathbf{u}, \mathbf{v}$.
(3) Sample $\mathbf{u_0} \leftarrow \mathbb{G}^n$ and $\mathbf{v_0} \leftarrow \mathbb{H}^n$. Set $\mathbf{u_1} := \mathbf{u} - \mathbf{u_0}$ and $\mathbf{v_1} := \mathbf{v} - \mathbf{v_0}$.
Outputs: Party P_b receives $\mathbf{u_b}, \mathbf{v_b}$, for $b \in \{0,1\}$.

Figure 1: Ideal functionality for sorting with payload as the optional parameter highlighted in blue.

2 PRELIMINARIES

Let $\mathbb{G}$ be a well-ordered Abelian group with efficient comparison. We use $[n]$ to denote the set of integers $\{1, \ldots, n\}$. We use lowercase letter x to denote elements in $\mathbb{G}$ and bold letter $\mathbf{x}$ to denote a vector of elements in $\mathbb{G}$. We use $\pi : [n] \to [n]$ to denote a permutation on $[n]$. We use λ to denote a concrete computational security parameter (typically $\lambda = 128$ in practice) that serves as the length of a seed for a pseudorandom generator (PRG). We use $[\![x]\!]$ and $\langle x \rangle$ to denote the additive and replicated sharing respectively of private x.

2.1 Problem description and security model

[3]For fair comparison, we consider the implementation of Garbled circuit solution in a (2+1) party setting where the "dealer" party locally generates and sends the garbled circuit along with labels to the other two parties during the offline phase (see the full version for details). For the offline cost, we only consider concretely efficient solutions specifically Batcher sorting circuit [8], excluding ones based on the AKS sorting network [2].

Functionality: $\mathcal{F}_{\text{select}}$

Public Parameters: Number of elements n, input domain size L, ordered Abelian group $\mathbb{G}$ where $[L] \subseteq \mathbb{G}$, rank $k \in [n]$, payload group $\mathbb{H}$.
Inputs: Party P_b (for $b \in \{0,1\}$) holds $\mathbf{x_b} \in \mathbb{G}^n$, $\mathbf{y_b} \in \mathbb{H}^n$, s.t. $\mathbf{x_0} + \mathbf{x_1} \in [L]^n$.

(1) Reconstruct $\mathbf{x} = \mathbf{x_0} + \mathbf{x_1}$ and $\mathbf{y} = \mathbf{y_0} + \mathbf{y_1}$ shares over $\mathbb{G}$.
(2) Let u be the rank-k element in $\mathbf{x}$, and v be its payload.
(3) Sample $u_0 \leftarrow \mathbb{G}$ and $v_0 \leftarrow \mathbb{H}$. Set $u_1 := u - u_0$ and $v_1 := v - v_0$.

Outputs: Party P_b receives u_b, v_b, for $b \in \{0,1\}$.

Figure 2: Ideal functionality for selection with payload as the optional parameter highlighted in blue.

Consider a vector $\mathbf{a} = (a_0, \ldots, a_{n-1})$ of n (secret-shared) items, where each item is $a_i \in [L] \subseteq \mathbb{G}$. Given this, we have the following two problems of interest:

- Sorting: In this problem, the task is to output a new vector of n secret shared items, denoted by $\mathbf{d} = (d_0, \ldots, d_{n-1})$, where $\mathbf{d}$ is the sorted version of $\mathbf{a}$. ($\mathcal{F}_{\text{sort}}$ in Figure 1).
- k^{th} order statistic: In this problem, we are only interested in computing a single secret shared item d, where d is the k^{th} smallest value in $\mathbf{a}$, for $k \in [n]$. ($\mathcal{F}_{\text{select}}$ in Figure 2).

In this work, we are interested in minimizing the asymptotic and concrete complexity of secure protocols for the above problems that only make use of symmetric cryptography. We will largely focus on input group $\mathbb{G} = \mathbb{Z}_{L'}$ for some $L' \geq L$ with some generalizations.

We prove the computational security of our protocols against static, semi-honest adversaries corrupting a single party. Intuitively, semi-honest means that the adversary follows the protocol description faithfully but attempts to learn additional information. See, e.g., [15, 20] for the standard security definition.

(2+1) and 3 party models. We provide protocols in the *(2+1)-party* and *3-party* settings with a single corruption. The 2+1 setting corresponds to standard 2-party computation given access to correlated randomness from some distribution $\mathcal{D}$, formalized by the $\mathcal{F}_{\mathcal{D}}$-hybrid model for the ideal functionality which samples and outputs from $\mathcal{D}$. The functionality $\mathcal{F}_{\mathcal{D}}$ can be securely instantiated either directly by a trusted dealer (which may be viewed as a semi-honest-corrupted third party) or via a secure 2-party computation protocol. We will present protocols in this fashion between two online parties $\mathsf{P}_0, \mathsf{P}_1$ and the "offline" party P_2 (which will sometime be referred to as the dealer party).

In the case of 3-party protocols, the above functionalities $\mathcal{F}_{\text{sort}}$, $\mathcal{F}_{\text{select}}$ are extended directly, where just the first two parties provide inputs and receive outputs while the third party is an input-less party that acts as a "helper" party to aid the secure computation during the online phase. Note that this can be modified to a setting where all 3 parties have inputs and outputs with low overhead by simply having the helper party secret-share its input between the other parties in an initial round.

2.2 Function secret sharing

We follow the definition of function secret sharing (FSS) from [17]. A (2-party) FSS scheme for a function family $\mathcal{F}$ is a pair of efficient algorithms (Gen, Eval) with the following syntax:

- $\mathsf{Gen}(1^\lambda, f)$ is a *key generation algorithm* that given 1^λ and description of a function f outputs a pair of keys (k_0, k_1). We assume that f explicitly contains descriptions of input and output groups $\mathbb{G}_{\text{in}}, \mathbb{G}_{\text{out}}$.
- $\mathsf{Eval}(i, k_i, x)$ is an *evaluation algorithm* that given party index $i \in \{0,1\}$, key k_i, and $x \in \mathbb{G}_{\text{in}}$ outputs a group element $y_i \in \mathbb{G}_{\text{out}}$.

Correctness of the FSS scheme requires that for every $f \in \mathcal{F}$ and input $x \in \mathbb{G}_{\text{in}}$ in the domain of f the corresponding evaluated output shares satisfy $y_0 + y_1 = f(x)$. Security requires that a single key k_i reveals no information beyond a specified leakage $\mathsf{Leak}(f)$. Unless otherwise stated, we assume $\mathsf{Leak}(f) = (\mathbb{G}_{\text{in}}, \mathbb{G}_{\text{out}})$; i.e., a single key reveals just the input and output domains of the function.

DPF and DCF. We make use of two particularly useful special cases of FSS: Distributed Point Functions (DPF), and Distributed Comparison Functions (DCF).

DPFs correspond to FSS for the class of point functions, $f_{\alpha,\beta}$ for which $f_{\alpha,\beta}(\alpha) = \beta$ and $f_{\alpha,\beta}(x) = 0 \in \mathbb{G}_{\text{out}}$ for all $x \neq \alpha \in \mathbb{G}_{\text{in}}$. DPFs can be achieved from length-doubling PRGs, with key size roughly $\lambda k + m$ for $k = \lceil \log |\mathbb{G}_{\text{in}}| \rceil$ and $m = \lceil \log |\mathbb{G}_{\text{out}}| \rceil$, and where Eval invokes the PRG $(k + \log\lceil m/\lambda \rceil)$ times [17].

DCFs are FSS for the class of comparison functions $f^{<}_{\alpha,\beta} : \{0,1\}^k \to \mathbb{G}_{\text{out}}$, for which $f^{<}_{\alpha,\beta}(x) = \beta$ for $x < \alpha \in \{0,1\}^k$ and 0 for $x \geq \alpha$ (where ordering on $\{0,1\}^k$ is given by integer interpretation). DCFs can be achieved from length-quadrupling PRGs with key size roughly $k(\lambda + m)$ bits where Eval invokes the PRG roughly $k \log\lceil m/\lambda \rceil$ times [14]. We defer discussion of exact concrete costs to the full version.

Multiplicative DPF. A natural variant of DPF, considered in [1, 19], is (N, t)-multiplicative DPF, where $\mathbb{G}_{\text{out}}$ is the additive group of a commutative ring R, and the outputs of Eval on x are an (N, t)-multiplicative secret sharing of the desired point function $f_{\alpha,\beta}(x)$. That is, any t FSS shares reveal no information on $f_{\alpha,\beta}$ beyond $\mathsf{Leak}(f_{\alpha,\beta})$, and a polynomial of degree $\lceil N/t \rceil$ can be computed on the secret by just executing local computation on the shares.

A construction of $(3,1)$-mult-DPF from length-doubling PRGs was given in [1], with per-party key size roughly $2(k\lambda + \lceil 3 \log |R|/2\lambda \rceil)$ bits. The Eval algorithm invokes the PRG roughly $k + \lceil 3 \log |R|/2\lambda \rceil$ times. For any input x, the resulting output shares are *replicated* secret shares of $f_{\alpha,\beta}(x)$: each party $i \in [3]$ receives two shares $\langle z \rangle_i := (z_i, z_{i+1 \pmod 3})$ for which $z := z_0 + z_1 + z_2 = f_{\alpha,\beta}(x)$ (see the full version). We denote such replicated sharing of value z as $\langle z \rangle$. Given shares $\langle z \rangle, \langle w \rangle$ of values z, w, one can locally convert to additive shares of the product zw at low cost, denoted by $\mathsf{LocalMult}(\langle z \rangle_i, \langle w \rangle_i) = [\![zw]\!]_i$ (see, e.g., [19]). We defer the details of distributed key generation procedure of $(3,1)$-mult-DPF to the full version.

For each FSS scheme $\mathsf{fss} \in \{\mathsf{DPF}, \mathsf{DCF}, \mathsf{multDPF}\}$ over input group $\mathbb{G}_{\text{in}}$ and output group $\mathbb{G}_{\text{out}}$, we will use the notations $\mathsf{fss.KeySize}[\mathbb{G}_{\text{in}}, \mathbb{G}_{\text{out}}]$, $\mathsf{fss.DistGen}[\mathbb{G}_{\text{in}}, \mathbb{G}_{\text{out}}]$, $\mathsf{fss.Eval}[\mathbb{G}_{\text{in}}, \mathbb{G}_{\text{out}}]$, $\mathsf{fss.FullEval}[\mathbb{G}_{\text{in}}, \mathbb{G}_{\text{out}}]$ to abstract away the per party costs of key size (in bits),

communication cost of distributed key generation protocol (in bits), computation cost of Eval and full domain evaluation (in terms of length doubling PRG calls) respectively.

