Efficient Secure Three-Party Sorting with Applications to Data Analysis and Heavy Hitters

Gilad Asharov
Bar-Ilan University
Ramat Gan, Israel
Gilad.Asharov@biu.ac.il

Koki Hamada
NTT Corporation
Tokyo, Japan
koki.hamada.rb@hco.ntt.co.jp

Dai Ikarashi
NTT Corporation
Tokyo, Japan
dai.ikarashi.rd@hco.ntt.co.jp

Ryo Kikuchi
NTT Corporation
Tokyo, Japan
9h358j30qe@gmail.com

Ariel Nof
Bar-Ilan University
Ramat Gan, Israel
ariel.nof@biu.ac.il

Benny Pinkas
Bar-Ilan University
Ramat Gan, Israel
benny@pinkas.net

Katsumi Takahashi
NTT Corporation
Tokyo, Japan
katsumi.takahashi.aw@hco.ntt.co.jp

Junichi Tomida
NTT Corporation
Tokyo, Japan
tomida.junichi@gmail.com

ABSTRACT

We present a three-party sorting protocol secure against passive and active adversaries in the honest majority setting. The protocol can be easily combined with other secure protocols which work on shared data, and thus enable different data analysis tasks, such as private set intersection of shared data, deduplication, and the identification of heavy hitters. The new protocol computes a stable sort. It is based on radix sort and is asymptotically better than previous secure sorting protocols. It improves on previous radix sort protocols by not having to shuffle the entire length of the items after each comparison step.

We implemented our sorting protocol with different optimizations and achieved concretely fast performance. For example, sorting one million items with 32-bit keys and 32-bit values takes less than 2 seconds with semi-honest security and about 3.5 seconds with malicious security. Finding the heavy hitters among hundreds of thousands of 256-bit values takes only a few seconds, compared to close to an hour in previous work.

1 INTRODUCTION

Secure sorting allows two or more participants to privately sort a list of m secret-shared values without revealing the underlying values to any of the participants. Secure sorting is an important building block in much more complex secure multiparty computations, such as private set intersection, join, graph analysis, private data analytics, secure deduplication, and more.

A secure sort must have a control flow that is data-independent. A common method is to apply any of the generic secure computation protocols (such as GMW [19] or BGW [9]) on common sorting networks, such as the AKS sorting network [2] or Bitonic sort [7]. This approach also allows to compose the sort with some pre-processing or post-processing, which is essential for the aforementioned applications. Nevertheless, this approach is often inefficient. Specifically, basing the secure computation on some (oblivious) sorting in the RAM model is often more efficient than basing the secure computation on sorting networks (circuits).

In this paper, we design a custom protocol for sorting m shared items among three servers in the presence of one corruption. We present both semi-honest and malicious variants and show that both outperform the state of the art. The run time of the malicious version is roughly just twice the running time of our semi-honest version. (This ratio depends on the security parameter that is used for the malicious variant.) Our sorting protocol can be easily composed with circuits or other protocols on shared data which can implement arbitrary pre-sorting and post-sorting computations on the data.

Our sorting protocol has better asymptotic communication complexity than that of previous protocols that compute the same task. Those include a radix sort protocol suggested in the context of the Sharemind system [11], oblivious quicksort [4, 23], and secure sorting using a Batcher sorting network. (We compare only between protocols that are run between three parties which share the data.) The asymptotic behavior of the different communication complexities is described in Section 1.3.

We implement our protocol with security against one corruption, and demonstrate in Section 7.2 that the implementation is concretely efficient. For example, sorting 1M, 5M and 10M records of 20-bit keys and 32-bit values, with security against semi-honest behavior, takes about 2.3 , 15 and 37 seconds, respectively. Obtaining security against malicious behavior with cheating probability 2^{-31}, increases the run time by a factor of less than 2.5.

1.1 Secure Sorting for Secure Data Analysis

Let us elaborate on a few examples of secure data analysis that can be implemented using secure sorting, such as **private set intersection (PSI)** and its variants, and **privacy-preserving telemetry** (computing heavy-hitters).

Private set intersection. Consider the private set intersection problem (PSI), where multiple parties have private sets of items and wish to learn the intersection of their sets (see, e.g. [10, 26, 29, 34] and references within). Most of the previous solutions were of protocols specific to PSI. These protocols could not be easily composed with other secure protocols for computing on the result of the intersection while keeping it private. For example, it is hard to use these protocols to compute the maximum of the values that appear in the intersection.

There are few protocols that compute PSI using a circuit (e.g. [24, 35]). Such circuit-based protocols can be composed with further processing that computes a function of the intersection. Still, just like in sorting, existing circuit-based protocols have a higher overhead than protocols specifically designed to solve the PSI problem.

A sorting protocol enables to easily compute the intersection of the input set of an arbitrary number of parties. This can be achieved by sorting the union of n input sets and then looking in the sorted union for n consecutive items which are equal to each other (as suggested in [24] for the two-party case). A sorting protocol on *shared data* seems ideal for computing the intersection of data of many parties, and further computing additional analysis on the intersection: Many parties (data owners) can share their data between a few semi-trusted servers, which then compute the sorting and the additional secure computation. This solution has a **communication pattern** which is very appealing for data owners since they do not need to communicate with (or even be aware of) other data owners. We describe here several examples of tasks for which previous secure solutions were prohibitively expensive, and which can efficiently and securely be computed using secure sorting as a building block.

Threshold multi-party PSI. Assume that many parties have private sets of data and wish to compute a modified version of the intersection functionality, which identifies any item appearing in at least 75% of the sets. This functionality is useful for identifying popular items even if they do not appear in all sets.

Sorting enables an efficient solution for computing this functionality: Suppose there are n parties. Each party separately shares its data between the servers. The servers first securely sort all inputs and obtain shares of all sorted items. The servers then compute a circuit that scans the list of sorted values looking for an item that appears in at least $0.75n$ consecutive locations. This final scan is done in linear time and can be implemented by a circuit of logarithmic depth. (In a sense, our heavy-hitters protocol of Section 6 implements a variant of this functionality where each client has a set of a single item.)

Data deduplication. In this task, the data owners submit data sets, and the goal is to compute a set which contains a single copy of each item regardless of the number of parties which submitted it. This computation is useful for cleaning data from duplicates before further analyzing it. It was also suggested in the iDash 2017 privacy challenge, in a context where many hospitals have private lists of patients, and need to identify patients who are registered in more than one hospital.[1]

Privacy-preserving telemetry and computing heavy hitters. To improve their products, device and service manufacturers must understand how products work in the field. For example, they would like to identify popular usage patterns, or conditions that cause crashes for many clients. Client data, however, must be treated as private and must not be revealed in this analysis.

[1]Track 1: De-duplication for Global Alliance for Genomics and Health (GA4GH), http://www.humangenomeprivacy.org/2017/competition-tasks.html

The Internet Security Research Group (ISRG) runs a system called Divvi-Up[2] for “privacy-respecting aggregate statistics”, based on the Prio system [15]. The service works with the clients that divide their data into shares sent to non-colluding servers, which can compute aggregate statistics such as sums and more complex functions. The system ensures robustness by verifying that corrupt clients cannot send syntactically incorrect data.

The recent Poplar system [12] extends this line of work to compute heavy-hitters of client data, namely compute the most popular items. That construction uses two non-colluding servers and is based on incremental distributed point functions. There are several areas in which Poplar can be improved: When finding heavy-hitters among strings of length n, it requires as many as n communication rounds with the clients. The construction also leaks some information about the distribution of the strings held by the clients. The runtime depends on the length of the strings, and is about an hour for 400,000 clients with 256 bit strings (although this number can be improved with stronger machines). The protocol can only be used for computing the basic heavy hitters problem, rather than securely computing additional analysis of this result.

Sorting can be used as a basic building block for computing aggregate statistics. We highlight here two sample use cases.

Computing heavy-hitters. A solution based on secure sorting can work in the following way: Each client shares its data between the servers, and can also prove in that step that its data is syntactically correct. The servers sort the elements they received, based on the desired key. As a result, items with the same key that different clients sent are moved to be adjacent to each other. The servers can then run a circuit for computing the length of runs of identical items, sort these values, and output the values appearing at least t times. In this construction clients have a single communication round with servers. Also, nothing else but the final output is released. Since the final output is computed by a circuit, this construction enables flexibility and control over the data that is released: the top values can be released with or without their counts, or even with noisy counts to support differential privacy. The output can include the most popular values, or only values which appear more often than some threshold. We describe a heavy-hitters protocol in Section 6, based on our 3-server honest-majority sorting protocols. Sorting takes seconds, and computing the circuits takes a similar amount of time. On the downside, our protocol assumes a trusted majority among three servers, whereas Poplar uses only two servers.

Computing percentiles. Assume that each client provides a value, such as a salary, or even multiple values. The goal is outputting the values at the 10%, 20%,. . .,90% percentiles of the union of all values provided by the clients. Sorting enables to securely compute this by sorting the set of contributed values, and outputting the values at locations $0.1m, 0.2m, \ldots$ (where m is the total number of values).

1.2 Our Contributions

Our contribution is a new sorting protocol based on radix sort and its optimized implementation. In order to describe and compare the asymptotic communication complexity of secure sorting, let us introduce the following notation. We consider m (shared) items of

[2]https://divviup.org/

the form $(key, value)$, and let ℓ_k be the bit-length of the keys and ℓ_p be the bit-length of the associated values (payload). Our protocol and implementation have the following advantages compared to previous works on the secure sorting of shared data.

- *Improved communication complexity.* We provide a semi-honest sorting protocol where each server has to send or receive $O(m \cdot \ell_k \cdot \log m + m \cdot \ell_p)$ bits. This improves the communication of all previous protocols (as described in Table 1.1). The hidden constants of our construction are also relatively small.
- *Security against malicious servers.* We show how to extend the protocol to provide security against malicious adversaries. The overhead of achieving this level of security is roughly twice compared to our semi-honest sort (for large enough fields). The transformation from semi-honest to malicious is based on the compiler of Chida et al. [13]. However, this compiler works on arithmetic circuits and supports few operations (additions and multiplications). We extend it to support also some additional gadgets, such as resharing and revealing of secrets. This contribution might be of independent interest.
- *Concretely efficient implementation.* We implemented our sorting protocol with novel optimizations. The results are described in Section 7 and are considerably faster than the state of the art.
- *Applications:* Our implementation enables a new set of applications on datasets whose sizes were previously beyond the reach of secure sorting protocols. We demonstrate it to implement the heavy hitters functionality.

Stable sort. Since our sorting protocol is based on radix sort, our sorting protocol is also *stable*. This means that two items with the same key are ordered according to their initial positions. Previous sorting protocols based on quicksort are unstable [23]. Stable sorting is particularly useful if, say, one needs to sort the list once according to a first key and a second key (say, a `state` and a `city`), and afterwards using the first key and a third key (say, the `state` and a `name`). With stable sort this can be done by sorting first using the first key (`state`), and then sorting the result once using the second key, and once using the third key. This minimizes the length of the keys that are used by the sorting operations.