2.3 FSS gates

Boyle *et al.* [15] provided general-purpose transformations for obtaining efficient secure computation protocols in the preprocessing model via FSS schemes for corresponding function families.

Definition 2.1 (Offset function family & FSS gates)**.** Let $\mathcal{G} = \{g : \mathbb{G}_{\mathsf{in}} \to \mathbb{G}_{\mathsf{out}}\}$ be a computation gate, parameterized by input and output groups $\mathbb{G}_{\mathsf{in}}, \mathbb{G}_{\mathsf{out}}$ (and possibly additional values param). The *family of offset functions* $\hat{\mathcal{G}}$ of $\mathcal{G}$ is given by

$$\hat{\mathcal{G}} := \left\{ g^{[r_{\mathsf{in}}]} : \mathbb{G}_{\mathsf{in}} \to \mathbb{G}_{\mathsf{out}} \;\middle|\; \begin{array}{c} g : \mathbb{G}_{\mathsf{in}} \to \mathbb{G}_{\mathsf{out}} \in \mathcal{G}, \\ r_{\mathsf{in}} \in \mathbb{G}_{\mathsf{in}} \end{array} \right\}, \text{ where}$$

$$g^{[r_{\mathsf{in}}]}(x) := g(x - r_{\mathsf{in}}),$$

and $g^{[r_{\mathsf{in}}]}$ contains an explicit description of r_{in}.

An *FSS gate for* $\mathcal{G}$ denotes an FSS scheme for the corresponding offset family $\hat{\mathcal{G}}$, with $\mathsf{Leak}(g) = (\mathbb{G}_{\mathsf{in}}, \mathbb{G}_{\mathsf{out}}, \mathsf{param})$.

As shown in [15], an FSS gate for computation gate $\mathcal{G}$ implies an "online-optimal" protocol for converting a masked input x to a masked output $g(x)$ for $g \in \mathcal{G}$. Concretely, the online phase consists of one round in which each party sends a message of length $|g(x)|$. Alternatively, we can have a similar one-round protocol converting additively shared input to additively shared output, where here the message length is $|x|$. The offline storage corresponds to the FSS key size produced by Gen, and the online compute time corresponds to the computational cost of Eval.

We will make use of the FSS gate for unsigned integer comparison uCMP (from [14]). This function $g_{\mathsf{uCMP}}(x, y)$ from the family $\mathcal{G}_{\mathsf{uCMP}}$ parameterized by input group $\mathbb{Z}_N \times \mathbb{Z}_N$ and output group $\mathbb{Z}_n$ where the groups have a total ordering defined on them. It takes as input $(x, y) \in \mathbb{Z}_N \times \mathbb{Z}_N$, where $|x - y| < N/2$, and outputs $1 \in Z_N$ if $x > y$ and 0 otherwise. (In the case $|x - y| \geq N/2$, no promise is made.) This gate can be computed with a key size $1 \cdot \mathsf{DCF.keysize}[\mathbb{Z}_N, \mathbb{Z}_n] + 2n$ bits, and with evaluation cost $1 \cdot \mathsf{DCF.Eval}[\mathbb{Z}_N, \mathbb{Z}_n]$.

2.4 Secure shuffle

Some of our protocols will make use of a secure protocol for randomly *shuffling* the list of secret-shared elements. We will work with a slight variant of the usual shuffling functionality where, instead of outputting just a secret-shared shuffled list, it also outputs a *public masked* shuffled list to all online parties. More precisely, given a list of inputs $\mathbf{x} = (x_1, \ldots, x_n)$, the secure shuffling functionality additionally outputs $\pi(\mathbf{x}) + \mathbf{r}$ to all parties where π is a permutation and $\mathbf{r} = (r_1, \ldots, r_n)$ is set of n random private masks unknown to any single party. Since we construct $(2 + 1)$ and 3 party protocols in this paper, we will use two versions of the functionality. In the full version, we provide the formal descriptions of these functionalities along with a concrete protocol instantiation which is adapted from prior works [21]. The rationale behind this modification to the usual shuffling functionality is that we can use the public masked shuffled array $\pi(\mathbf{x}) + \mathbf{r}$ as inputs to different types of FSS gates without additional communication where the FSS gates have $\mathbf{r}$ as the secret parameter. This will become clearer later in Section 4.1.

3 FSS GATES FOR RANKING

Given a set of n distinct elements, $x_1, \ldots, x_n$, the rank of an element x_i is simply the number of elements $\{x_j\}_{j \in [n], j \neq i}$ s.t. $x_j < x_i$. In this sense, the maximum element has rank $n - 1$ whereas the minimum element has rank 0. When elements are not necessarily distinct, we need to consider the notion of *stable* rank. Given an ordered set $S = (x_1, \ldots, x_n)$, the *stable* rank of an element x_i is the number of elements $\{x_j\}_{j \in [n], j \neq i}$ s.t. $x_j < x_i$ plus the number of elements $\{x_j\}_{j < i, x_j = x_i}$. From now on, we let $\mathsf{Rank}(x_i)$ denote the stable rank of x_i among an ordered set of elements.

Mirroring the notation of $\mathcal{F}_{\mathsf{sort}}, \mathcal{F}_{\mathsf{select}}$, we define

$$\mathcal{G}_{\mathsf{Rank}} = \left\{ g_{\mathsf{Rank},n} : \mathbb{G}_{\mathsf{in}}^n \to \mathbb{G}_{\mathsf{out}}^n \right\}, \text{where}$$
$$g_{\mathsf{Rank},n}(x_1, \ldots, x_n) = (\mathsf{Rank}(x_1), \ldots, \mathsf{Rank}(x_n)),$$

to be the family of ranking functions parametrized by $n, L, \mathbb{G}_{\mathsf{in}}, \mathbb{G}_{\mathsf{out}}$, where: number n of elements to be ranked, input domain size L, input group $\mathbb{G}_{\mathsf{in}}$ with $[L] \subseteq \mathbb{G}_{\mathsf{in}}$ in which input elements are represented, and output group $\mathbb{G}_{\mathsf{out}}$ where ranks are represented. We assume that both $\mathbb{G}_{\mathsf{in}}$ and $\mathbb{G}_{\mathsf{out}}$ are endowed with an (efficiently computable) total order, and $|\mathbb{G}_{\mathsf{out}}| \geq n$. Our default choice of $\mathbb{G}_{\mathsf{in}}$ and $\mathbb{G}_{\mathsf{out}}$ in this work would be $\mathbb{Z}_L$ and $\mathbb{Z}_n$ respectively.

We next proceed to construct FSS gates for $\mathcal{G}_{\mathsf{Rank}}$. In Section 3.1, we demonstrate a 2-party construction with key size scaling quadratically in n. In Section 3.2, we show in the 3-party setting (with security against 1 party) how to leverage computation linear in the domain size to achieve key size *linear* in n.

3.1 Secure ranking via DCF

A natural way to compute the rank of an ordered set of n elements is by simply performing all pairwise comparisons. In the context of secure computation, this would involve running a secure comparison protocol between every pair of elements which would typically lead to $O(n^2)$ elements of online communication over multiple rounds [11]. Our simple yet powerful observation is that this cost can be reduced to a single online round and $O(n)$ elements of online communication using DCFs.

Main idea. In a bit more detail, given shares of inputs $(x_1, \ldots, x_n)$ and shares of $(r_1, \ldots, r_n)$, where each r_i is a random mask sampled during preprocessing, parties publicly reveal $(x'_1, \ldots, x'_n)$ where $x'_i = x_i + r_i$. This requires a single round and n elements of communication per party. Each such x'_i can be thought of as a masked version of the corresponding x_i. Then, using $O(n^2)$ preprocessed DCFs containing $r_i - r_j$ as the secret parameter for all $1 \leq i < j \leq n$, parties can locally compute an additive sharing of all the $O(n^2)$ pairwise comparison results. Moreover, since $\mathsf{Rank}(x_i)$ is a simple linear combination of the results of comparisons in which x_i is involved, parties can locally aggregate the shares of comparison results in order to produce the shares of ranks of each item x_i.

Formalization. We formalize the above idea by defining a special kind of gate called compare-aggregate gate denoted by $\mathcal{G}_{\mathsf{CmpAgg}}$. Intuitively, this gate will perform pairwise comparisons among elements and then aggregate the results. To make this idea more general, instead of restricting to all pairwise comparisons among n

FSS gate: Compare Aggregate

$\mathsf{Gen}^{\mathsf{CmpAgg}}_{m,H=([m],E)}(1^\lambda, r_{\mathsf{in}} = (r_1, \ldots, r_n))$:
(1) For all $e = (i,j) \in E : (k_0^e, k_1^e) \leftarrow \mathsf{Gen}_n^{\mathsf{uCMP}}(1^\lambda, r_i, r_j)$
(2) Output (K_0, K_1) where $K_b = \{k_b^e\}_{e \in E}$

$\mathsf{Eval}^{\mathsf{CmpAgg}}_{n,H=([n],E)}(b, K_b, x_1, \ldots, x_n)$:
(1) Parse K_b as $\{k_b^e\}_{e \in E}$
(2) $\forall e = (i,j) \in E : [\![\mathsf{gt}^e]\!] \leftarrow \mathsf{Eval}_n^{\mathsf{uCMP}}(b, k_b^e, x_i, x_j)$
(3) For all $i \in [n] : [\![z_i]\!] := \sum_{e=(j,i)\in E}([\![1]\!] - [\![\mathsf{gt}^e]\!]) + \sum_{e=(i,j)\in E}[\![\mathsf{gt}^e]\!]$.
(4) Output $([\![z_1]\!], \ldots, [\![z_n]\!])$.

Figure 3: FSS gate for $\mathcal{G}_{\mathsf{CmpAgg}}$; b refers to party id.

elements, we allow any arbitrary subset of $\binom{n}{2}$ possible comparisons to be specified as a parameter to the gate. Such a subset would be represented using an undirected graph H on the node set $[n]$. While the complete graph (clique), which represents all pairwise comparisons, captures the most natural use-case that we outlined above, we will see in Section 5 how other useful types of graphs can be leveraged in non-trivial ways to reduce some cost metrics.

We are now ready to formally define our gate $\mathcal{G}_{\mathsf{CmpAgg}}$. It is a family of functions $g_{\mathsf{CmpAgg},m,H} : \mathbb{G}_{\mathsf{in}}^m \to \mathbb{G}_{\mathsf{out}}^m$ parameterized by a value $m \in \mathbb{Z}^+$, an undirected graph $H = (V, E)$, where $V = [m]$ and $E \subseteq \binom{[m]}{2}$, input group $\mathbb{G}_{\mathsf{in}}$ and output group $\mathbb{G}_{\mathsf{out}}$. We assume that both $\mathbb{G}_{\mathsf{in}}$ and $\mathbb{G}_{\mathsf{out}}$ are endowed with a total order and $|\mathbb{G}_{\mathsf{out}}| \geq m$. Also, without loss of generality, we assume that $i < j$ for every $(i,j) \in E$. The function $g_{\mathsf{CmpAgg},m,H}$ is defined as follows.

$$g_{\mathsf{CmpAgg}}(x_1, \ldots, x_m) = (z_1, \ldots, z_m)$$
$$\text{where } z_i = \Big|\{x_j | (j,i) \in E \text{ or } (i,j) \in E, x_j < x_i\}\Big| + \Big|\{x_j | (j,i) \in E, x_j = x_i\}\Big|$$

Intuitively, z_i captures the number of neighbours of x_i, w.r.t the index graph H, which are smaller than x_i plus those which are equal to x_i and also have index less than i (in the expression above, we don't need to explicitly enforce this because we have assumed that for every $(i,j) \in E$, we have $i < j$). This definition bears similarity with the notion of stable rank that we defined earlier. Indeed, when H is a clique on m nodes, then $g_{\mathsf{CmpAgg},m,H}$ is identical to the ranking function $g_{\mathsf{Rank},m}$. In Figure 3, we provide the construction of a FSS gate for $\mathcal{G}_{\mathsf{CmpAgg}}$ which leverages the FSS gate for uCMP (Section 2.3) for performing comparison checks.

Lemma 3.1. *There exists a two-party FSS gate for $\mathcal{G}_{\mathsf{CmpAgg}}$ (Fig. 3) for parameters $m \in \mathbb{Z}^+, H = ([m], E), \mathbb{G}_{\mathsf{in}} = \mathbb{Z}_{L'}, \mathbb{G}_{\mathsf{out}} = Z_{m'}$ (for $L' \in \mathbb{Z}^+$ and $m' \geq m$), with key size $|E| \cdot (\mathsf{DCF.keysize}[\mathbb{Z}_{L'}, \mathbb{Z}_{m'}] + \lceil \log |\mathbb{G}_{\mathsf{out}}| \rceil)$ and eval cost $|E| \cdot \mathsf{DCF.Eval}[\mathbb{Z}_{L'}, \mathbb{Z}_{m'}]$.*

Proof Sketch. Security of the construction follows by the security of the underlying FSS gate for uCMP. Namely, one can use the simulator for the FSS scheme for uCMP to simulate the key for the FSS scheme for $\mathcal{G}_{\mathsf{CmpAgg}}$ for the corrupt party. Moreover, correctness follows from the correctness of the FSS scheme for the underlying uCMP gate together with the fact that it outputs additive shares of 0 or 1 over $\mathbb{Z}_{m'}$ which can be linearly combined to compute additive shares of the output of g_{CmpAgg} function. □

FSS gate: Multiplicative DPF ranking

$\mathsf{Gen}_n^{\mathsf{multDPFRank}}(1^\lambda, r_{\mathsf{in}} = (r_1, \ldots, r_n))$:
(1) Set $\ell' = \lceil \log_2 |\mathbb{G}_{\mathsf{in}}| \rceil$ and $L' = 2^{\ell'}$.
(2) For $i \in [n] : (k_0^i, k_1^i, k_2^i) \leftarrow \mathsf{Gen}_{\ell'}^{\mathsf{multDPF}}(1^\lambda, \mathsf{Bitwise}(-r_i))$
(3) Output (K_0, K_1, K_2) where $K_b = \{k_b^i\}_{i \in [n]}$.

$\mathsf{Eval}_n^{\mathsf{multDPFRank}}(b, K_b, x_1, \ldots, x_n)$:
(1) Parse K_b as $\{k_b^i\}_{i \in [n]}$.
(2) For all $i \in [n]$, $\langle \mathbf{u}_\mathbf{i}' \rangle := \mathsf{FullEval}_{\ell'}^{\mathsf{multDPF}}(1^\lambda, k_b^i)$.
(3) Set permutation map $\pi_i : \{0,1\}^{\ell'} \to \{0,1\}^{\ell'}$ where $\pi(\mathsf{Bitwise}(g)) = \mathsf{Bitwise}(g + x_i)$ for all $g \in \mathbb{G}$ and $\pi_i(j) = j$ otherwise.
(4) For all $i \in [n]$, $\langle \mathbf{u_i} \rangle = \pi_i(\langle \mathbf{u}_\mathbf{i}' \rangle)$.
(5) Compute $\langle \mathbf{v} \rangle := \langle \mathbf{u_1} \rangle + \ldots + \langle \mathbf{u_n} \rangle$.
(6) Compute $\langle \mathbf{w} \rangle$ where $\langle w_i \rangle = \langle v_i \rangle + \ldots + \langle v_n \rangle$ for $i \in [n]$.
(7) For all $i \in [n]$, compute $\langle \mathbf{y_i} \rangle = \langle \mathbf{u_1} \rangle + \ldots + \langle \mathbf{u_{i-1}} \rangle$
(8) For all $i \in [n]$, compute $[\![z_i]\!] := [\![n]\!] - (\langle \mathbf{y_i} \rangle - \langle \mathbf{w} \rangle) \cdot \langle \mathbf{u_i} \rangle$.
(9) Output $([\![z_1]\!], \ldots, [\![z_n]\!])$.

Figure 4: FSS gate for ranking using multiplicative DPF, b refers to party id.

The following corollary is an immediate consequence of Lemma 3.1 by taking the graph H to be the clique on n nodes. The gate class $\mathcal{G}_{\mathsf{CmpAgg}}$ for such a graph will be denoted by $\binom{n}{2}$-CmpAgg.

Corollary 3.1.1. *There exists a two-party FSS gate for $\mathcal{G}_{\mathsf{Rank}}$ for parameters $(n, L, \mathbb{G}_{\mathsf{in}} = Z_{L'}, \mathbb{G}_{\mathsf{out}} = \mathbb{Z}_n)$ for $L' \geq 2L$ and with key size $\binom{n}{2} \cdot \mathsf{DCF.keysize}[\mathbb{Z}_{L'}, \mathbb{Z}_n]$ and eval cost $\binom{n}{2} \cdot \mathsf{DCF.Eval}[\mathbb{Z}_{L'}, \mathbb{Z}_n]$.*

3.2 Secure ranking via multiplicative DPF

In the previous section, we developed an approach for ranking via pairwise comparisons. As pointed out in Corollary 3.1.1, a drawback of the previous comparison based approach for ranking is that it requires key size and computational cost proportional to $O(n^2)$ which makes it favorable only for small values of n. In this section, we will design an alternative technique, hereafter referred to as mult-DPF ranking, that will reduce the cost to $O(n)$. This comes at the expense of making the computation scale linearly (rather than logarithmically) with the input domain size $L = 2^\ell$, rendering it practical only for small values of ℓ (e.g. $\ell \leq 20$ bits). This caveat can be avoided by using mult-DPF ranking as a subroutine inside radix sort for sorting each digit/radix where the radix domain is small[4].

Main idea. Suppose parties want to rank n distinct items, $(x_1, \ldots, x_n)$, where each $x_i \in \mathcal{D} = [0, L)$ is additively shared. Then, given

[4]Specifically, mult-DPF ranking can replace Protocol 4.1 in [7] (SOTA on 3-party radix sort) with the advantage that it can handle digits of bit length $k > 1$. This reduces the total rounds of [7] by a factor of k while avoiding the exponential (in k) communication overhead of the previous protocol.

n random DPF keys, $k_1, \ldots, k_n$, on domain $\mathcal{D}$ (generated during the offline phase), the parties can convert each secret-shared x_i into a secret-sharing of the characteristic vector representation $\mathbf{u_i}$ of length L i.e. $\mathbf{u_i}$ is a vector containing 1 at location x_i and 0 elsewhere. Now, if we add the unit-vectors $u_1, \ldots, u_n$ entry-wise, we get a histogram of the items $(x_1, \ldots, x_n)$ over the domain $\mathcal{D}$. Let this aggregated vector be $\mathbf{v} = \mathbf{u_1} + \ldots + \mathbf{u_n}$. Now if we perform a postfix sum over $\mathbf{v}$ and let the resulting vector be $\mathbf{w}$, then the observation is that the k^{th} co-ordinate of $\mathbf{w}$ represents the number of items[5] that are $\geq k$. Hence, we get the following relationship:

$$\mathsf{Rank}(x_i) = n - \mathbf{u_i} \cdot \mathbf{w},$$

where $\cdot$ is an inner product.

Our next observation is that most of the aforementioned operations on unit-vectors, including entry-wise additions and postfix summation, can be securely and non-interactively performed by parties given entry-wise sharing of each $\mathbf{u_i}$ for the simple reason that these operations are linear. However, towards the end, parties need to securely perform an inner product operation which is non-linear. Here, we leverage the power of a new tool called multiplicative DPF (Section 2.2) which, as the name suggests, enables us to perform multiplication. More precisely, the outputs of such a DPF is shared in the form of a linear secret sharing scheme that enables a single multiplication operation non-interactively (besides the usual linear operations)[6].

Lastly, we note that the case of non-distinct elements in the input $(x_1, \ldots, x_n)$ can be handled using a slight modification. Let $\mathbf{y_i} = \sum_{j=1}^{i-1} u_i$. Then $\mathbf{y_i} \cdot \mathbf{u_i}$ denotes the number of elements x_k where $x_k = x_i$ and $k < i$. Hence, the stable rank of x_i will be captured by the following equation.

$$\mathsf{Rank}(x_i) = n - (\mathbf{y_i} - \mathbf{w}) \cdot \mathbf{u_i}$$

We thus obtain the following construction of FSS gates for $\mathcal{G}_{\mathsf{Rank}}$ in the 3-party setting based on multiplicative DPFs, described in full in Fig. 4.

Lemma 3.2. *There exists a 3-party FSS gate for $\mathcal{G}_{\mathsf{Rank}}$ (Fig. 4) for parameters $(n, L, \mathbb{G}_{\mathsf{in}}, \mathbb{G}_{\mathsf{out}} = \mathbb{Z}_n)$ with $[L] \subseteq \mathbb{G}_{\mathsf{in}}$ (where $\mathbb{G}_{\mathsf{in}}$ has efficiently computable total ordering and an efficient bitwise representation* $\mathsf{Bitwise} : \mathbb{G}_{\mathsf{in}} \to \{0,1\}^{\lceil \log |\mathbb{G}_{\mathsf{in}}| \rceil}$*), with:*

- *key size* $n \cdot \mathsf{multDPF.keysize}[\{0,1\}^{\lceil \log |\mathbb{G}_{\mathsf{in}}| \rceil}, \mathbb{Z}_n]$,
- *eval cost* $n \cdot \mathsf{multDPF.FullEval}[\{0,1\}^{\lceil \log |\mathbb{G}_{\mathsf{in}}| \rceil}, \mathbb{Z}_n]$.