1.3 Related Work

Efficient sorting for MPC can be implemented based on **sorting networks**. Ajtai et al. [2] proposed an asymptotically optimal sorting network known as the AKS sorting network, which has a complexity of $O(m \log m)$ comparisons, where m is the number of input items. However, this algorithm is not practical since its constant factor is very large. By contrast, Batcher's bitonic sort [7] has a complexity of $O(m \log^2 m)$ comparisons with a small constant, and is more efficient for any reasonable input size.

An algorithm is data-oblivious if its control flow is independent of the input. Data-oblivious sorting algorithms have been studied to use in MPC schemes, or in the setting where a client outsources its data to the cloud and it wishes to hide its access pattern. Goodrich proposed a data-oblivious sort called randomized Shellsort [20]. While returning a wrong output with low probability, it exhibits a complexity of $O(m)$ rounds and $O(m \log m)$ comparisons. ZigZag sort [21] is a deterministic algorithm with only $O(km \log m)$ gates (for k-bit keys) but a large depth of $O(m \log m)$ [22]. Asharov et al. [5] presented a simple data-oblivious protocol in the client-server model that works in $6(\ell_k + \ell_p)m \log m$ work and has a negligible failure probability. Such algorithms are important for the client-server model, and are designed to work in the case of a semi-honest server. Faster, maliciously-secure protocols can be constructed when there are three parties with an honest majority, such as in our settings.

Quicksort is very efficient in practice, but the control flow of this algorithm is data dependent and therefore might leak information about the inputs, even if the comparisons themselves are implemented using a secure algorithm. Hamada et al. [23] describe a secure sorting algorithm that first randomly shuffles the data, and then applies quicksort to the shuffled data. Therefore, since quicksort is applied to a random permutation of the data, the control flow of the algorithm is independent of the original order of the inputs and is easily simulatable. A drawback of this protocol is that the security proof is valid only if *all keys are different.* If this is not guaranteed by the setting in which the protocol is used, then this property can be ensured by concatenating a running index to all keys, but this solution increases the overhead. Another disadvantage of quicksort is that it does not implement a stable-sort. A quicksort protocol in the 3-party setting, secure against malicious server, was recently presented and implemented in [4].

The sorting protocol that we present is most similar to the protocol of Bogdanov et al. that is based on **radix sort** [11]. The protocols repeatedly sort items according to bits of the keys, starting from the least significant bit. The advantage of our protocol is that while the protocol of [11] computes a secure permutation of all items after sorting by each bit, our protocol computes a composition of these permutations, and only applies it to the items after processing all bits of the keys. This means that for sorting m items which have ℓ_k-bit keys and ℓ_p-bit values (payloads) associated with the keys, the m payloads are permuted only once instead of ℓ_k times. This significantly improves performance since the bulk of the communication was previously used to permute the $(\ell_k + \ell_p)$-bit items according to each bit. In addition, Bogdanov et al. [11] only considered semi-honest security, while we also achieve malicious security.

Comparison to other sorting protocols. We summarize the complexities of previous protocols that work in the same setting in Table 1.1. Our protocol is asymptotically better than the previous works. The asymptotics of our malicious protocol is the same as the semi-honest one (the overhead is roughly as little as 2).

Protocol	Communication (bits)
Semi-honest	
Sharemind system [11]	$O(\ell_k m \log m + \ell_k^2 m + \ell_k \ell_p m)$
Quicksort [4, 23]	$O((\ell_k + \log m) m \log m + \ell_p m)$
Batcher sorting network [7]	$O(\ell_k m \log^2 m + \ell_p m \log^2 m)$
Ours	$O(\ell_k m \log m + \ell_p m)$
Malicious	
Quicksort [4]	$O((\ell_k + \log m) m \log m + (\ell_p + \log \frac{1}{\epsilon}) m)$
Ours	$O(\ell_k m (\log \frac{m}{\epsilon}) + (\ell_p + \log \frac{1}{\epsilon}) m)$

Table 1.1: Asymptotic Communication of secure sorting protocols, where m is the number of items, ℓ_k is the number of bits of each key, ℓ_p is the number of bits of each associated payload, and ϵ is the failure probability.

Our sort achieves a substantial advantage when ℓ_k (the number of bits of the key) is smaller than $\log m$ (number of bits required for representing an index in the input array). In many scenarios, ℓ_k is much smaller than $\log m$. For example, when retrieving elements that satisfy a one-bit predicate, the key is just one bit. This includes examples such as retrieving all employees with a salary greater than some threshold, all users that were exposed to a certain ad, or all patients that have diabetes and are treated using some specific medicine. For 1M records, ℓ_k is 1 whereas $\log m$ is 20, i.e., our protocol is x20 faster than previous works.

2 PRELIMINARIES

We describe here the notation and definitions used in this paper, and the basic protocols used in our protocol.

Let $a := b$ denote that a is defined by b, $a||b$ denote the concatenation of a and b, and $\mathcal{G}$, $\mathcal{R}$, $\mathbb{F}$, $\mathbb{Z}$, and $\mathbb{Z}_2$ be a group, a ring, a field, the set of integers, and $\mathbb{Z}/2\mathbb{Z}$, respectively. If a is an ℓ-bit element, $a^{(i)}$ denotes the i-th bit of a, where we count the indices in the right-to-left order with 1 being the initial index, i.e., $a := a^{(\ell)}|| \cdots ||a^{(1)}$. If A is a probabilistic algorithm, $a \leftarrow \mathsf{A}(b)$ means a is an output of A on input b. If A is a set, $|A|$ denotes the cardinality of A and $||A||$ denotes the required bits to represent an element in A.

2.1 Setting and Security Model

We assume a setting of three servers P_1, P_2 and P_3, of which at most one server might be corrupt. This is the same setting and security model as in [3].

For notational simplicity, when we use an index to denote the i-th party, $i-1$ and $i+1$ refer to the previous and subsequent party. For example, P_{i+1} means P_3 if $i = 2$, P_{i+1} means P_1 if $i = 3$, and P_{i-1} means P_3 if $i = 1$.

In this work, we consider the client/server model. This model is used to outsource secure computation, where any number of clients send shares of their inputs to the servers. Therefore, we assume that the (three) servers see the input and the output of the protocol in a shared way (defined below).

We prove the security of our protocols in the standard ideal-real world paradigm [18]. We begin with semi-honest (passive) security, where the corrupted party controlled by the adversary follows the protocol but may try to learn private information. Then, we show how to compile our protocol to malicious (active) security, where the corrupted party can act in an arbitrary manner. We stress that our malicious secured protocol achieves *security with abort*, meaning that the adversary can cause the honest parties to abort without obtaining their output.

To prove security, we work in a hybrid model, where parties run a protocol with real messages and also have access to a trusted party computing a subfunctionality for them. When the subfunctionality is g, we say that the protocol works in the g-hybrid model.

2.2 Linear Secret Sharing

Our protocols rely on linear secret sharing schemes. In the 3-party setting, it is convenient to instantiate this using replicated secret sharing scheme [25]. In this scheme, a secret x is shared by choosing three random elements x_1, x_2 and x_3 such that $x_1 + x_2 + x_3 = x$, and then P_1's share is (x_1, x_2), P_2's share is (x_2, x_3) and P_3's share is (x_3, x_1). In general, party P_i holds the pair (x_i, x_{i+1}). We use the notation $[\![x]\!]$ to denote a secret sharing of x.

We define the following local operations:

- $\underline{[\![x]\!] + [\![y]\!]}$: Each party adds its local shares of x and y and stores the result, i.e., P_i stores $(x_i + y_i, x_{i+1} + y_{i+1})$.
- $\underline{\alpha \cdot [\![x]\!]}$: Each party P_i computes and stores $(\alpha \cdot x_i, \alpha \cdot x_{i+1})$.
- $\underline{\alpha + [\![x]\!]}$: P_1 computes and stores $(\alpha+x_1, x_2)$, P_2 stores (x_2, x_3) and P_3 computes and stores $(x_3, \alpha + x_1)$.

It is easy to verify that $[\![x + y]\!] = [\![x]\!] + [\![y]\!]$, $[\![\alpha \cdot x]\!] = \alpha \cdot [\![x]\!]$ and $[\![\alpha + x]\!] = \alpha + [\![x]\!]$.

Next, we define two basic interactive procedures that will be used in our protocols:

- $\underline{\mathsf{reveal}[\![x]\!]}$: In this interactive procedure, the parties reveal x by sending their shares to each other. Recall that each party misses one share, and so it suffices for party P_i to send x_i to P_{i+1} and x_{i+1} to P_{i-1}. Upon receiving x_{i-1} from the other parties, party P_i can check that it received the same share from both parties. If the shares are not identical, then P_i aborts. Otherwise, it can reconstruct x. When there are many values to be reconstructed, we can use the following optimization which reduces the communication by half. Each party only sends x_i to P_{i+1} and defers the verification to a later stage. Then, when verification takes place, P_i can send a collision-resistance hash of all x_{i+1}s to P_{i-1}, who can compare it with the hash of the values it received before.
- $\underline{\mathsf{reshare}([\![x]\!], i)}$: In this procedure, party P_{i-1} and party P_{i+1} reshare $[\![x]\!]$ to party P_i. This is done in two steps: first, P_{i-1} and P_{i+1} add a random secret sharing of 0 to the shares, and then send P_i its new shares. More formally, given the shares x_1, x_2, x_3 of x, party P_{i-1} and party P_{i+1} first choose r_1, r_2 and r_3 such that $r_1 + r_2 + r_3 = 0$ and set $x'_1 = x_1 + r_1$, $x'_2 = x_2 + r_2$ and $x'_3 = x_3 + r_3$. Then, P_{i-1} sends x'_i to P_i and P_{i+1} sends him x'_{i+1}.

Note that the total communication cost of reveal is three ring elements, whereas the cost of $\mathsf{reshare}$ is two ring elements. This assumes that each pair of parties has some shared PRF key for generating the shared randomness.

Pseudorandom secret sharing. It is well known that the parties can generate shares of a random number *without interaction*. This is known as pseudorandom secret sharing (PRSS) [16]. In PRSS, each pair of parties preliminary shares a key, i.e., P_i share key_i with P_{i-1} and key_{i+1} with P_{i+1}. When the parties compute shares of a random number α, each party P_i computes a pseudorandom function with their keys as $\alpha_i := \mathrm{PRF}_{key_i}(ctr)$ and $\alpha_{i+1} := \mathrm{PRF}_{key_{i+1}}(ctr)$ for some onetime counter ctr, and regard (α_i, α_{i+1}) as the share of a random number. PRSS securely computes the following functionality $\mathcal{F}_{\mathrm{rand}}$.

Functionality 2.1 ($\mathcal{F}_{\mathrm{rand}}$ – Generate shares of a random value)**:**

Let P_i be the corrupted party controlled by the ideal world adversary. $\mathcal{F}_{\mathrm{rand}}$ is invoked by the honest parties and $\mathcal{S}$. Upon receiving P_i's shares (α_i, α_{i+1}) from $\mathcal{S}$, functionality $\mathcal{F}_{\mathrm{rand}}$ chooses a random $\alpha \in \mathcal{R}$ and set $\alpha_{i-1} = \alpha - (\alpha_i + \alpha_{i+1})$. Then, $\mathcal{F}_{\mathrm{rand}}$ hands the pair (α_{i-1}, α_i) to P_{i-1} and the pair $(\alpha_{i+1}, \alpha_{i-1})$ to P_{i+1}.