4 MINIMIZING ONLINE ROUNDS

As we claimed in the introduction, a dividing line between our work and all of the prior works is that we achieve asymptotically optimal online rounds and communication. In this section, we will push that to the extreme and design secure sorting and selection protocols having just 2 or 3 online rounds with (per-party) online communication just twice the input size. A summary of the cost tradeoffs between our approaches was presented in the introduction (Table 1) and this section will dive into the details.

[5]This approach bears similarity to the ranking procedure implicit in counting sort and bead sort [6].

[6]Alternatively, we could perform the multiplication via the traditional Beaver-style technique using an additional round of communication. However, this would incur $O(L)$ bits of communication which we would like to avoid.

Modular approach. As evident from prior works [7], the high-level design principle of secure sorting protocols involves two key components: ranking and routing. In section 3, we explored ranking in depth; we now define routing. Given a set of ordered items $(x_1, \ldots, x_n)$ and their associated (stable) ranks $(\mathsf{Rank}_1, \ldots, \mathsf{Rank}_n)$, the process of routing simply involves reordering/permuting the items in the correct order to get $(y_1, \ldots, y_n)$ where $y_{\mathsf{Rank}_i} = x_i$. In the setting of secure computation, the list of items and their ranks would be additively shared between the parties and the goal would be to securely perform the routing so that parties end up with an additive sharing of the routed result $(y_1, \ldots, y_n)$.

Advantage of the modularity. As observed in prior works [7], this approach of splitting the process of sorting into ranking and routing has a nice advantage that we can efficiently handle payloads. More precisely, when we want to sort a list of items $((x_1, y_1), \ldots, (x_n, y_n))$, where x_i are the usual items which will be used as the keys and y_i is the corresponding attached payload, then the ranking subroutine only needs to be invoked on the keys and not on the payloads. Since secure ranking is usually the cryptographically heavier component, making its cost independent of the payload is a nice feature (which will be present in all the protocols that we construct in this paper).

It is easy to see that by combining (secure) ranking and (secure) routing subroutines, we can get a (secure) sorting scheme. For the easier problem of (secure) selection (Figure 2), it is sometimes possible to replace routing (which involves reordering *all* the elements) with something simpler that only outputs the element of a *specific* rank of interest. In the following subsections, we will see two different approaches for secure routing and combine it with the two secure ranking approaches (Section 3) to yield secure sorting and selection protocols.

4.1 Shuffle-based sorting and selection

A well-known solution for secure routing is the shuffle-then-reveal paradigm [5, 7, 11, 27]. The idea is that by pre-shuffling the secret-shared input $(x_1, \ldots, x_n)$ before ranking the items, it becomes safe to reveal the ranks and then perform the routing in the clear. This of course requires a secure shuffling protocol in the first place which takes secret-shared input and produces secret-shared shuffled output. However, secure shuffling is an easier and much more well-studied problem for which there are highly efficient protocols both in the 2-party and 3-party setting (see [21, 30] and references therein).

As mentioned in Section 2.4, we will work with a shuffling functionality which takes as input secret-shares of the list from two parties and outputs a *public masked* shuffled list to all online parties (in addition to outputting the shares of shuffled list to the two input parties as usual). More precisely, given a list of inputs $\mathbf{x} = (x_1, \ldots, x_n)$, the secure shuffling functionality outputs $\mathbf{y} = \pi(\mathbf{x}) + \mathbf{r}$ to all online parties where π is any permutation and $\mathbf{r} = (r_1, \ldots, r_n)$ is set of n randomly sampled masks. The rationale behind this modification to the usual shuffling functionality is that we can use the public masked shuffled version $\mathbf{y}$ as inputs to different kinds of FSS gates without any additional communication where the FSS gates have $\mathbf{r}$ as the secret parameter. Specifically, in our use-case for sorting and selection, we will invoke an FSS gate for ranking $\mathcal{G}_{\mathsf{Rank}}$. Combining

the two ranking protocols presented in the previous section along with the shuffle-based sorting gives us the following theorem:

Theorem 4.1. *There exists protocols which securely realize $\mathcal{F}_{\mathsf{sort}}$ with 3 online rounds and $p \in \mathbb{N}$ bit payloads. It has the following efficiency features:*

- *(Compare-Aggregate ranking, 2 + 1 parties): For $\mathbb{G} = \mathbb{Z}_{L'}$ where $L' \geq 2L$ and $\ell' = \lceil \log L' \rceil$:*
  - *Online communication: $4n(\ell' + p) + 2n \lceil \log n \rceil$ bits total across both online parties.*
  - *Online computation: $2\binom{n}{2} \cdot \mathsf{DCF.Eval}[\mathbb{G}, \mathbb{Z}_n]$ across both online parties.*
  - *Offline communication: $6n(\ell'+p)+4n \lceil \log n \rceil + 2\binom{n}{2} \cdot (\mathsf{DCF.KeySize}[\mathbb{G}, \mathbb{Z}_n] + \lceil \log n \rceil)$ from offline party to the two online parties.*
- *(mult-DPF ranking, 3 parties): For $\mathbb{G}$ s.t. $|\mathbb{G}| = L'$ and $\ell' = \lceil \log L' \rceil$:*
  - *Online communication: $8n(\ell' + p) + 4n \lceil \log n \rceil$ bits across all three online parties.*
  - *Online computation: $3n \cdot \mathsf{multDPF.FullEval}[\mathbb{G}, \mathbb{Z}_n]$ across all three online parties.*
  - *Offline communication: $6n(\ell' + p) + 4n \lceil \log n \rceil + 3n \cdot \mathsf{multDPF.DistGen}[\mathbb{G}, \mathbb{Z}_n]$ bits across all three offline parties.*

In particular, using the DCF *construction from [14] and* multDPF *construction from [1], we get the concrete costs as shown in Table 1 for Protocol I and Protocol II.*

Proof. We refer the readers to the full version. □

4.2 Secure routing via standard DPF

We next describe an alternative method for obtaining secure protocols for $\mathcal{F}_{\mathsf{sort}}$ and $\mathcal{F}_{\mathsf{select}}$. In the place of the shuffle-then-reveal paradigm, we instead construct a form of DPF-based routing, in which the (secret-shared) rank values are never made public.

We first describe a modular version of the DPF routing procedure, which enables converting from a vector of n secret shared rank values (plus key and payloads) to secret shares of the rank-ordered payloads, with 2 rounds of linear-size communication. (In fact, this can be viewed as building an FSS gate for a corresponding routing gate.) Combining this routing procedure together with the FSS gates for $\mathcal{G}_{\mathsf{Rank}}$ from Section 3, this yields a 3-round protocol for $\mathcal{F}_{\mathsf{select}}$ and $\mathcal{F}_{\mathsf{sort}}$. We then show an optimization that enables collapsing one round in the combined protocol. Our final resulting protocols securely instantiate $\mathcal{F}_{\mathsf{sort}}$ (or $\mathcal{F}_{\mathsf{select}}$) in total *two rounds* in both the 2+1 party setting and the 3-party setting.

DPF routing: modular version. We present the procedure for k^{th} order statistic selection; full sorting extends in a straightforward manner by repeating for each k (note that much of the preprocessing material can be reused for all k). Consider the parties beginning with secret shares of the vector of n rank values $(y_1, \ldots y_n) \in \mathbb{Z}_n^n$, each with a given associated secret shared payload value $z_i \in \mathbb{G}_{\mathsf{payload}}$. We assume $\mathbb{G}_{\mathsf{payload}}$ has a further multiplicative structure.

As part of the preprocessing, the parties will receive (for $i \in [n]$):

(1) Additive secret shares of n random masks $r_i \in \mathbb{Z}_n$.
(2) n DPF keys, for $f_{\alpha_i,\beta_i} : \mathbb{Z}_n \to \mathbb{G}_{\mathsf{payload}}$, where $\alpha_i = r_i \in \mathbb{Z}_n$ and $\beta_i = 1 \in \mathbb{G}_{\mathsf{payload}}$.
(3) n Beaver multiplication triples, i.e. additive shares of random $(a_i, b_i, c_i = a_i b_i) \in \mathbb{G}_{\mathsf{payload}}^3$.

In **Round 1** of communication, the parties reconstruct the **r**-masked values of the ranks, by sending their share of y_i masked by their share of r_i, for each $i \in [n]$. (Communication $n \log n$ bits/party.) Denote the resulting reconstructed masked values by $\hat{y}_i$.

A given masked rank value $\hat{y}_i$ corresponds to plaintext rank $k \in [n]$ iff $\hat{y}_i - r_i = k$: equivalently, iff $f_{\alpha_i,\beta_i}(\hat{y}_i - k) = 1$. This enables a direct procedure for obtaining secret shares of a $1 \times n$ indicator vector in which the i-th position has a $1 \in \mathbb{G}_{\mathsf{payload}}$ exactly in position k if y_i is the k-th ranked element (and $0 \in \mathbb{G}_{\mathsf{payload}}$ otherwise), where each position i requires a single evaluation of the ith DPF key on input $(\hat{y}_i - k)$.

In the final phase, the parties will perform a secure computation protocol for evaluating the inner product of this secret shared indicator vector with the secret shared vector of input payloads, by use of the n Beaver multiplication triples.

Concretely, in **Round 2** of communication, the parties will reconstruct the **a**-masked payload vector and the **b**-masked indicator vector (where **a**, **b** are from the multiplication triples). (Communication $2n\lceil \log |\mathbb{G}_{\mathsf{payload}}| \rceil$ bits/party.) Each party can then use the remaining portion **c** of the multiplication triple together with the communicated values to locally compute additive secret shares of the degree-2 inner product. The result is precisely additive secret shares of the kth ranked payload over $\mathbb{G}_{\mathsf{payload}}$.

Round-compressed ranking + routing. We now describe how to save one round of communication when combining the above-described DPF-based routing procedure together in conjunction with a protocol for *obtaining* secret shares of the ranks of n original input values, as constructed in Section 3. The high-level idea will be to remove Round 2 of the above interaction, instead pushing the corresponding communication up into prior rounds.

Recall that Round 2 communication consists of two components: (shares of) the **a**-masked payload vector and **b**-masked indicator vector. First observe that the **a**-masked payload vector communication can directly be pushed to a prior round, as these values are known at the start of the protocol. In contrast, the **b**-masked indicator vector requires the parties to first compute the corresponding secret shares, which is the result of the previous rounds of communication. We will need to find a new way to effectively perform this inner product.

Note that no interaction is required to compute secret shares of an inner product between an additively shared vector with a *public* vector. Of course, we cannot simply make the payload vector public, as these values are secret. However, if $\mathbb{G}_{\mathsf{payload}}$ is a *field*, and the plaintext payloads are known to be nonzero,[7] then it would be safe to reveal the payload vectors scaled by *multiplicative masks*: i.e., $\tilde{z}_i = z_i s_i$ for random nonzero mask $s_i \leftarrow \mathbb{G}_{\mathsf{payload}}^*$.

This suggests the following modified protocol. In parallel to computing the secret shared vector of ranks, the parties will perform a secure (component-wise) multiplication of the secret shared payload vector $\mathbf{z}$ with a secret shared multiplicative mask vector $\mathbf{s}$, and publicly reveal the masked vector $\tilde{\mathbf{z}}$. The n DPFs in the preprocessing will be modified to no longer have payload $\beta_i = 1$, but rather

[7] The protocols in this work are agnostic to relations between payload values. As such, one can enforce nonzero encodings of payload values by, e.g., adding a fixed constant to all payloads, and then removing this constant at the conclusion of the protocol execution.

$\beta_i = s_i^{-1} \in \mathbb{G}^*_{\mathsf{payload}}$. After the additively masked vector $\hat{\mathbf{y}}$ of rank values is made public (as before), the parties can similarly compute secret shares of the corresponding kth indicator vector. *But*, if the vector previously had 1 in the ith position, then the modified (secret shared) vector will have in position i the value s_i^{-1}. The inner product of this resulting secret shared vector with the public vector $\tilde{\mathbf{z}}$ will thus pull out exactly $\tilde{z}_i s_i^{-1} = (z_i s_i) s_i^{-1} = z_i$, as desired.

Remark. In order to extend to the full sorting scenario, the parties can use the DPFs to locally compute additive shares of an $n \times n$ *permutation* matrix of 0s and s_i^{-1}s over $\mathbb{G}_{\mathsf{payload}}$. Note that this does not require further offline or online *communication.* (In particular, shares of position (i, k) in the permutation matrix are computed reusing the *same* n DPFs by means of $f_{\alpha_i,\beta_i}(\hat{y}_i - k)$.) The additional computation cost will indeed be dominated by evaluating each DPF at n sequential input values instead of at a single input. While this cost is ultimately quadratic in n, the sequential structure of inputs enables useful concrete optimizations.

We present the complete protocol structure for achieving $\mathcal{F}_{\mathsf{select}}$ (details in the full version) for the case of 2+1 parties. See above the discussion for how to extend from $\mathcal{F}_{\mathsf{select}}$ to $\mathcal{F}_{\mathsf{sort}}$. The protocol relies on an FSS gate for $\mathcal{G}_{\mathsf{Rank}}$, as obtained in Section 3. The case of 3 parties is directly analogous, but where the offline phase is jointly computed via a secure computation across the parties (instead of generated directly by a semi-honest dealer party). This cost is dominated by the expense of securely generating DPF keys and multiplicative DPF keys for secret shared values; in both cases, a version of the protocol due to Doerner and shelat [25] provides competitive costs black box in the underlying pseudorandom generator for relevant (mult-)DPF domain sizes. We refer the reader to the full version for detailed discussion.

Combining our (2+1) party protocol for $\mathcal{F}_{\mathsf{select}}$ together with the corresponding (2+1) party and 3-party FSS gate instantiations for $\mathcal{G}_{\mathsf{Rank}}$ in Corollary 3.1.1 and Lemma 3.2, we obtain the following theorem:

Theorem 4.2. *There exist protocols which securely realize $\mathcal{F}_{\mathsf{select}}$ when $\mathbb{H}$ is a field, with 2 online rounds, and the following features, where $\ell' = \lceil \log L' \rceil$ and $(\ell' + p) = \lceil \log |\mathbb{H}| \rceil$ (where $\mathbb{H}$ is assumed to encode both key and payload):*

- *(Compare-Aggregate ranking, 2+1 parties):*
  - *Online communication: $6n\ell' + 2n \log n + 4np$ bits across both online parties*[8].
  - *Online computation: $2\binom{n}{2}\cdot\mathsf{DCF.Eval}[\mathbb{Z}_{L'}, \mathbb{Z}_n] + 2n\cdot\mathsf{DPF.Eval}[\mathbb{Z}_n, \mathbb{H}]$ across both online parties.*
  - *Offline communication: Dominated by $2\binom{n}{2}\cdot\mathsf{DCF.keysize}[\mathbb{Z}_{L'}, \mathbb{Z}_n] + 2n \cdot \mathsf{DPF.keysize}[\mathbb{Z}_n, \mathbb{H}] + 8np$ bits from offline party to the two online parties.*
- *(mult-DPF ranking, 3 parties):*
  - *Online communication: $8n\ell' + 4n \log n + 4np$ bits across all three online parties.*
  - *Online computation: $3n \cdot \mathsf{multDPF.FullEval}[\{0,1\}^{\ell'}, \mathbb{Z}_n] + 2n \cdot \mathsf{DPF.Eval}[\mathbb{Z}_n, \mathbb{H}]$ across all three online parties.*
  - *Offline communication: Dominated by $3n\cdot\mathsf{multDPF.DistGen}[\{0,1\}^{\ell'}, \mathbb{Z}_n] + 2n \cdot \mathsf{DPF.KeySize}[\mathbb{Z}_n, \mathbb{H}]$ bits across all three offline parties.*

The protocols can be extended to securely realize $\mathcal{F}_{\mathsf{sort}}$ with additional online computation $\mathsf{DPF.FullEval}[\mathbb{Z}_n, \mathbb{H}]$ in the place of $\mathsf{DPF.Eval}[\mathbb{Z}_n, \mathbb{H}]$.

In particular, using the DCF *construction from [14],* DPF *construction from [17] and* multDPF *construction from [1], we get the concrete costs as shown in Table 1 for Protocol III and Protocol IV.*

4.3 Barrier toward one-round solutions

The resulting protocols from the previous subsections require 2 or 3 online rounds. We partially complement this by showing a barrier toward obtaining an efficient *one-round* protocol for sorting or selection by means of a corresponding FSS gate.

More concretely, we show that an FSS gate for the Minimum computation on n inputs from a domain of size $L \geq 3$ implies the existence of an FSS scheme for the class of conjunctions on n-bit inputs with comparable key size, where the class of conjunctions consists of all functions of the form $f_S = \wedge_{i \in S} x_i$. Since FSS for conjunctions from symmetric-key primitives (with key size polynomial in n and any constant number of parties) remains a major open problem in the field, we interpret this as a barrier toward obtaining FSS gates for Minimum from symmetric-key primitives. See, e.g., [14] for further discussion within the context of a similar barrier result. Finally, since an FSS gate for Minimum is implied by an FSS gate for sorting by just restricting the output to the first entry, the barrier extends to the latter case as well.

Theorem 4.3 (Barrier: FSS Gate for Minimum). *Let $L \geq 3$, let $\mathbb{G} = \mathbb{Z}_{L'}$ for $L' \geq L$, and let $g : \mathbb{G}^n \to \mathbb{G}$ be the Minimum function with input domain of size L. Namely, for $x \in [L]^n$ (viewed as a vector over $\mathbb{G}$), $g(x)$ outputs the smallest value of an entry of x. Suppose there is an FSS scheme for the offset class $g^{[r_{\mathsf{in}}]} : \mathbb{G}^n \to \mathbb{G}$ with key size s. Then there is an FSS scheme for the class of conjunctions $f_S = \wedge_{i \in S} x_i$ with the same key size.*

Proof. By definition of an offset class, we have $g^{[r_{\mathsf{in}}]}(x) = \mathsf{Minimum}(x - r_{\mathsf{in}})$. Given a nonempty set $S \subseteq [n]$ defining a conjunction function f_S, consider the $r_{\mathsf{in}} \in \mathbb{G}^n$ whose ith entry is 0 if $i \in S$ and -1 otherwise. It is easy to verify that $g^{[r_{\mathsf{in}}]}(x) = f_S(x)$ for all $x \in \{0,1\}^n$. Thus, FSS for conjunctions can be embedded in FSS for the offset class of g, from which the theorem follows. □

Remark. The requirement that $L \geq 3$ is necessary: with $L = 2$, the minimum of n inputs is equivalent to the conjunction of *all* inputs, whose offset class admits an efficient FSS (via a DPF).

We further remark that the barrier does not provide implications for small, constant n, for which FSS for conjunctions can be achieved (with key size exponential in n). Indeed, for example, FSS gate for Maximum of 2 elements can be obtained via a single DCF.

Finally, we note that while we assumed that inputs and outputs are shared over a cyclic group $\mathbb{G}$, a similar barrier holds even for XOR-sharing a bit representation of the inputs, namely $\mathbb{G} = \mathbb{Z}_2^k$.

[8] This can be optimized to $4n\ell' + 2n \log n + 4np$ by using a common mask for the beaver triple and the ranking gate.

5 APPLYING COMPARISON-BASED ALGORITHMS

Since the focus of this work is on improving the round and communication complexity of secure sorting and selection, we explore the algorithm literature which is conducive to that objective. To that end, in this work we identify that the most relevant model is the parallel comparison model introduced by Valiant [32]. Valiant's model is aimed to reduce the $O(n^2)$ computational cost of comparison based ranking (see Protocols I, III in Table 1) while maintaining constant round complexity. At a high level, algorithms in Valiant's model perform multiple (typically constant) rounds of comparisons in a smart fashion to reduce the total number of comparisons to sub-quadratic. Although one could directly compile an algorithm in Valiant's model into a secure protocol simply by performing the comparisons securely via FSS and revealing *all* the comparison results (after a shuffling step), this approach would result in protocols whose *communication cost* also scales with the number of comparisons. Our main insight is that we can improve the communication cost by aggregating the results of comparisons, only revealing this aggregate information (referred to as "local rank" in Section 5.2). We formalize this methodology by introducing a new intermediate algorithmic model, called the Compare-Aggregate model, which leads to secure protocols where the communication cost scales only with the number of elements involved in comparisons rather than the total number of comparisons. Since each element is usually involved in multiple comparisons, this approach yields significantly better asymptotic and/or concrete communication cost.

In Section 5.1, we describe Valiant's parallel comparison model along with some useful complexity metrics. We then introduce our novel Compare-Aggregate model in Section 5.2. Following that, in Section 5.3 we describe how Valiant's model algorithms can be cast into our Compare-Aggregate model. Finally, in Section 5.4 we present a compiler from (insecure) algorithms in Compare-Aggregate model to secure protocols in the $(2+1)$ party model.