Remark 2.2. *Although the above operations and procedures are defined for replicated secret sharing, it is easy to define similar procedures also for other linear secret sharing schemes such as Shamir's secret sharing scheme [36].*

2.3 Permutation

A permutation σ is a bijective function from a finite set to itself, usually the underlying set is $[m] = \{1, \dots, m\}$. In the context of this work, we have a vector of items $\vec{a} = (a_1, \dots, a_m)$ and we apply the permutation σ over $\vec{a}$. Then, the item a_i is moved to position $\sigma(i)$. We abuse notation and use $\sigma(\vec{a})$ to denote applying σ on $\vec{a}$, which results with a vector $\vec{b}$. Since a_i is moved to position $\sigma(i)$, it holds that $b_j = a_{\sigma^{-1}(j)}$, or equivalently, $a_i = b_{\sigma(i)}$.

We use $\vec{\sigma} = (\sigma(1), \dots, \sigma(m))$ to denote the vector of destinations of the permutation σ. Note that we distinguish between the notations σ and $\vec{\sigma}$: σ denotes the bijection function (the permutation itself), whereas $\vec{\sigma}$ is the representation of the vector of destinations. Using our notation above, we have that $\sigma([m]) = \vec{\sigma}$.

We denote by $\pi \circ \sigma$ the *composition* of two permutations, meaning that $\pi \circ \sigma(i) = \pi(\sigma(i))$. This implies that $\pi \circ \sigma(\vec{a})$ results with $\vec{b}$, where $b_j = a_{\sigma^{-1}(\pi^{-1}(j))}$. We thus have the following fact:

Fact 2.3. *Let σ, π be permutations from $[m]$ to $[m]$ and let $\vec{a} = (a_1, \dots, a_m)$ be a vector of items. Then:*

- $(\pi \circ \sigma)^{-1}(\vec{a}) = \sigma^{-1} \circ \pi^{-1}(\vec{a})$.
- $\pi \circ \pi^{-1}(\vec{a}) = \pi^{-1} \circ \pi(\vec{a}) = \vec{a}$.

We next show an important observation used in our protocols.

Observation 2.4. *Let π and σ be permutations over the set $[m]$, and let $\vec{\sigma} = (\sigma(1), \dots, \sigma(m))$ be the vector of destinations of the permutation σ. Then, $\pi(\vec{\sigma}) = \sigma \circ \pi^{-1}([m])$, i.e., a vector of destinations for the permutation $\sigma \circ \pi^{-1}$.*

Proof: Let $\vec{\sigma} = (\sigma(1), \dots, \sigma(m))$ and let $\vec{a} = \pi(\vec{\sigma})$. We have that $a_j = \vec{\sigma}_{\pi^{-1}(j)} = \sigma(\pi^{-1}(j))$. Therefore, $\pi(\vec{\sigma}) = \sigma \circ \pi^{-1}([m])$ as required. □

Notations. We work with permutations that are shared across the parties. Specifically, we have two types of sharing of a secret permutation σ:

- $[\![\vec{\sigma}]\!]$: This notation means that we consider the permutation σ as a vector of destinations $\vec{\sigma} = (\sigma(1), \dots, \sigma(m))$, and each element of the vector is secret shared across the parties. That is, the parties hold a sharing of $([\![\sigma(1)]\!], \dots, [\![\sigma(m)]\!])$.
- $\langle\!\langle \sigma \rangle\!\rangle$: This notation means that the permutation σ is shared among the parties in the following sense. We let $\sigma = \sigma_3 \circ \sigma_2 \circ \sigma_1$ for random permutations $\sigma_1, \sigma_2, \sigma_3$ under the above constraints, and σ is secret shared across the parties in a replicated way: Specifically, party P_i and party P_{i+1} hold the permutation σ_{i+1}, and thus, each party P_i holds the pair (σ_i, σ_{i+1}).

3 BUILDING BLOCKS

In this section, we specify the fundamental sub-protocols used as the components of our new sorting protocol: multiplication and shuffling.

3.1 Multiplication

A multiplication protocol receives as input $[\![a]\!]$ and $[\![b]\!]$ and outputs $[\![a \cdot b]\!]$. The ideal functionality for the multiplication protocol appears in Functionality 3.1. Observe that $\mathcal{F}_{\text{mult}}$ lets the corrupted party choose its shares of the output.

Functionality 3.1 ($\mathcal{F}_{\text{mult}}$ – Multiplication)**:**

Let P_i be the corrupted party. Upon receiving from the honest parties their shares of a and b, and an input (c_i, c_{i+1}) from P_i, $\mathcal{F}_{\text{mult}}$ reconstructs (a, b), computes $c = ab$, computes $c_{i-1} = c - (c_i + c_{i+1})$ and send the honest parties their shares.

This functionality can be realized with semi-honest (passive) security via the multiplication protocol of Araki et al. [3], which requires each party P_i to send exactly one single element to P_{i+1}.

3.2 Shuffling and Unshuffling

We describe in Protocol 3.2 a shuffling protocol based on Laur et al. [30]. The protocol receives as input a shared input vector $[\![\vec{a}]\!]$ and a shared random permutation $\langle\!\langle \pi \rangle\!\rangle$ and outputs a shared vector $[\![\vec{b}]\!]$, where $b_i = a_{\pi^{-1}(i)}$. Recall that $\pi = \pi_3 \circ \pi_2 \circ \pi_1$ where each π_i is known by exactly two parties P_i and P_{i-1}. Thus, the protocol works by letting the two parties P_i and P_{i-1} knowing π_i, to apply it over the shares of $[\![\vec{a}]\!]$ (recall that P_i and P_{i-1} together know all the shares of $\vec{a}$), and then reshare it to the third party. The resharing procedure ensures that the third party receives random shares, and therefore does not learn any information about the permutation π_i. Since each party misses one permutation (i.e., π_1, π_2 or π_3), it follows that the resulting permutation is random.

Protocol 3.2 (Shuffling protocol)**:**

Notation: $[\![\vec{b}]\!] \leftarrow$ Shuffle($\langle\!\langle \pi \rangle\!\rangle$; $[\![\vec{a}]\!]$).
Input: A secret-shared permutation $\langle\!\langle \pi \rangle\!\rangle$ and a shared vector $[\![\vec{a}]\!]$, where $\pi = \pi_3 \circ \pi_2 \circ \pi_1$ and $\vec{a} = (a_1, \dots, a_m)$. Each P_i holds π_i, π_{i+1} and shares of each a_j.
Output: The secret-shared shuffled vector $[\![\vec{b}]\!] = [\![\pi(\vec{a})]\!]$.

```
1: Let $\vec{b}_0 = \vec{a}$.
2: for $i = 1$ to 3 do
3:     Party $P_{i-1}$ and $P_i$ define $[\![\vec{b}_i]\!] = [\![\pi_i(\vec{b}_{i-1})]\!]$.
4:     The parties run reshare($[\![\vec{b}_i]\!]$, $i + 1$)
5: The parties set $\vec{b} := \vec{b}_3$.
6: return $[\![\vec{b}]\!]$.
```

Similarly, it is possible to define an unshuffling protocol, where the parties apply the inverse of π on $\vec{a}$. To achieve this, the parties work in reverse order, i.e., apply first π_3^{-1}, then π_2^{-1} and then π_3^{-1}. This ensures that $\pi^{-1} = (\pi_3 \circ \pi_2 \circ \pi_1)^{-1} = \pi_1^{-1} \circ \pi_2^{-1} \circ \pi_3^{-1}$ is applied on $\vec{a}$ as required. The unshuffling protocol is described in Protocol 3.3.

Communication complexity. In both the shuffling and the unshuffling protocol, the parties communicate when resharing the vector $[\![\vec{b}_i]\!]$. When a pair of parties reshare a secret to the third party, each of them sends him one ring element. Assuming $\vec{b}$ is of size m, each party thus sends m elements in each resharing. Since each party is required to reshare twice, the overall communication per party is $2m$ ring elements.

Protocol 3.3 (Unshuffling Protocol)**:**

```
Notation: $[\![\vec{a}]\!] \leftarrow$ UNSHUFFLE$\langle\!\langle \pi \rangle\!\rangle; [\![\vec{b}]\!]$.
Input: A secret-shared permutation $\langle\!\langle \pi \rangle\!\rangle$ and a vector $[\![\vec{b}]\!]$.
Output: The secret-shared unshuffled vector $[\![\vec{a}]\!] = [\![\pi^{-1}(\vec{b})]\!]$.
1: Let $\vec{b}_4 = \vec{b}$.
2: for $i = 3$ to $1$ do
3:     Party $P_{i-1}$ and $P_i$ compute $[\![\vec{b}_i]\!] = [\![\pi_i^{-1}(\vec{b}_{i+1})]\!]$.
4:     The parties run reshare($[\![\vec{b}_i]\!]$, $i+1$)
5: The parties set $\vec{a} := \vec{b}_1$.
6: return $[\![\vec{a}]\!]$.
```

4 SECURE SORTING WITH SEMI-HONEST SECURITY

In this section, we introduce our secure sorting protocol. Our protocol begins with the parties holding keys and items to sort in a secret shared way. Moreover, we assume that the parties hold a secret sharing of *each* bit of the keys. Note that if this is not the case, then the parties can run a secure protocol for bit decomposition such as [28]. Our protocol computes *stable* sorting: it rearranges the order of the items based on the keys, and maintains the relative order of items that have equal keys. In other words, given two items v_i and v_j with keys $k_i = k_j$ and $i < j$, the sorting permutation will map them into positions i' and j' that satisfy the condition $i' < j'$.

Our protocol is inspired by the protocol of Bogdanov et al. [11]. We reduce significantly their communication cost. Furthermore,that protocol considered only semi-honest security while we show in Section 5 how to add malicious security.

The protocol of Bogdanov et al. [11] implements a distributed version of Radix sort. In the kth step of the protocol, the parties compute a secret sharing of a stable permutation of the kth bit of the keys and then apply the permutation over the keys and items.

GENBITPERM: Generating stable bit sorting permutation. The protocol GENBITPERM, which is described in Protocol 4.1, computes the permutation of a stable sort *for a single bit key*. That is, given the keys, it finds which permutation should be applied on the keys such that the result would be a stable sorting of the input keys. Stable sorting one-bit keys is also known in the literature as stable compaction [6, 32, 33].

Intuitively, GENBITPERM works as follows. If we are interested in sorting $\vec{k} = (1, 1, 0, 0)$, then the output permutation should be $\rho = (3, 4, 1, 2)$. Observe that $\rho(\vec{k}) = (0, 0, 1, 1)$, as required. The key idea is to count the number of 0 keys in $\vec{k}$; let Z be this number. Then, the ith 0-key in $\vec{k}$ should be moved to the ith position in the output vector, and the jth 1-key in $\vec{k}$ should be moved to the $(Z + j)$th position in the output vector.

An MPC-friendly implementation of the above procedure is the following. First, compute two vectors $\vec{f}^{(0)} = \vec{1} - \vec{k}$ and $\vec{f}^{(1)} = \vec{k}$, where the i-th element in each one of the vectors represents whether the i-th key of $\vec{k}$ is 0 and 1, respectively. In our example, $\vec{f}^{(0)} = (0, 0, 1, 1)$ and $\vec{f}^{(1)} = (1, 1, 0, 0)$. Then, we compute prefix sums of the vector $\vec{f}^{(0)}$ to obtain a vector $\vec{s}^{(0)} = (0, 0, 1, 2)$, where $s_i^{(0)} = \sum_{j\leq i} f_j^{(0)}$. It is easy to see that the vector $\vec{s}^{(0)}$ tells the destination of each 0-element in $\vec{k}$. In our example, this tells us that the third element in $\vec{k}$ should be moved to position 1, and the forth element should be moved to position 2. Moreover, the last element in $\vec{s}^{(0)}$ represents the number of 0s in the input $\vec{k}$. This is Z from above. We then compute also prefix sums of the vector $\vec{f}^{(1)}$, while starting the sum from Z, to obtain $\vec{f}^{(1)} = (3, 4, 4, 4)$. The vector $\vec{s}^{(1)}$ tells the destination of each 1-element in $\vec{k}$. To obtain the final permutation ρ, we need to select for each element in $\vec{k}$ whether to take the destination from $\vec{s}^{(0)}$ or $\vec{s}^{(1)}$. This is achieved by computing

$$(1 - k_i) \cdot s_i^{(0)} + k_i \cdot s_i^{(1)} = s_i^{(0)} + k_i \cdot (s_i^{(1)} - s_i^{(0)}) .$$

The result is $\vec{\rho} = (3, 4, 1, 2)$.

To convert the above into a secure protocol, our input is now a vector of shared keys $[\![\vec{k}]\!]$, and the output should be $[\![\vec{\rho}]\!]$. Observe that all steps in the above computation are linear except for the last step, which involves one multiplication per key. Proceed to Protocol 4.1 for the specification.

Protocol 4.1 (Generating permutation of stable sort for a single bit key)**:**

```
Notation: $[\![\vec{\rho}]\!] \longleftarrow$ GENBITPERM($[\![\vec{k}]\!]$).
Input: Secret-shared bit-wise keys $[\![\vec{k}]\!]$, where $\vec{k} = (k_1, \ldots, k_m)$ and
    $k_i \in \{0, 1\}$ for $1 \leq i \leq m$.
Output: The secret-shared permutation $[\![\vec{\rho}]\!]$ which is a stable sorting
    of $\vec{k}$.
1: for $1 \leq i \leq m$ do
2:     $[\![f_i^{(0)}]\!] := 1 - [\![k_i]\!]$.
3:     $[\![f_i^{(1)}]\!] := [\![k_i]\!]$.
4: $[\![s]\!] := [\![0]\!]$.
5: for $j = 0$ to $1$ do
6:     for $i = 1$ to $m$ do
7:         $[\![s]\!] := [\![s]\!] + [\![f_i^{(j)}]\!]$.
8:         $[\![s_i^{(j)}]\!] := [\![s]\!]$.
9: for $1 \leq i \leq m$ do
10:    The parties send ($[\![k_i]\!]$, $[\![s_i^{(1)}]\!] - [\![s_i^{(0)}]\!]$) to $\mathcal{F}_{\text{mult}}$ (Functionality 3.1), and receive $[\![t_i]\!]$.
11:    $[\![\rho(i)]\!] := [\![s_i^{(0)}]\!] + [\![t_i]\!]$.
12: return $[\![\vec{\rho}]\!] = ([\![\rho(1)]\!], \ldots, [\![\rho(m)]\!])$.
```

Applying the shared permutation. Recall that in Radix sort, we use stable sort for sorting each bit of the keys, each such sort will be performed using an independent invocation of Protocol 4.1. Once the parties obtain the permutation ρ_j for the jth bit of the key, they need to apply it over the keys $\vec{k}$, while keeping ρ_j as secret. I.e., the goal is to compute $[\![\rho_j(\vec{k})]\!]$. To keep ρ_j secret, the parties choose a random shared permutation $\langle\!\langle \pi \rangle\!\rangle$ and apply it over ρ_j. Recall also that ρ_j is shared as $[\![\vec{\rho}_j]\!]$ and π is shared as $\langle\!\langle \pi \rangle\!\rangle$, i.e., $[\![\vec{\rho}_j]\!] = ([\![\rho_j(1)]\!], \ldots, [\![\rho_j(m)]\!])$ and $\langle\!\langle \pi \rangle\!\rangle = \left((\pi_i, \pi_{i+1})_{i=1}^3\right)$. Thus, we can invoke Protocol 3.2 to apply π on $\vec{\rho}_j$ and the parties reveal the result publicly. By Observation 2.4, we know that $\pi(\vec{\rho}_j) = \rho_j \circ \pi^{-1}$, and this permutation is now public. The parties invoke Protocol 3.2 again to shuffle $[\![\vec{k}]\!]$ using π to obtain $[\![\pi(\vec{k})]\!]$. Finally, by applying locally the public permutation $\rho_j \circ \pi^{-1}$ over $[\![\pi(\vec{k})]\!]$, the parties obtain $[\![\rho_j \circ \pi^{-1} \circ \pi(\vec{k})]\!] = [\![\rho_j(\vec{k})]\!]$ as required.

Our protocol to apply a shared permutation over a list of shared keys is called ApplyPerm and is formally described in Protocol 4.2.

Protocol 4.2 (Applying a shared-vector permutation)**:**

```
Notation: $[\![\vec{k}']\!] \leftarrow \textsc{ApplyPerm}([\![\vec{\rho}]\!]; [\![\vec{k}]\!])$.
Input: A secret-shared permutation $[\![\vec{\rho}]\!]$ and a secret-shared vector
    $[\![\vec{k}]\!] = ([\![k_1]\!], \ldots, [\![k_m]\!])$.
Output: The secret-shared vector $[\![k']\!]$ such that $\vec{k}' = \rho(\vec{k})$.
1: The parties call $\mathcal{F}_{\text{rand}}$ and receive $\langle\langle\, \pi \,\rangle\rangle$.
2: $[\![\pi(\vec{\rho})]\!] \leftarrow \textsc{Shuffle}(\langle\langle\, \pi \,\rangle\rangle; [\![\vec{\rho}]\!])$        ▷ (Protocol 3.2)
3: $[\![\pi(\vec{k})]\!] \leftarrow \textsc{Shuffle}(\langle\langle\, \pi \,\rangle\rangle; [\![\vec{k}]\!])$        ▷ (Protocol 3.2)
4: The parties run reveal($[\![\pi(\vec{\rho})]\!]$) and obtain $\pi(\vec{\rho}) = \rho \circ \pi^{-1}$
                                                  ▷ (see Observation 2.4)
5: The parties locally apply $\rho \circ \pi^{-1}$ on $[\![\pi(\vec{k})]\!]$ and obtain $[\![\vec{k}']\!] =$
   $[\![\rho(\vec{k})]\!]$.
6: return $[\![\vec{k}']\!]$.
```

A bottleneck in [11]. In [11], each permutation ρ_j is applied over each one of the bits of each key. That is, the parties first compute GenBitPerm on the least significant bit to obtain ρ_1, and apply it over $\vec{k}$ to sort the keys according to the least significant bit. Denote the result by $\vec{k}'$. Then, the parties compute GenBitPerm again on $\vec{k}'$, this time on the second less significant bit, and then apply it over the $\vec{k}'$ to obtain a vector which is sorted according to the two less significant bits, and so on. Since each bit of the key is secret shared across the parties separately, this yields high communication overhead, as when the parties shuffle $\vec{k}$, each pair of parties need to reshare the secrets to the third party. If the keys are of size ℓ_k, this implies $2\ell_k \cdot m$ ring elements sent per party for resharing (since there are m keys and each party does resharing twice for each of the ℓ_k shared bits).

Reducing the overhead. To reduce this overhead, we observe that instead of applying the permutation over the entire key, it suffices to apply a *composed* permutation over a single shared bit each time. Specifically, after the first call to GenBitPerm to sort according to the least significant bit, the parties obtain the permutation ρ_1. Instead of applying ρ_1 over all bits of the keys $\vec{k}$, it suffices to apply the permutation over just the the shared *second* least significant bit. The parties then can apply GenBitPerm on the result to obtain ρ_2, and then apply $\rho_2 \circ \rho_1$ on the third significant bit of the keys $\vec{k}$, and run GenBitPerm again to obtain ρ_3, and so on. To implement this idea, we show how to compose two secret permutations (e.g., $\rho_2 \circ \rho_1$) below.

The crucial point here is that composing two permutations requires each pair of parties to reshare a *single* secret. Hence, in each step of the protocol, instead of applying a permutation over all the remaining bits which are shared separately, the parties now need to compose two permutations and apply the result over a *single* shared bit. The communication overhead is now $4 \cdot m$, thereby removing the multiplicative ℓ_k factor.

Composing two secret permutations. Our protocol to compose two permutations ρ and σ is called Compose and is formally described in Protocol 4.3. Similarly to ApplyPerm, it starts by shuffling σ with a random permutation π. From Observation 2.4, it follows that $\pi(\vec{\sigma}) = \sigma \circ \pi^{-1}$. Then, the parties reveal $\sigma \circ \pi^{-1}$ and apply its inverse locally over $[\![\vec{\rho}]\!]$. The parties then obtain $[\![\gamma]\!] = [\![\pi \circ \sigma^{-1}(\vec{\rho})]\!]$, which is a sharing of the permutation $\rho \circ \sigma \circ \pi^{-1}$.