5.1 Valiant's comparison model

Given an input $\mathbf{x} = (x_1, \ldots, x_n)$ consisting of n distinct items with a well-defined total ordering, a sorting algorithm $\mathcal{A}$ in Valiant's model proceeds in iterations[9] where, in each iteration, it performs a set of pairwise comparisons between the items. These pairwise comparisons are specified using an undirected graph $H = (V, E)$ where $V \subseteq [n]$, n denotes the number of items in the input, and each edge $e \in E$ denotes the index of items to be compared. Crucially, the graph H_i at iteration i can depend on the results of comparisons from all the previous iterations.

Given a fixed number of iterations k and an arbitrary set of inputs $\mathbf{x} = (x_1, \ldots, x_n)$, a central objective of sorting and selection algorithms in the Valiant's model is to minimize the total number of comparisons (as a function of n and k) across all iterations. In the graph theoretic viewpoint mentioned earlier, this corresponds the total number of edges in the comparison graphs across all the iterations. Let's say $H_1, \ldots, H_k$ are the series of comparison graphs invoked by the sorting algorithm $\mathcal{A}$, where each $H_i = (V_i, E_i)$, $V_i \subseteq [n], E_i \subseteq \binom{[n]}{2}$. Then, we can define the "edge complexity" of $\mathcal{A}$, denoted by $e_{\mathcal{A}}(n, k)$ as the following:

$$e_{\mathcal{A}}(n, k) = |E_1| + \ldots + |E_k|$$

For our purpose, we will also separately define the "node complexity" of $\mathcal{A}$ to capture the total number of nodes in the comparison graphs across all iterations. More formally, the node complexity will be denoted by $v_{\mathcal{A}}(n, k)$ and defined as follows:

$$v_{\mathcal{A}}(n, k) = |V_1| + \ldots + |V_k|$$

If the algorithm is randomized, the total number of edges (resp. nodes) is a random variable with expected value $e_{\mathcal{A}}(n, k)$ (resp. $v_{\mathcal{A}}(n, k)$). We remark that while the edge complexity captures the comparison complexity of sorting algorithms and is well-known in the literature, the node complexity is a (natural) measure we introduce in this work. While this doesn't seem to be a useful measure in the algorithms literature, it will turn out to be very crucial when we design the "secure" versions of these algorithms. Looking ahead, the node complexity will control the online communication cost whereas the edge complexity will control the online computation and offline communication cost of our secure protocols.

5.2 Compare-and-Aggregate (CA) model

As mentioned earlier, in principle one could directly compile an algorithm in Valiant's model into a secure protocol simply by performing the comparisons securely via FSS and revealing *all* the comparison results (after a shuffling step). This would yield protocols whose communication cost is proportional to the edge complexity. In this section, we introduce a new "Compare-and-Aggregate" model which leads to secure protocols in which the communication cost is proportional only to the node complexity (which is typically much smaller than the edge complexity).

Our "Compare-and-Aggregate" model can be viewed as a slight variant of Valiant's model where, for every comparison graph $H = (V, E)$ specified by the algorithm $\mathcal{A}$, instead of learning the results of all the comparisons specified by the edge set E, the algorithm only learns the *local rank* of each item indexed by V w.r.t to its neighbours in H. More formally, the local rank of an item x_i, where $i \in V$, is denoted by $\mathsf{LRank}(x_i)$ and is defined as follows:

$$\mathsf{LRank}(x_i) = \left|\{x_j | (i, j) \in E, x_i > x_j\}\right|$$

The name "Compare-and-Aggregate" is inspired from the fact that $\mathcal{A}$ only learns an aggregate information of the pairwise comparison results which, in our use-case, is the local rank. At this point, an astute reader might notice similarity between this model and the way we defined the compare aggregate gate $\mathcal{G}_{\mathsf{CmpAgg}}$ earlier in Section 3.1. Indeed, as we will see soon in Section 5.4, this particular way of defining the model enables communication efficient compilation of sorting/selection algorithms to secure protocols via FSS. We note that this model might be useful for problems beyond sorting and selection, such as clustering and convex hull computations, which rely on comparisons as a basic operation.

We conclude this subsection by noting that for non-distinct item case, the definitions for the comparison graph $H = (V, E)$ and the local rank function LRank need to be modified slightly. In particular, we enforce, without loss of generality, that V is an *ordered* set of nodes and for every edge $(i, j) \in E$ we have $i < j$. Furthermore,

[9] In the algorithm literature, the term "rounds" is used instead of "iterations". We introduce the term "iterations" to disambiguate it from "rounds" in secure protocols.

the local rank function will return the local *stable* rank, which is formally defined as follows:

$$\mathsf{LRank}(x_i) = \left|\{x_j | (i,j) \in E, x_i > x_j\}\right| + \left|\{x_j | (j,i) \in E, x_j \leq x_i\}\right|$$

5.3 Transforming algorithms in Valiant's model to CA model

In this section, we will show that well-identified sorting and selection algorithms which have appeared in the literature in the Valiant's model can actually be cast in our more restrictive Compare-and-Aggregate model without any overhead in the edge and node complexity. More formally, for an algorithm $\mathcal{A}$ in the Valiant's model, we will obtain an equivalent representation $\mathcal{A}'$ in the Compare-Aggregate model s.t. $e_{\mathcal{A}'}(n,k) = e_{\mathcal{A}}(n,k)$ and $v_{\mathcal{A}'}(n,k) = v_{\mathcal{A}}(n,k)$. Note that it is not immediately obvious how to do this as the Compare-Aggregate model severely restricts the information learnt by the algorithm. In fact, it is not clear whether such a transformation is possible in general; we manage to do so only for a handful albeit interesting and practically efficient algorithms. We refer the readers to the full version for more discussion related to this aspect.

Illustrative example. Having said that, we provide an example of the transformation here for Alon et. al. [4] randomized sorting algorithm (hereafter referred as AAV86). This algorithm has the best known edge-complexity for sorting in the Valiant's model. We describe its Valiant's model version in Algorithm 1 and its transformed version in the Compare-Aggregate model in Algorithm 2 with the modified parts highlighted in blue. At a high-level, Algorithm 1 is very similar to the well-known randomized quicksort with the difference being, instead of selecting a single pivot, we select multiple pivots at each of the k iterations (Lines 1 and 2). Then, as usual, the non-pivot elements are compared with the pivot elements and, additionally, all the pivot elements are compared among themselves (Lines 3 and 4). After deriving the comparison results, the non-pivot elements are split into disjoint subsets and recursively sorted (Lines 5 and 6).

Main observation. Our crucial observation is that for Algorithm 1 to work correctly, it is actually an overkill to reveal the results of *all* the comparisons (Line 3 in Algorithm 1). Indeed, in order to partition the non-pivot items correctly (Line 4 in Algorithm 1), all we need is to reveal the rank of the non-pivot items w.r.t. the pivot items along with the rank of pivot items w.r.t. each other. This information is precisely captured by the notion of local rank which we defined in Section 5.2. With that observation, we can easily get the transformed algorithm in our Compare-Aggregate model simply by substituting Lines 4 and 5 in Algorithm 1 to Lines 4 and 5 in Algorithm 2. Note that this transformation doesn't modify the control flow of the algorithm at all, thus preserving the edge and node complexity of the original algorithm.

Transforming other algorithms. In an effort to have practically efficient Compare-Aggregate algorithms, we did a thorough literature survey of the sorting and selection algorithms in the Valiant's model. These included algorithms for maximum, median/selection, sorting and sorted top-k. Besides making the algorithm steps and comparison graphs explicit, we made an effort to analyze the concrete complexity of the algorithms both analytically and empirically.

Algorithm 1: AAV86 [4] sorting in the Valiant's model

Parameters: Number of iterations k, number of items n.
Input: An ordered set $\mathbf{x} = (x_1, \ldots, x_n)$
Output: An ordered sorted set $\mathbf{y} = (y_1, \ldots, y_n)$

1 Let $p = \lceil n^{1/k} \rceil$.
2 Sample uniformly (without replacement) a set $P \subseteq [n]$ s.t. $|P| = p - 1$.
3 Let $H = (V, E)$ be a undirected graph where $V = [n] = A \cup B$ s.t. $A = [n] \setminus P$ and $B = P$. Moreover, (A, B) form a complete bipartite graph and B forms a clique.
4 Let $\{c_e\}_{e \in E} :=$ Compare$(\mathbf{x}, H)$, be the comparison results.
5 Let $\mathbf{x_A}$ denote the items in $\mathbf{x}$ indexed by A. Based on $\{c_e\}_{e \in E}$, partition $\mathbf{x_A}$ into p disjoint blocks, $\mathbf{x_A^1}, \ldots, \mathbf{x_A^p}$ s.t. all elements in $\mathbf{x_A^i} \leq \mathbf{x_A^j}$ for $i < j$.
6 For each $i \in [p]$: if $k > 1$, sort $\mathbf{x_A^i}$ recursively using iteration parameter $k - 1$ and number of items $|\mathbf{x_A^i}|$.

Algorithm 2: AAV86 [4] sorting in the Compare-Aggregate model

Parameters: Number of iterations k, number of items n.
Input: An ordered set $\mathbf{x} = (x_1, \ldots, x_n)$
Output: An ordered sorted set $\mathbf{y} = (y_1, \ldots, y_n)$

1 Let $p = \lceil n^{1/k} \rceil$.
2 Sample uniformly (without replacement) a set $P \subseteq [n]$ s.t. $|P| = p - 1$.
3 Let $H = (V, E)$ be a undirected graph where $V = [n] = A \cup B$ s.t. $A = [n] \setminus P$ and $B = P$. Moreover, (A, B) form a complete bipartite graph and B forms a clique.
4 Let $\{\mathsf{LRank}_v\}_{v \in V} :=$ Compare-Aggregate$(\mathbf{x}, H)$, be the local rank results.
5 Let $\mathbf{x_A}$ denote the items in $\mathbf{x}$ indexed by A. Based on $\{\mathsf{LRank}_v\}_{v \in V}$, partition $\mathbf{x_A}$ into p disjoint blocks, $\mathbf{x_A^1}, \ldots, \mathbf{x_A^p}$ s.t. all elements in $\mathbf{x_A^i} \leq \mathbf{x_A^j}$ for $i < j$.
6 For each $i \in [p]$: if $k > 1$, sort $\mathbf{x_A^i}$ recursively using iteration parameter $k - 1$ and number of items $|\mathbf{x_A^i}|$.

Such an analysis is absent from most of the papers which only include the asymptotic costs. In the full version we provide a summary of our survey.

5.4 From insecure CA algorithms to secure protocols via FSS

High-level idea. In this section, we will show a simple compiler from algorithms in the Compare-Aggregate model to secure protocols in the $(2+1)$ party model. This compiler will be very similar to the shuffle-then-reveal approach that we took in Section 4.1 with the difference being, instead of computing the full ranking of all elements using all pairwise comparisons, we will only compute the *local rank* of elements iteration-wise as prescribed by the underlying Compare-Aggregate algorithm.