The goal is to obtain a sharing of $\rho \circ \sigma$, and thus we have to cancel π^{-1}. To do that, we simply unshuffle $\vec{\gamma}$ with π. In other words, we compute $\pi^{-1}(\vec{\gamma})$. Using Observation 2.4, it holds that

$$\pi^{-1}(\vec{\gamma}) = \gamma \circ (\pi^{-1})^{-1} = \rho \circ \sigma \circ \pi^{-1} \circ \pi = \rho \circ \gamma \ .$$

Protocol 4.3 (Composition of two share-vector permutations)**:**

```
Notation: $[\![\vec{\tau}]\!] \leftarrow \textsc{Compose}([\![\vec{\sigma}]\!], [\![\vec{\rho}]\!])$.
Input: Secret-shared two permutations ($[\![\vec{\sigma}]\!]$, $[\![\vec{\rho}]\!]$).
Output: The secret-shared permutation $[\![\vec{\tau}]\!]$, where $\tau = \rho \circ \sigma$.
1: The parties call $\mathcal{F}_{\text{rand}}$ and obtain $\langle\langle\, \pi \,\rangle\rangle$.
2: $[\![\pi(\vec{\sigma})]\!] \leftarrow \textsc{Shuffle}(\langle\langle\, \pi \,\rangle\rangle; [\![\vec{\sigma}]\!])$.        ▷ (Protocol 3.2)
3: Reveal $[\![\pi(\vec{\sigma})]\!]$ and obtain $\pi(\vec{\sigma})$, i.e., $\sigma \circ \pi^{-1}$ ▷ (Observation 2.4)
4: The parties locally apply $(\sigma \circ \pi^{-1})^{-1} = \pi \circ \sigma^{-1}$ to $[\![\vec{\rho}]\!]$ and obtain
   $[\![\vec{\gamma}]\!] = [\![\pi \circ \sigma^{-1}(\vec{\rho})]\!]$, i.e., $\gamma = \rho \circ \sigma \circ \pi^{-1}$    ▷ (Observation 2.4)
5: $[\![\vec{\tau}]\!] \leftarrow \textsc{Unshuffle}(\langle\langle\, \pi \,\rangle\rangle; [\![\vec{\gamma}]\!])$        ▷ (Protocol 3.3)
   Note that $\tau = \pi^{-1}(\vec{\gamma}) = \gamma \circ \pi = \rho \circ \sigma$.
6: return $[\![\vec{\tau}]\!]$.
```

Putting it all together: Generating the permutation of stable sort. We describe in Protocol 4.4 our main stable sorting protocol. As explained above, the parties go over the shared bits from the least significant to the most significant. In each step, they generate a stable sorting permutation of the current bit, compose it with permutation from the previous step, and apply it over the next shared bit.

Protocol 4.4 (Securely Generating a Stable Sorting Permutation)**:**

```
Notation: $[\![\vec{\sigma}]\!] \leftarrow \textsc{GenPerm}([\![\vec{k}]\!])$.
Input: Secret-shared keys $[\![\vec{k}]\!] = ([\![k_1]\!], \ldots, [\![k_m]\!])$ where $[\![k_i]\!] =$
    $([\![k_i^{(1)}]\!], \ldots, [\![k_i^{(\ell_k)}]\!])$.
Output: $[\![\vec{\sigma}]\!]$, where $\vec{\sigma}$ is the permutation for stable sorting the vec-
    tor $\vec{k}$.
1: For each $j \in [\ell_k]$, let $[\![\vec{k}^{(j)}]\!] = ([\![k_1^{(j)}]\!], \ldots, [\![k_m^{(j)}]\!])$
2: $[\![\vec{\rho}_1]\!] \leftarrow \textsc{GenBitPerm}([\![\vec{k}^{(1)}]\!])$.        ▷ (Protocol 4.1)
3: $[\![\vec{\sigma}_1]\!] := [\![\vec{\rho}_1]\!]$.
4: for $j = 2$ to $\ell_k$ do
5:     $[\![\vec{k}'^{(j)}]\!] \leftarrow \textsc{ApplyPerm}([\![\vec{\sigma}_{j-1}]\!]; [\![\vec{k}^{(j)}]\!])$.        ▷ (Protocol 3.2)
6:     $[\![\vec{\rho}_j]\!] \leftarrow \textsc{GenBitPerm}([\![\vec{k}'^{(j)}]\!])$.        ▷ (Protocol 4.1)
7:     $[\![\vec{\sigma}_j]\!] \leftarrow \textsc{Compose}([\![\vec{\sigma}_{j-1}]\!], [\![\vec{\rho}_j]\!])$.        ▷ (Protocol 4.3)
8: return $[\![\vec{\sigma}_{\ell_k}]\!]$
```

Once the parties hold a the shared permutation, the parties can call the protocol ApplyPerm once again on the data itself to apply the permutation on the shared data. Then, the parties will obtain a secret sharing of the sorted vector of data items, as required.

4.1 Communication complexity

In this section, we analyze the communication cost of our protocol. Assume that there are m items to sort and the size of each key is ℓ_k bits and the size of the payload is ℓ_p bits. We let $||\mathcal{R}||$ be the bit length of representing an element in the ring. Recall that our protocol uses the following three building blocks to sort m items:

- GenBitPerm includes m calls to $\mathcal{F}_{\text{mult}}$, each involving sending one ring element. Thus, each party sends $m||\mathcal{R}||$ bits.
- ApplyPerm includes two calls to the shuffling protocol (Protocol 3.2) and revealing of m secrets in the ring $\mathcal{R}$. Each secret represents a destination in the permutation, and those we assume that $|\mathcal{R}| \geq m$. The shuffling protocol requires communication of $2m$ ring elements sent per party. Revealing requires sending m ring element in the semi-honest setting. So, ApplyPerm requires sending $5m||\mathcal{R}||$ bits.
- Compose has the same complexity as ApplyPerm, i.e., $5m||\mathcal{R}||$.

Iterating over the ℓ_k bits of the keys, we get that the number of bits sent by each party in GenPerm (Protocol 4.4) is

$$11m \cdot ||\mathcal{R}|| \cdot \ell_k$$

The parties then call ApplyPerm to apply the stable sorting permutation on the shared data. In that call we distinguish between working on the keys and the shuffle of the payload (Step 3 in Protocol 4.2) where there we embed the payload in a ring $\mathcal{R}'$ with $|\mathcal{R}'| \geq 2^{\ell_p}$. Overall we get:

$$11m \cdot ||\mathcal{R}|| \cdot \ell_k + 3m \cdot ||\mathcal{R}|| + 2m||\mathcal{R}'|| \tag{1}$$

bits sent per party. Letting $||\mathcal{R}|| = \log m$ and $||\mathcal{R}'|| = \ell_p$, we get that each party sends is bounded by $14\ell_k \cdot m \log m + 2\ell_p m$.

This communication complexity is asymptotically better than that of all the previous protocols, as depicted in Table 1.1. Furthermore, we improve the concrete efficiency of our protocol by introducing several optimization techniques in Appendix A, which allow us to reduce the communication cost of the protocol described above to about 1/3. Some of the optimization techniques are applicable to the protocol with malicious security presented in Section 5.

4.2 Security

In order to prove security, we first define an ideal functionality for stable sorting. Observe that our protocol actually supports computing two functionalities: generating the permutation of a stable sort, and sorting a vector of shares by applying this permutation. We define the functionality $\mathcal{F}_{\text{sort}}$ (in Functionality 4.5) to support these two functions.Note that $\mathcal{F}_{\text{sort}}$ outputs only a secret sharing to the parties. This is in alignment with the client-server model that we consider in this work.

Functionality 4.5 ($\mathcal{F}_{\text{sort}}$ – Stable sorting)**:**

- **GenPerm:** Upon receiving (`GenPerm`, $[\![\vec{k}]\!]$) from the honest parties, $\mathcal{F}_{\text{sort}}$ reconstructs $\vec{k}$, and computes the permutation σ such that $\sigma(i) \leq \sigma(k)$ if $k_i \leq k_j$ and $i < j$. Then, it generates $[\![\vec{\sigma}]\!]$ and sends each party its shares.
- **ApplyPerm:** Upon receiving (ApplyPerm, $[\![\vec{\sigma}]\!]$, $[\![\vec{v}]\!]$) from the honest parties, $\mathcal{F}_{\text{sort}}$ reconstructs $\vec{v}$ and the permutation σ and applies $\sigma(\vec{v})$ to obtain $\vec{v}'$. It computes shares of $\vec{v}'$ and sends each party its shares.

Therefore, we have the following theorem. proven in Appendix B:

Theorem 4.6. *Protocol (GenPerm, ApplyPerm) securely computes $\mathcal{F}_{sort}$ in the ($\mathcal{F}_{rand}$, $\mathcal{F}_{mult}$)-hybrid model in the presence of semi-honest adversaries controlling a single party.*

5 ACHIEVING MALICIOUS SECURITY

In this section we show how to augment our protocol to ensure malicious security. Following the same approach as in, e.g., [8, 14, 31], we use an existing efficient compiler from semi-honest to malicious security– namely, the popular compiler of Chida et al. [13]– and adapt it to our protocol. This compiler works by adding an information-theoretic MAC for each secret throughout the computation, and then use the MAC to detect cheating. However Chida et al. [13] only considered protocols for computing simple arithmetic circuits, using only addition and multiplication operations. In contrast we have two additional types of operations with interaction: revealing and resharing. It turns out that applying the compiler to these operations is not straightforward. We show how to extend the compiler to this richer set of operations. These new techniques may be of independent interest.

5.1 Building Blocks

We start with introducing some building blocks that we use in our malicious protocol. Specifically, we take the same building blocks and protocols from the semi-honest setting and examine what security they guarantee when the adversary is malicious.

Authenticated secret sharing. The first step towards achieving malicious security, according to the compiler, is to add a MAC to each secret. Specifically, for each secret x that is held by the parties during the computation, the parties will hold the pair ($[\![x]\!]$, $[\![r \cdot x]\!]$), where r is a random secret unknown to any of the parties. Note that linear operations can be still carried-out locally without interaction. To multiply ($[\![x]\!]$, $[\![r \cdot x]\!]$) and ($[\![y]\!]$, $[\![r \cdot y]\!]$), the idea of [13] is to call $\mathcal{F}_{\text{mult}}$ twice: once to multiply $[\![x]\!]$ and $[\![y]\!]$, and a second time to multiply $[\![r \cdot x]\!]$ with $[\![y]\!]$. If no one cheats, then clearly the parties will obtain the pair ($[\![x \cdot y]\!]$, $[\![r \cdot (xy)]\!]$) as required. However, if the adversary cheats, then the honest parties can use the MAC to detect cheating, relying on the fact that r is unknown by the adversary.

Security up to additive attacks. The compiler of Chida et al. [13] converts a semi-honest protocol to a maliciously secure protocol. In order for the compiler to work, the semi-honest protocol has to be secure up to an additive attack [17]. Informally, this requirement means that if we run the semi-honest protocol, as is, but with a malicious adversary, then this the only attack that this adversary can carry out is introducing additive errors to the output. E.g., when multiplying two private inputs x and y, the adversary can only cause the output to be $xy + e$ for an error e of its choice.

To model this, one can modify the functionality that the semi-honest protocol realizes with an augmented functionality, in which the ideal world adversary chooses an offset and hands it to the functionality. The offset should be independent of the underlying secrets. The augmented functionality computes the output in the same manner as the underlying semi-honest functionality, but adds the offset to the outputs. Then, we have to show that the semi-honest protocol realizes this augmented functionality in the presence of a malicious adversary. The compiler of Chida et al. [13] then adds a lightweight verification step that can detect such attacks, thereby providing security against fully-malicious adversaries.

The inner-product functionality ($\mathcal{F}^{\text{add}}_{\ell-\text{mult}}$). The first building block that we discuss is the inner product of two shared vectors.

Note that this functionality also implements a standard multiplication of two shared inputs.

In Functionality 5.1, we describe $\mathcal{F}^{\text{add}}_{\ell-\text{mult}}$ which is just an explicit description of an augmented functionality of a semi-honest inner product. The inputs of the parties are shares of two secret vectors, and the functionality first receives those inputs from the honest parties (recall that in the honest majority setting, the honest parties hold all the shares). It then reconstructs the vectors and computes their inner product. It adds to the resulting vector an offset chosen by the adversary, and sends shares of this output to the parties.

Functionality 5.1 ($\mathcal{F}^{\text{add}}_{\ell-\text{mult}}$ – Extended Multiplication with an Additive Error)**:**

Let $\mathcal{S}$ be the ideal world adversary controlling party P_i.
Upon receiving from the honest parties their shares of $(a_1, \ldots, a_\ell)$ and $(b_1, \ldots, b_\ell)$, and an input (c_i, c_{i+1}, Δ) from $\mathcal{S}$, $\mathcal{F}^{\text{add}}_{\ell-\text{mult}}$ proceeds as follows:

(1) Reconstruct $(a_1, \ldots, a_\ell), (b_1, \ldots, b_\ell)$ and send $\mathcal{S}$ all the corrupted party's shares of these values.
(2) Compute $c = \sum_{k=1}^{\ell} a_k \cdot b_k + \Delta$.
(3) Compute $c_{i-1} = c - (c_i + c_{i+1})$.
(4) Send (c_{i-1}, c_i) to P_{i-1} and $(c_{i+1, c_{i-1}})$ to P_{i+1}.

Chida et al. [13] showed that the efficient three party semi-honest multiplication protocol of [3], which is the multiplication protocol that we used in the semi-honest setting, securely computes $\mathcal{F}^{\text{add}}_{\ell-\text{mult}}$ with $\ell = 1$ in the presence of a malicious adversary. Moreover, the protocol can also be easily extended for inner product, i.e., $\ell > 1$, and securely computes $\mathcal{F}^{\text{add}}_{\ell-\text{mult}}$, without increasing the cost (regardless of the size of ℓ).

The resharing local shuffle functionality ($\mathcal{F}^{\text{add}}_{\text{reshare-shuffle}}$). We proceed to the resharing operation. Recall that resharing is carried-out in Shuffle (Protocol 3.2), when a pair of parties locally permute a vector of shares and then reshare the result to the third party. We model a functionality associated with each such iteration in the protocol. Specifically, in each iteration parties P_{i-1} and P_i locally shuffle the current vector $\vec{x}$ and then reshare the result, by having each of them sending one element to P_{i+1}. A corrupted party might introduce an error in such an iteration, and we now show that this is just an additive attack.

Looking ahead, the reason why we do not show a functionality for the entire shuffling protocol, but rather just for each iteration of the protocol, is that we will have to store the intermediate values that the parties receive in this procedure and verify that no additive attack was introduced in the process. A verification procedure could have been introduced at the end of this building block to ensure that no additive attack was introduced; this would allow a “cleaner" modeling. However, this would increase the cost of the protocol, and we defer the verification step only when truly needed.

In the functionality (Functionality 5.2), the honest parties hand the functionality all the input shares and the permutation (in each iteration, at least one of the honest parties knows the permutation). Then, the functionality applies the permutation and randomizes the output shares, by adding a sharing of 0. If the corrupted party is one of the resharing parties, then the adversary is allowed to add an error to one of its shares.

Functionality 5.2 ($\mathcal{F}^{\text{add}}_{\text{reshare-shuffle}}$ – Resharing after a Local Shuffling with an Additive Error)**:**

Let P_i be the corrupted party controlled by the ideal world adversary $\mathcal{S}$.
The functionality receives from the honest parties the vector of shares of $\vec{x}_1, \vec{x}_2, \vec{x}_3$ of length m and an iteration index j. The shares of P_i are handed to $\mathcal{S}$.
The functionality chooses $\vec{r}_1, \vec{r}_2, \vec{r}_3$ such that $\vec{r}_1 + \vec{r}_2 + \vec{r}_3 = 0$.
If $j = i$ or $j = i + 1$ (i.e., P_i is one the resharing parties), then it hands $\vec{r}_1$, $\vec{r}_2$ and $\vec{r}_3$ to $\mathcal{S}$.
Then:

- **Case I:** $j = i$**:** *(In this case, P_i and P_{i-1} permute their shares and reshare them to P_{i+1}.)*
  Then, upon receiving π from P_{i-1}, the functionality:
  (1) Sends π to $\mathcal{S}$ to receive back $\vec{\Delta}$.
  (2) Computes $\vec{y}_1 = \pi(\vec{x}_1) + \vec{r}_1$, $\vec{y}_2 = \pi(\vec{x}_2) + \vec{r}_2$ and $\vec{y}_3 = \pi(\vec{x}_3) + \vec{r}_3$.
  (3) Hands $(\vec{y}_{i-1}, \vec{y}_i)$ to P_{i-1} and $(\vec{y}_{i+1} + \vec{\Delta}, \vec{y}_{i-1})$ to P_{i+1}.
- **Case II:** $j = i+1$**:** *(In this case, P_i and P_{i+1} permute their shares and reshare them to P_{i-1}).*
  Then, upon receiving π from P_{i+1}, the functionality:
  (1) Sends π to $\mathcal{S}$ to receive back $\vec{\Delta}$.
  (2) Computes $\vec{y}_1 = \pi(\vec{x}_1) + \vec{r}_1$, $\vec{y}_2 = \pi(\vec{x}_2) + \vec{r}_2$ and $\vec{y}_3 = \pi(\vec{x}_3) + \vec{r}_3$.
  (3) Hands $(\vec{y}_{i-1}, \vec{y}_i + \vec{\Delta})$ to P_{i-1} and $(\vec{y}_{i+1}, \vec{y}_{i-1})$ to P_{i+1}.
- **Case III:** $j = i - 1$**:** *(In this case, P_{i-1} and P_{i+1} permute their shares and reshare them to P_i.)*
  Then, upon receiving π from P_{i+1}, the functionality:
  (1) Computes $\vec{y}_1 = \pi(\vec{x}_1) + \vec{r}_1$, $\vec{y}_2 = \pi(\vec{x}_2) + \vec{r}_2$ and $\vec{y}_3 = \pi(\vec{x}_3) + \vec{r}_3$.
  (2) Hands $(\vec{y}_{i-1}, \vec{y}_i)$ to P_{i-1}, $(\vec{y}_{i+1}, \vec{y}_{i-1})$ to P_{i+1} and $(\vec{y}_i, \vec{y}_{i+1})$ to $\mathcal{S}$.

Lemma 5.3. *Each iteration i ($i \in [3]$) in our Shuffling protocol (Protocol 3.2) securely computes $\mathcal{F}^{add}_{reshare\text{-}shuffle}$ in the presence of a single malicious party.*

Unshuffling. We remark that in the unshuffling protocol (Protocol 3.3), each iteration is also secure up to an additive error. This can be showed via a proof that is identical to the proof above and therefore we omit the details.

Insecurity of the revealing procedure. We proceed to the third interactive procedure in our protocol - revealing a secret. Recall that ensuring that the opened secret is indeed the secret that was shared is easy: each share is known by two parties, and so sending an incorrect share by a corrupted P_i will be detected by comparing it to the message received from an honest party. However, this is not enough. We need also to show that no information is leaked when the secret is revealed. It turns out that in our semi-honest revealing protocol, the adversary can carry out attacks that cannot be described as additive attacks.

To see this, consider the following protocol. The parties hold shares $[\![\rho]\!]$ of a secret permutation ρ, and the parties wish to apply a random permutation $\langle\!\langle \pi \rangle\!\rangle$ on ρ and then reveal the result. Recall that we have such a step in Protocol 4.2. To apply the random permutation π on ρ, the parties use our shuffling protocol. When the parties run SHUFFLE($\langle\!\langle \pi \rangle\!\rangle$, $[\![\vec{\rho}]\!]$), the adversary can introduce an additive error to each one of the coordinates of the result $\pi(\vec{\rho})$.

Assume that the attacker is P_1 and it guesses that $\rho(3) = \rho(2) + 1$. To check if this is correct, the attacker adds 1 to the second item and -1 to the third item. Now, if the guess is correct, e.g., $\rho = (3, 1, 2)$, then the revealed output would be a permutation π of $(3, 2, 1)$, and therefore would be a permutation. On the other hand, say if $\rho = (3, 2, 1)$, then the revealed vector would be a permutation π of $(3, 3, 0)$, i.e., it would not be a permutation. The adversary sees the revealed vector and can distinguish between the two cases.

Preventing the above attack. To prevent the attack, we essentially apply the compiler of Chida et al. [13] that verifies whether additive errors were introduced by the adversary. Only after the parties ensure that cheating did not take place, they can proceed to reveal secrets. This enforces honest behavior up to the revealing phase, and thus it is safe to reveal. Note that the verification step in Chida et al. [13] is deferred to the final step of the protocol just before the parties reveal their outputs, whereas in our protocol we have to apply the verification step several times in the computation, essentially before each call of the revealing procedure. This is because Chida et al. [13] supports only multiplications and additions and there are no intermediate values that are being revealing throughout the computation. In our protocol we do reveal (masked) intermediate values, which allow us to perform more operations locally and in the clear, thus process the task faster.

Verifying correctness in the presence of additive errors. We next describe how the verification phase of Chida et al. [13] works in our protocol. The main idea is that given many secrets $z_1, \ldots, z_n$ that the parties wish to verify, the parties can take the MACs $r \cdot z_1, \ldots, r \cdot z_n$ and use them to verify correctness. Specifically, the parties take a random linear combination $u = \sum_{k=1}^{n} \alpha_k \cdot z_k$ and $v = \sum_{k=1}^{n} \alpha_k \cdot (r \cdot z_k)$ and check that $r \cdot u - v = 0$.

Since the z_ks and their MACs are shared across the parties, they need to run a secure protocol for carrying-out this check. Thus, the parties will compute the above linear combination over shared values. This is where $\mathcal{F}^{\text{add}}_{\ell-\text{mult}}$ becomes handy, as it allows the parties to compute $[\![u]\!] = \sum_{k=1}^{n} [\![\alpha_k]\!] \cdot [\![z_k]\!]$ and $[\![v]\!] = \sum_{k=1}^{n} [\![\alpha_k]\!] \cdot [\![r \cdot z_k]\!]$ efficiently. The verification protocol is formally described in Protocol 5.4.

The communication cost of the verification protocol is *constant*: the parties call $\mathcal{F}_{\text{rand}}$ once, $\mathcal{F}^{\text{add}}_{\ell-\text{mult}}$ three times (with $\ell = n$) and finally they run the reveal procedure. Since $\mathcal{F}_{\text{rand}}$ can be realized without any communication and the cost of $\mathcal{F}^{\text{add}}_{\ell-\text{mult}}$ and reveal is constant. Thus, the overall cost is a small constant, which is independent of n (the number of shares to verify). This property is highly important in our protocol, since we call the verification protocol before every revealing of a permutation.