Details. We first recall our definition of compare-aggregate gate $\mathcal{G}_{\mathsf{CmpAgg}}$ from Section 3.1. This gate, parameterized by a graph H, computes the function g_{CmpAgg} on the a set of elements. As readers might note now, this function is exactly the LRank function that we defined in Section 5.2. Hence, the idea behind the compiler is simply to invoke a FSS gate for $\mathcal{G}_{\mathsf{CmpAgg}}$ in every iteration of

the algorithm. In order to do so, the two parties, holding secret-shares of the elements $\mathbf{x} = (x_1, \ldots, x_n)$, reveal the masked version $\mathbf{x}' = (x_1 + r_1, \ldots, x_n + r_n)$ (using a round of communication) and invoke the FSS gate for $\mathcal{G}_{\mathsf{CmpAgg}}$ where the masks $r_1, \ldots, r_n$ are sampled during the offline phase and used as secret-parameters in the FSS gate for $\mathcal{G}_{\mathsf{CmpAgg}}$. After invoking this FSS gate, the parties end up with a secret sharing of the *local ranking* of the elements, $\mathsf{LRank}(\mathbf{x}) = (\mathsf{LRank}(x_1), \ldots, \mathsf{LRank}(x_n))$. If the parties start by shuffling the set of inputs at the very start of the protocol, then they can safely reveal the local ranks (using another round of communication) and reorder the (shares of) elements of $\mathbf{x}$ locally as prescribed by the Compare-Aggregate algorithm.

Theorem 5.1 (Secure protocol from Compare-Aggregate algorithm). *Let $\mathcal{A}$ be an algorithm in the Compare-and-Aggregate model for a problem $P \in \{\mathsf{sort}, \mathsf{select}\}$ with edge complexity $e_{\mathcal{A}}(n,k)$ and $v_{\mathcal{A}}(n,k)$ where n is the number of elements and k is the number of iterations. Then, there exists a $(2+1)$-party protocol which securely realizes $\mathcal{F}_P$ for $\mathbb{G} = \mathbb{Z}_{L'}$ where $L' \geq 2L$ and $\ell' = \lceil \log L' \rceil$ and $p \in \mathbb{N}$ bit payloads. It has the following efficiency features.*

- *Online rounds:* $2k + 1$.
- *Online communication:* $2v_{\mathcal{A}}(n,k)(\ell' + \lceil \log n \rceil) + 2n(\ell' + 2p)$ *bits across both online parties.*
- *Online computation:* $2e_{\mathcal{A}}(n,k) \cdot \mathsf{DCF.Eval}[\mathbb{G}, \mathbb{Z}_n]$ *across both online parties.*
- *Offline communication:* $6n(\ell' + p) + 4n\lceil \log n \rceil + 2\ell'(v_{\mathcal{A}}(n,k) - n) + 2e_{\mathcal{A}}(n,k) \cdot (\mathsf{DCF.KeySize}(\mathbb{G}, \mathbb{Z}_n) + \lceil \log n \rceil)$ *bits from the offline party to the two online parties.*

6 COST ANALYSIS

Our goal in this section is to explore the features of each protocol in Table 1 and the protocols in the Compare-Aggregate model. Namely, we want to identify, for each protocol, the application settings (for different n, ℓ) where the protocol has competitive advantage over prior works, in terms of online performance.

Estimation methodology. To faithfully estimate the costs of our protocols and baselines, we consider *system-independent* measures (i.e., communication cost, number of rounds) and *system-dependent* measures (i.e., local computation time, end-to-end run time). One of the most important indicator for a protocol's performance is the run time, which is the sum of local computation time and network time[10]. It depends on the computation configuration (CPU frequency, number of cores) and the network configuration (bandwidth, latency). Hence, the run time numbers can vary on different architectures; therefore, beyond the conventional benchmarks under a typical configuration (Table 2), we also give an unconventional measure (Table 3) to help readers extrapolate the protocol performance under different architectures. Specifically, in Table 3, we find the computational resources (in CPU cores) for which computation breaks even with communication for our protocols, given a standard network speed (e.g, 1Gbps) and standard CPU frequency (e.g., 3GHz). That is, we give the number of cores required by each party to make the local computation time equal to the time to transmit data (data size divided by bandwidth), which we call the *breakeven point*. We believe the above information is sufficient not only to show the advantages and bottlenecks of our protocols, but also for extrapolating their behavior under different architectures.

Estimation assumptions. For FSS cost, we count the number of length-doubling PRG needed and we apply the half-evaluation optimization [17] when evaluating the (left or right) half of the PRG output. Here we do not consider the early-termination optimization for DPF evaluation [17]. We assume 300 million AES per second on 3GHz CPU, following benchmarks in prior work [26].

The local computation can be abstracted as AES calls (DCF or DPF for ranking) and random memory accesses (reordering the keys), which is experimentally shown to be dominated by AES calls[11], as follows. Concretely, on a typical architecture (3GHz CPU, 512 KB L1 cache, 8 MB L2 cache, 128 MB L3 cache), in the amortized setting, a single AES call takes 3 nanosecond while a single access to a uniformly random access to a size-10^8 integer array takes 31 nanoseconds, so the latter is 10× of the AES invocation. Now, in all of our Compare-Aggregate based secure sorting protocols, the number of AES calls is at least $100n\ell$ when $n \geq 20$ and number of iterations $r \leq 5$, and the number of times we access a uniformly random index is at most $2n$ (during shuffling) plus rn (during r rounds of Compare-Aggregate) which in total is $(r+2)n$. The main point here is that even though a single random access can cost 10× more than a single AES call, the number of AES calls is at least ℓ times larger than the number of random access calls. Therefore, for usual values of ℓ, e.g., 32, 64, AES will be the bottleneck.

For online communication, we balance it across the online parties if the protocol has asymmetric communication, using the standard technique of role rotation across different instances. Therefore, the communication cost shown in Tables 2 and 3 indicate online communication cost normalized by the number of online parties.

Baselines. We identify two baselines: garbled circuit (GC) solution (using Batcher sorting circuit [8]) and Radix sort by Asharov et al. [7] (refer to Section 1.2 for the rationale behind this choice). At a high level, the GC approach is communication bound, where it has only 2 rounds but high communication cost; the Radix approach is latency bound, where it has fast computation and low communication but an extremely large number of rounds. For details, the Radix baseline has $\approx 12\ell$ (for ℓ bit inputs) online rounds and per-party online communication $n \cdot \ell \cdot (7/3 + 32/9 \cdot \log n) + 3n \log n$ bits.

Comparison. Table 2 shows the microbenchmark estimates of our protocols and the baselines, under the conventional setting (fixing typical computation and network configurations). Besides single instance sorting benchmarks, we also estimate the numbers for a batch of multiple concurrently executed instances (as is typically the case in applications). In all the cases of n, ℓ that we show, the GC baseline has much higher communication than our protocols and this translates to a higher run time when the parties run a batch of instances (see "batch inst, 64 cores" in Table 2). The Radix sort [7] baseline, aside from higher communication compared to ours, has a large number of rounds which translates to a high run time in the WAN setting. Although our computation cost is not the best, neither our communication nor rounds becomes a bottleneck (as opposed

[10]The network time is the sum of the latency delay (caused by rounds) and the data transmitting time (the communication divided by the bandwidth).

[11]This conclusion should be insensitive to common architectures with AES-NI instruction and standard cache settings.

n	ℓ	Protocols	Rounds	Comm (KB), single inst	Comp (ms) single inst, single core	Run time (sec) LAN setting		Run time (sec) WAN setting	
						single inst, single core	batch inst, 64 cores	single inst, single core	batch inst, 64 cores
20	8	Protocol I / III	3 / 2	0.05 / 0.04	0.01 / 0.01	0.01 / < 0.01	0.24 / 0.2	0.18 / 0.12	0.41 / 0.37
		Protocol II / IV	3 / 2	0.07 / 0.07	0.03 / 0.04	0.01 / < 0.01	0.42 / 0.43	0.18 / 0.12	0.59 / 0.55
		5-iteration AAV	11	0.17	0.01	0.02	0.56	0.66	1.2
		GC	2	1.28	0.01	< 0.01	4.12	0.12	4.24
		Radix sort [7]	85	0.39	< 0.01	0.17	1.41	5.1	6.34
	64	Protocol I / III	3 / 2	0.33 / 0.22	0.09 / 0.09	0.01 / < 0.01	0.21 / 0.16	0.18 / 0.12	0.38 / 0.32
		5-iteration AAV	3 / 2	1.01	0.08	0.02	0.42	0.66	1.06
		GC	2	10.24	0.06	< 0.01	4.12	0.12	4.24
		Radix sort [7]	757	2.86	0.01	1.51	2.67	45.42	46.57
10^3	8	Protocol I / III	3 / 2	3.25 / 2.17	26.67 / 33.33	0.03 / 0.04	3.55 / 4.31	0.21 / 0.18	3.72 / 4.49
		Protocol II / IV	3 / 2	4.33 / 4.33	1.71 / 8.37	0.01 / 0.01	0.44 / 1.27	0.15 / 0.13	0.62 / 1.39
		5-iteration AAV	11	12.23	1.06	0.02	0.9	0.66	1.54
		GC	2	64	1.99	0.01	4.22	0.12	4.34
		Radix sort [7]	85	41.5	0.11	0.17	2.84	5.1	7.77
	64	Protocol I / III	3 / 2	17.25 / 11.5	213.33 / 220	0.22 / 0.22	3.48 / 3.53	0.39 / 0.34	3.65 / 3.7
		5-iteration AAV	11	54.23	8.49	0.03	0.57	0.67	1.21
		GC	2	512	15.89	0.02	4.22	0.13	4.34
		Radix sort [7]	757	305.88	0.82	1.52	3.97	45.42	47.88

Table 2: Concrete online costs of secure sorting using our protocols and baselines under different combinations of n and ℓ, estimated using 3GHz CPU, single core, and network configuration 1Gbps bandwidth, 2ms latency for LAN and 60ms latency for WAN. "Comm" means communication, and "Comp" means computation, both are online per-party for all rounds. We show protocols for n, ℓ such that they are feasible to run. We estimate run time in both the microbenchmark setting (a single instance with a single core) or the batch setting (a batch of instances concurrently run with 64 cores, where the batch size is selected so that the total input size (i.e. batch size $\times n \times \ell$) is kept same as 8 MB across all cases).

to the communication-bound GC baseline or the latency-bound Radix baseline); hence we have better run time overall.

To highlight the best protocol under various n, ℓ settings and the best improvement of the protocol over the baselines, Table 3 shows the communication, computation, and run time at the breakeven points. We give 6 combinations of small (20), medium (1,000), large (1,000,000) n, and small (8) and large (64) ℓ, and fix the input sizes (which is $n \times \ell \times$ #instances) to be the same across all these cases.