Protocol 5.4 (Secure Verification Protocol)**:**

```
Notation: $\beta \leftarrow \text{Verify}\left([\![r]\!], \{[\![z_k]\!]\}_{k=1}^{n}, \{[\![r \cdot z_k]\!]\}_{k=1}^{n}\right)$.
Input: $[\![r]\!], [\![z_1]\!], \ldots, [\![z_n]\!]$ and $[\![r \cdot z_1]\!], \ldots, [\![r \cdot z_n]\!]$.
Output: $\beta \in \{\text{accept}, \text{reject}\}$
  1: The parties call $\mathcal{F}_{\text{rand}}$ to receive $[\![\alpha_1]\!], \ldots, [\![\alpha_n]\!]$.
  2: The parties call $\mathcal{F}^{\text{add}}_{\ell-\text{mult}}$ with $\ell = n$ twice to compute $[\![u]\!] = \sum_{k=1}^{n}[\![\alpha_k]\!] \cdot [\![z_k]\!]$ and $[\![v]\!] = \sum_{k=1}^{n}[\![\alpha_k]\!] \cdot [\![r \cdot z_k]\!]$.
  3: The parties call $\mathcal{F}^{\text{add}}_{\ell-\text{mult}}$ with $\ell = 1$ to compute $[\![r]\!] \cdot [\![u]\!]$.
  4: The parties locally compute $[\![w]\!] = [\![r]\!] \cdot [\![u]\!] - [\![v]\!]$.
  5: The parties securely check whether $w = 0$ or not. If $w = 0$, then
     the parties set $\beta$ = accept. Otherwise, they set $\beta$ = reject.
  6: return $\beta$
```

5.2 Secure Sorting with Malicious Security

We are now ready to present the protocol for secure sorting in the presence of one malicious party. Essentially, we begin by generating some shared random $[\![r]\!]$ using $\mathcal{F}_{\text{rand}}$. Then, for each input key $[\![k_i]\!]$ and its bit decomposition $[\![k_i]\!] = [\![k_i^{(1)}]\!], \ldots, [\![k_i^{(\ell_k)}]\!]$, we call $\mathcal{F}^{\text{add}}_{\ell-\text{mult}}$ with $\ell = 1$ to obtain $[\![r \cdot k_i^{(1)}]\!], \ldots, [\![r \cdot k_i^{(\ell_k)}]\!]$. We then follow the steps of protocol GENPERM while maintaining the invariant in which for each shared value $[\![x]\!]$ we also compute $[\![rx]\!]$. Moreover, we use $\mathcal{F}^{\text{add}}_{\ell-\text{mult}}$ with $\ell = 1$ instead of $\mathcal{F}_{\text{mult}}$, and $\mathcal{F}^{\text{add}}_{\text{reshare-shuffle}}$ in each iteration of Protocol 3.2 (SHUFFLE). Moreover, before revealing each secret we perform a verification step. To elaborate further:

(1) *Input randomization:* Given an input $[\![\vec{k}]\!] = [\![k_1]\!], \ldots, [\![k_m]\!]$ and the bit decomposition $[\![k_i]\!] = [\![k_i^{(1)}]\!], \ldots, [\![k_i^{(\ell_k)}]\!]$ for each $i \in [m]$:
  (a) The parties call $\mathcal{F}_{\text{rand}}$ to receive $[\![r]\!]$.
  (b) For each shared input $[\![k_j^{(i)}]\!]$, the parties call $\mathcal{F}^{\text{add}}_{\ell-\text{mult}}$ with $\ell = 1$ on $[\![k_j^{(i)}]\!]$ and $[\![r]\!]$ to receive $[\![r \cdot k_j^{(i)}]\!]$

  Denote the output of this step by $([\![\vec{k}]\!], [\![r \cdot \vec{k}]\!])$.

(2) *Generating a stable sorting permutation:* The parties run GENPERM (Protocol 4.4) on $[\![\vec{k}]\!]$ and $[\![r \cdot \vec{k}]\!]$ with the following modifications:
  (a) Replace each call to $\mathcal{F}_{\text{mult}}$ with $\mathcal{F}^{\text{add}}_{\ell-\text{mult}}$ with $\ell = 1$.
  Then, on input $([\![a]\!], [\![r \cdot a]\!]), ([\![b]\!], [\![r \cdot b]\!])$, the parties:
    (i) Call $\mathcal{F}^{\text{add}}_{\ell-\text{mult}}$ (with $\ell = 1$) on $[\![a]\!]$ and $[\![b]\!]$ to receive $[\![a \cdot b]\!]$.
    (ii) Call $\mathcal{F}^{\text{add}}_{\ell-\text{mult}}$ (with $\ell = 1$) on $[\![r \cdot a]\!]$ and $[\![b]\!]$ to receive $[\![r \cdot (a \cdot b)]\!]$.
  (b) Whenever there is a call to SHUFFLE (Protocol 3.2): Then, on input $\langle\!\langle \pi \rangle\!\rangle, ([\![\vec{a}]\!], [\![r \cdot \vec{a}]\!])$, the parties:
    (i) Run SHUFFLE on $\langle\!\langle \pi \rangle\!\rangle$ and $[\![\vec{a}]\!]$ to receive $[\![\vec{b}]\!]$, while calling $\mathcal{F}^{\text{add}}_{\text{reshare-shuffle}}$ in each iteration i.
    (ii) Run SHUFFLE on $\langle\!\langle \pi \rangle\!\rangle$ and $[\![r \cdot \vec{a}]\!]$ to receive $[\![r \cdot \vec{b}]\!]$, while calling $\mathcal{F}^{\text{add}}_{\text{reshare-shuffle}}$ in each iteration i.
  (c) Whenever there is a call reveal($[\![\vec{a}]\!]$), the parties first run the verification protocol:
  Let $([\![z_1]\!], [\![r \cdot z_1]\!]), \ldots, ([\![z_n]\!], [\![r \cdot z_n]\!])$ be all the outputs of calls to $\mathcal{F}^{\text{add}}_{\ell-\text{mult}}$ and $\mathcal{F}^{\text{add}}_{\text{reshare-shuffle}}$ since the last time Verify was called.
  Then:

(i) The parties call Verify $\left([\![r]\!], \{[\![z_k]\!]\}_{k=1}^{n}, \{[\![r \cdot z_k]\!]\}_{k=1}^{n}\right)$ (Protocol 5.4) to receive β.

(ii) If β = accept, the parties proceed to run reveal($\vec{a}$). If β = reject, the parties abort the protocol.

Cheating probability. We first show the success cheating probability of our protocol when working over a field $\mathbb{F}$ instead of a ring $\mathcal{R}$ as in the semi-honest case. We prove Lemma 5.5 in App. B:

Lemma 5.5. *If the corrupted party sends any $\Delta \neq 0$ in any of the calls to $\mathcal{F}_{\ell-\text{mult}}^{\text{add}}$ or $\mathcal{F}_{reshare\text{-}shuffle}^{add}$ before the verification step, then β = accept in the verification (Prot. 5.4) with probability at most $\frac{2}{|\mathbb{F}|}$.*

Security. We are now ready to prove that our protocol computes $\mathcal{F}_{\text{sort}}$ against malicious parties. We remark that for malicious security, we need that $\mathcal{F}_{\text{sort}}$, as described in Functionality 4.5, will also hand the ideal world adversary the corrupted party's input shares (these are anyway known to the adversary in a real world execution) and also allow it to choose the corrupted party's shares of the output. We prove the following in Appendix B:

Theorem 5.6. *Let s be a statistical security parameter, and let $\mathbb{F}$ be a finite field such that $\frac{2}{|\mathbb{F}|} \leq 2^{-s}$. Then, our protocol (as described in the text) securely computes $\mathcal{F}_{sort}$ with abort in the ($\mathcal{F}_{rand}$, $\mathcal{F}_{\ell-\text{mult}}^{\text{add}}$, $\mathcal{F}_{reshare\text{-}shuffle}^{add}$)-hybrid model, in the presence of one malicious party.*

Communication cost. Compared to the semi-honest setting (See analysis in Section 4.1), each shuffling or multiplication operation is called *twice* to maintain the MAC.

Moreover, in the semi-honest protocol we used a ring with size at least $[m]$. In the malicious, the size of the field affects the probability of error. For ϵ error, we need a field of size $\approx 1/\epsilon$. Overall, the communication cost per party is bounded by:

$$O\left(m\ell_k \cdot (\log \frac{m}{\epsilon}) + m \cdot (\ell_p + \frac{1}{\epsilon})\right)$$

Remark 5.7 (Small Field and Rings.). *In [27] and [1], it was shown how to extend the compiler of Chida et al. [13] to small fields and the ring $\mathbb{Z}_{2^\ell}$ respectively. Our compiler can be extended to small field or rings in a similar way.*

We note that when using the extensions of [27] or [1] for small fields or rings, the MAC is set to be over a larger field/ring. This is needed to achieve sufficiently small statistical error, and this further increases costs.

6 COMPUTING HEAVY HITTERS

The setting for the heavy hitters problem includes many clients that have private values/strings, and servers that wish to identify the most frequent among these strings. Variants of this problem could compute, for instance, the t most frequent values or, alternatively, the values which are reported by at least a certain percentage of the clients (say, 0.1%). The heavy hitters problem is relevant in many data collection and telemetry scenarios, such as collecting popular URLs, application usage patterns, or other performance data.

The recent work in [12] describes a system, Poplar, for securely computing the items which are reported by at least a certain percentage of the clients. It uses two non-colluding servers, and a technique based on incremental distributed point functions. For values of length n, the servers run n communication rounds with the clients, where the i'th round "zooms in" on strings which begin with prefixes of length $i-1$ that were popular in the previous round.

We suggest using secure sorting for this task. Protocol 6.1 describes how to securely identify, using a sorting protocol, all values which appear at least t times (where the threshold t can be set to any fraction of the total number of values).

Protocol 6.1 uses one secure sort and one secure shuffle, as well as computing simple operations that can be implemented as a circuit of binary gates. The circuit depth is only logarithmic in the length of the values, and is otherwise independent of the number of values. This is vital for performance, since the number of rounds of the secure three-party MPC protocol that we use is determined by the depth of the circuit. The implementation and the performance of this protocol are described in Section 7.3.

Secure computation of other variants of the heavy hitters problem, outputting the number of appearances of each heavy hitter, or outputting the s most popular items regardless of their individual counts, can be constructed in a similar way. We will give a description of those in Appendix C. The overhead of these variants is similar to that of Protocol 6.1.

Protocol 6.1 (Computing heavy hitters)**:**

Input: A secret-shared vector $[\![\vec{v}]\!]$ of the values of the clients. A threshold t.