In the "small-n-small-ℓ" and "small-n-large-ℓ" cases, Protocol III is the best. This is because when n is small, the computation is cheap (and same) for both Protocol I and III but the latter has less online per-party communication and 1 less round. In the "medium-n-small-ℓ" and "large-n-small-ℓ" setting, Protocol II is the best. This is because both Protocol II/IV have computation linear in n (as opposed to $n \log^2 n$ for GC and n^2 for Protocol I/III), and Protocol II outperforms IV for the same reason of less online-per-party communication and 1 less round. Finally, in the "medium-n-large-ℓ" and "large-n-large-ℓ" setting, the 5-iteration AAV based protocol stands out. This is because it has much better computation compared to I/III, and much better communication over the baseline protocols while still maintaining low (11) constant rounds.

For the secure *median* selection problem, we provide a summary here and defer the full cost analysis to the full version. For the baselines (GC and Radix sort), computing median has the same costs as sorting. For our approaches, Protocols I, II and III have same costs both for sorting and median selection whereas Protocol IV has a lower computational cost for median compared to sorting (as shown in Table 1). Moreover, we swap out the 5-iteration AAV with a more efficient 5-iteration version of BB90 algorithm [12] that only requires linear (in n) number of comparisons. In Table 3, for the (n, ℓ) settings where Protocols II and III win for sorting, they continue to win for median. For the remaining two settings, BB90 wins with a better margin than AAV. Concretely, for $(n, \ell) = (10^3, 64)$ and $(n, \ell) = (10^6, 64)$, BB90 has an improvement factor of 4.7× (AAV has 3.6×) and 3.91× (AAV has 2.89×) respectively.

Acknowledgements. E. Boyle was supported in part by AFOSR Award FA9550-21-1-0046 and ERC Project HSS (852952). N. Gilboa was supported by ISF grant 2951/20, and MOST grant 8109282. Y. Ishai was supported by ERC grant NTSC (742754), BSF grant 2022370, ISF grant 2774/20, and ISF-NSFC grant 3127/23. Y. Ma was supported by a Microsoft Research PhD Fellowship.

REFERENCES

[1] Amit Agarwal, Elette Boyle, Niv Gilboa, Yuval Ishai, Mahimna Kelkar, and Yiping Ma. 2024. Compressing Unit-Vector Correlations via Sparse Pseudorandom Generators. In *CRYPTO*. Springer, 346–383.
[2] Miklós Ajtai, János Komlós, and Endre Szemerédi. 1983. An O(n log n) Sorting Network. In *STOC*. ACM, 1–9.
[3] Noga Alon and Yossi Azar. 1988. Sorting, approximate sorting, and searching in rounds. *SIAM Journal on Discrete Mathematics* 1, 3 (1988), 269–280.
[4] Noga Alon, Yossi Azar, and Uzi Vishkin. 1986. Tight complexity bounds for parallel comparison sorting. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*. IEEE, 502–510.
[5] Toshinori Araki, Jun Furukawa, Kazuma Ohara, Benny Pinkas, Hanan Rosemarin, and Hikaru Tsuchida. 2021. Secure graph analysis at scale. In *CCS*. 610–629.
[6] Joshua J. Arulanandham, Cristian S Calude, and Michael Dinneen. 2002. *Bead–sort: A natural sorting algorithm.* Technical Report. Department of Computer

n	ℓ	#Inst	BP cores	Protocols	T_{comp}	T_{comm}	T_{ltcy}	Online run time	Impr. Fac.
20	8	$4 \cdot 10^5$	48	Protocol III	0.11	0.11	0.12	0.34	13×
				GC	0.03	4.1	0.12	4.25	
				Radix sort	0.01	1.24	5.1	6.35	
20	64	$5 \cdot 10^4$	48	Protocol III	0.09	0.09	0.12	0.3	14×
				GC	0.03	4.1	0.12	4.25	
				Radix sort	0.01	1.15	45.42	46.57	
10^3	8	$8 \cdot 10^3$	64	Protocol II	0.23	0.22	0.18	0.63	7×
				GC	0.13	4.1	0.12	4.35	
				Radix sort	0.01	2.66	5.1	7.77	
10^3	64	10^3	20	5-iteration AAV	0.42	0.41	0.66	1.5	4×
				GC	0.4	4.1	0.12	4.61	
				Radix sort	0.04	2.45	45.42	47.91	
10^6	8	8	50	Protocol II	0.27	0.28	0.18	0.73	7×
				GC	0.64	4.1	0.12	4.85	
				Radix sort	0.03	5.16	5.1	10.3	
10^6	64	1	72	5-iteration AAV	0.47	0.48	0.66	1.61	3×
				GC	0.44	4.1	0.12	4.66	
				Radix sort	0.02	4.74	45.42	50.19	

Table 3: Online run time estimation for secure sorting, based on WAN network configuration 1Gbps bandwidth, 60ms latency and 3GHz CPU. T_{comp} is the time (in seconds) for local computation normalized by number of cores, T_{comm} is the time (in seconds) for transmitting data under the 1Gbps network, namely the communication size divided by the bandwidth. T_{ltcy} is the time (in seconds) for all rounds for the given network latency. We fix the total input size as 8MB, and show under six different application cases (different combinations of n and ℓ) the best of our protocols and the baselines. #Inst means the number of concurrently executed instances of (n, ℓ). BP means the breakeven point between T_{comp} and T_{comm} for our protocols (see detailed explanation in Section 6). Impr. Fac. means the improvement factor of our protocols compared to the best baseline protocol rounded off to the nearest integer.

Science, The University of Auckland, New Zealand.

[7] Gilad Asharov, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Ariel Nof, Benny Pinkas, Katsumi Takahashi, and Junichi Tomida. 2022. Efficient secure three-party sorting with applications to data analysis and heavy hitters. In *CCS*. 125–138.

[8] Kenneth E Batcher. 1968. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*. 307–314.

[9] Donald Beaver, Joan Feigenbaum, Joe Kilian, and Phillip Rogaway. 1997. Locally Random Reductions: Improvements and Applications. *J. Cryptology* 10 (1997), 17–36. https://doi.org/10.1007/s001459900017

[10] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. 2011. Semi-homomorphic Encryption and Multiparty Computation. In *EUROCRYPT*. Springer, 169–188.

[11] Dan Bogdanov, Sven Laur, and Riivo Talviste. 2014. A practical analysis of oblivious sorting algorithms for secure multi-party computation. In *Nordic Conference on Secure IT Systems*. Springer, 59–74.

[12] Béla Bollobás and Graham Brightwell. 1990. Parallel selection with high probability. *SIAM Journal on Discrete Mathematics* 3, 1 (1990), 21–31.

[13] Béla Bollobás and Moshe Rosenfeld. 1981. Sorting in one round. *Israel Journal of Mathematics* 38 (1981), 154–160.

[14] Elette Boyle, Nishanth Chandran, Niv Gilboa, Divya Gupta, Yuval Ishai, Nishant Kumar, and Mayank Rathee. 2021. Function Secret Sharing for Mixed-Mode and Fixed-Point Secure Computation. In *EUROCRYPT*. 871–900.

[15] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. 2019. Efficient pseudorandom correlation generators: Silent OT extension and more. In *CRYPTO*. Springer, 489–518.

[16] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2015. Function secret sharing. In *EUROCRYPT*. Springer, 337–367.

[17] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2016. Function secret sharing: Improvements and extensions. In *CCS*. 1292–1303.

[18] Mark Braverman, Jieming Mao, and S Matthew Weinberg. 2016. Parallel algorithms for select and partition with noisy comparisons. In *STOC*. 851–862.

[19] Paul Bunn, Eyal Kushilevitz, and Rafail Ostrovsky. 2022. CNF-FSS and its applications. In *Public-Key Cryptography–PKC 2022: 25th IACR International Conference on Practice and Theory of Public-Key Cryptography, Virtual Event, March 8–11, 2022, Proceedings, Part I*. Springer, 283–314.

[20] Ran Canetti. 2000. Security and Composition of Multiparty Cryptographic Protocols. *J. Cryptol.* 13, 1 (2000), 143–202.

[21] Melissa Chase, Esha Ghosh, and Oxana Poburinnaya. 2020. Secret-shared shuffle. In *ASIACRYPT*. Springer, 342–372.

[22] Ivan Damgård, Jesper Buus Nielsen, Michael Nielsen, and Samuel Ranellucci. 2017. The TinyTable Protocol for 2-Party Secure Computation, or: Gate-Scrambling Revisited. In *EUROCRYPT*. 167–187.

[23] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. 2012. Multiparty Computation from Somewhat Homomorphic Encryption. In *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7417)*, Reihaneh Safavi-Naini and Ran Canetti (Eds.). Springer, 643–662. https://doi.org/10.1007/978-3-642-32009-5_38

[24] Yevgeniy Dodis, Shai Halevi, Ron D. Rothblum, and Daniel Wichs. 2016. Spooky Encryption and Its Applications. In *CRYPTO*. Springer, 93–122.

[25] Jack Doerner and Abhi Shelat. 2017. Scaling ORAM for secure computation. In *CCS*. 523–535.

[26] Chun Guo, Jonathan Katz, Xiao Wang, and Yu Yu. 2020. Efficient and Secure Multiparty Computation from Fixed-Key Block Ciphers. In *IEEE SP*. 825–841.

[27] Koki Hamada, Ryo Kikuchi, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. 2013. Practically efficient multi-party sorting protocols from comparison sort algorithms. In *ICISC*. Springer, 202–216.

[28] Yuval Ishai, Eyal Kushilevitz, Sigurd Meldgaard, Claudio Orlandi, and Anat Paskin-Cherniavsky. 2013. On the Power of Correlated Randomness in Secure Computation. In *TCC*. 600–620.

[29] Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. 2009. Improved garbled circuit building blocks and applications to auctions and computing minima. In *CANS*. Springer, 1–20.

[30] A Pranav Shriram, Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, Bhavish Raj Gopal, and Somya Sangal. 2023. Ruffle: Rapid 3-party shuffle protocols. *Proceedings on Privacy Enhancing Technologies* 3 (2023), 24–42.

[31] Thomas Schneider and Michael Zohner. 2013. GMW vs. Yao? Efficient secure two-party computation with low depth circuits. In *FC*. Springer, 275–292.

[32] Leslie G Valiant. 1975. Parallelism in comparison problems. *SIAM J. Comput.* 4, 3 (1975), 348–355.

[33] Andrew Chi-Chih Yao. 1986. How to Generate and Exchange Secrets. In *FOCS*. 162–167.

[34] Bingsheng Zhang. 2011. Generic constant-round oblivious sorting algorithm for MPC. In *International Conference on Provable Security*. Springer, 240–256.