Output: All values which appear at least t times.

```
1: Sort the elements of $[\![\vec{v}]\!]$. (As a result, all identical values are moved
   to be adjacent to each other.)
2: Compute a shared vector of bits $[\![\vec{u}]\!]$, where $u_i = 1$ iff $v_i = v_{i-t+1}$.
   ($u_i$ is 1 iff there are at least $t$ occurrences of $v_i$ up to location $i$.)
3: Compute a shared vector of bits $[\![\vec{f}]\!]$, where $f_i = 1$ iff $v_i \neq v_{i+1}$.
   ($f_i$ is 1 iff $v_i$ is the last in a sequence of identical values.)
4: Compute a secret-shared vector $[\![\vec{w}]\!]$, where $w_i = u_i \cdot f_i \cdot v_i$. (If a
   value $v_i$ occurs at least $t$ times, then there is exactly one location
   in $\vec{w}$ where $v_i$ appears. Otherwise, $v_i$ does not appear in $\vec{w}$.)
5: Shuffle the vector $[\![\vec{w}]\!]$ and open it.
```

Client communication and security: Protocols for computing heavy hitters receive inputs from many clients and must therefore be secure against *malicious clients* that might want to disrupt the computation. The mechanism for ensuring this in the Poplar system is quite intricate, since that system uses distributed point functions and multiple interactions with each client. In our construction each client simply shares a single value with the servers. Corrupt clients can be identified and eliminated by verifying that this sharing is correct (namely, that it is a replicated sharing where each pair of servers receives two copies of the same share.)

Security against malicious servers: Our construction can be implemented using existing protocols, to have security against *malicious servers.* Moreover, even though we only describe in this work the implementation for the case of three servers with an honest majority, the template of Protocol 6.1 can be used with any number of servers, using appropriate protocols for sorting and for computing circuits. This is in contrast to solutions based on distributed point functions which are only supported in a setting with two servers.

7 IMPLEMENTATION AND EXPERIMENTS

We describe in this section the experimental measurements of our implementations of the sorting and heavy hitters protocols.

7.1 Settings

Hardware. Our experiments run all three servers on AWS c5.18xlarge machines. in the same region. Each server has 72 vCPUs and 144 GB of memory. All servers are connected over a 25Gb network, but in the experiment we limit the communication bandwidth to 10Gb for comparison[3].

Implementation. We implemented the protocols in C++11, on CentOS 7.2.1511, using GCC 4.8.5. Our implementation utilized asynchronous processing, multi-threading, pipelining of CPU and network, and fast implementation of low-level cryptographic operations such as a pseudorandom function. We utilized the extended instructions of AES-NI, RdRand, and SSE4. Especially, pseudorandom secret sharing (PRSS) with AES was used as a PRF for $\mathcal{F}_{\text{rand}}$.

Domain of input/output shares. The input and output of our protocols are shares in $\mathbb{Z}_p$, where $p = 2^{61} - 1$ (a Mersenne prime, enabling efficient computation). Since our sorting protocol requires that the keys are shared bit-wise, we first bit-decompose the key shares, using a technique from [28].

Statistical security parameter. The protocols with full security against malicious behavior were implemented with the security parameter λ. Here, λ represents the size of the field $\mathbb{F}_{2^\lambda}$ used in the MAC for a small field. We set λ to values of 8, 32, and 64, limiting the cheating probability to $2/2^8 = 2^{-7}, 2/(2^{31}-1) \approx 2^{-30}$ and $2/(2^{-61}-1) \approx 2^{-60}$, respectively[4].

7.2 Sorting Experiments

We present in Table 7.1 the run times of applying our sorting protocol to sort records based on 8-bit and 32-bit keys. The records have value (payloads) of length 32 bits. We measure the average run time of the three servers after the clients complete the sharing of their values. The measurements include the time it takes to decompose the client shares into shares of bits, the time of computing the permutation that sorts the records, and the time of applying this permutation to the records. We also present the total time of computing the permutation and applying it (it would be appropreate if the input is bitwise shared to begin with), and (in the last column) the total time of the protocol including bit decomposition.

The first thing to note is that the run time of the protocol is quite fast. For example, sorting one million records with keys of length 8 or 32 bits takes 0.86 or 2.26 seconds with semi-honest security, and only 1.84 or 5.89 seconds with full security when the cheating probability is limited to 2^{-30} ($\lambda = 32$).

The run time of the protocol with semi-honest security is described in the first part of the table ("security bits = 0"). The following parts of the table set the statistical security parameter to $\lambda = 8, 32$ and 64 bits. The run times with $\lambda = 8$ are only 1.72-2.05 times larger than with semi-honest security. The run times with $\lambda = 32$ are 2.15-2.63 times larger than with semi-honest security, and the run times with $\lambda = 64$ are 3.60-5.09 times larger.

The time of computing the permutation that sorts the records is expected to be linear in the key length. Therefore, theoretically the run time for computing the permutation with 32 bit keys should be 4 larger than this time when using 8 bit keys (with all other parameters being the same). In the experiments, this ratio is in the range 2.66-3.77. As expected, the time of applying the permutation is independent of the key length, and is almost the same for measurements with key length of length 8 or 32.

Table 7.2 compares our results with the reported run times of the secure quicksort protocol of [4], for the case of 32 bit sorting keys and 32 bit payloads. The protocols were measured in comparable settings. Our protocol performs substantially faster, except for the case of 10M records and semi-honest security. The gain in performance in the case of full security against malicious adversaries is higher than the gain in the semi-honest case.

sec. bits	key len	rec-ords	Bit De-comp	Gen Perm	Apply Perm	Gen + Apply	**TOTAL** BD+G+A
0	8	1M	67	685	105	790	857
		5M	131	3,455	622	4,078	4,209
		10M	375	8,264	1,402	9,666	10,041
	32	1M	90	2,084	87	2,171	2,261
		5M	367	13,947	650	14,597	14,964
		10M	804	34,922	1,434	36,355	37,159
8	8	1M	337	1,020	193	1,212	1,549
		5M	1,131	6,356	1,155	7,512	8,643
		10M	2,388	14,321	2,683	17,004	19,392
	32	1M	552	3,575	166	3,742	4,294
		5M	2,833	23,639	1,134	24,773	27,605
		10M	6,291	54,947	2,688	57,636	63,926
32	8	1M	710	969	161	1,130	1,840
		5M	3,480	6,330	1,240	7,570	11,050
		10M	7,146	14,046	2,693	16,740	23,886
	32	1M	1,825	3,563	197	3,760	5,585
		5M	10,846	24,587	1,236	25,822	36,668
		10M	24,731	58,273	2,697	60,971	85,702
64	8	1M	1,071	1,880	137	2,017	3,088
		5M	6,574	12,461	1,209	13,670	20,244
		10M	15,290	28,944	2,822	31,766	47,056
	32	1M	3,418	7,073	153	7,225	10,644
		5M	23,241	51,684	1,297	52,981	76,222
		10M	51,834	114,604	2,905	117,509	169,343

Table 7.1: Run times in milliseconds for the different parts of the protocol. Columns include the security parameter λ for full security (0 is semi-honest), the length of key used for sorting, # of records, and the times for bit decomposition, computing the permutation that sorts the records, applying the permutation, and different totals.

7.3 Private Heavy Hitter

We implemented Protocol 6.1 for finding heavy hitters and measured it in our test environment; see the results in Table 7.3. We compare its performance to that of the Poplar system [12]. Both our

[3]We used c5.18xlarge instances for supporting the memory requirements, but we confirmed that runtimes were comparable when running on c5.9xlarge instances.

[4] We store an element of $\vec{\sigma}$ in $\mathbb{F}_p$, where $p = 2^{61} - 1$ when $\lambda = 64$ and $p = 2^{31} - 1$ otherwise. The cheating probability is then $\max(2/2^\lambda, 2/p)$.

	Size	Our protocol	[4]	Ratio
Semi-honest	1M	2.261	5.016	0.45
	5M	14.964	15.689	0.95
	10M	37.159	29.157	1.27
Malicious	1M	5.585	33.337	0.17
	5M	36.668	96.771	0.38
	10M	85.702	202.724	0.42

Table 7.2: Run times in seconds for sorting records with 32-bit keys and 32-bit values. Data size is in multiples of 2^{20} records. The run times of our protocol are compared to the run time of [4].

Size	Server time (semi-honest)	Server time (malicious)	Ratio: malicious / semi-honest
100,000	1.556	6.286	4.04
200,000	2.629	8.843	3.36
400,000	5.189	14.896	2.87

Table 7.3: Run times in seconds for the finding heavy hitters among 256-bit strings. The columns show the run times for the semi-honest and malicious protocol, and the ratio between the two run times

experiments and Poplar's experiments worked with 256-bit strings and searched for values appearing at least 0.1% of the time. The run time is measured starting after our servers receive the shares of the clients, and after the Poplar servers receive the last incremental distributed point function keys from the clients.

The Poplar system was implemented in Rust, and its experiments were done using two AWS c4.8xlarge servers (32 virtual cores) which were located in the east and west coast regions, with 61.8msec latency. Our servers were located in the same data center, with a latency of 1-2msec.

The run times of Poplar for input sizes of 100,000, 200,000 and 400,000 items, were 828.1, 1,633 and 3,226 seconds, respectively. These times are about 500-600 times slower than the our semi-honest implementation, and about 130-200 times slower than our malicious implementation. (Poplar provides partial security against a malicious server, and therefore we compare it to the two security levels of our implementation.) The differences in latency between the Poplar testbed and ours should account for only a small fraction of the difference in the run times.

8 CONCLUSION

We proposed a novel three-party sorting protocol secure against passive adversaries in the honest majority setting. The new sorting protocol is based on radix sort and therefore it is stable. It is asymptotically better compared to previous sorting protocols since it does not need to shuffle the entire length of the elements after each comparison step. We also proposed novel protocols and optimizations that reduce about 85% of communication.

We implemented our protocol with those optimizations. Our experiments show that the resulting sorting protocol is considerably faster than the currently most efficient sorting protocol [4]. Furthermore, efficient sorting enables new data analysis applications on large datasets. We demonstrate this by showing an extremely efficient solution to the heavy hitters problem.

ACKNOWLEDGMENTS

Sponsored in part by the Israel Science Foundation (grant No. 2439/20), by JPM Faculty Research Award, by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office, and by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 891234.

REFERENCES

[1] Mark Abspoel, Anders P. K. Dalskov, Daniel Escudero, and Ariel Nof. 2021. An Efficient Passive-to-Active Compiler for Honest-Majority MPC over Rings. In *Applied Cryptography and Network Security - 19th International Conference, ACNS 2021, Kamakura, Japan, June 21-24, 2021, Proceedings, Part II (Lecture Notes in Computer Science)*, Kazue Sako and Nils Ole Tippenhauer (Eds.), Vol. 12727. Springer, 122–152. https://doi.org/10.1007/978-3-030-78375-4_6

[2] Miklós Ajtai, János Komlós, and Endre Szemerédi. 1983. An $O(n \log n)$ Sorting Network. In *STOC*. 1–9. https://doi.org/10.1145/800061.808726

[3] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. 2016. High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority. In *CCS*. 805–817. https://doi.org/10.1145/2976749.2978331

[4] Toshinori Araki, Jun Furukawa, Kazuma Ohara, Benny Pinkas, Hanan Rosemarin, and Hikaru Tsuchida. 2021. Secure Graph Analysis at Scale. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, 2021*. ACM, 610–629.

[5] Gilad Asharov, T.-H. Hubert Chan, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. 2020. Bucket Oblivious Sort: An Extremely Simple Oblivious Sort. In *3rd Symposium on Simplicity in Algorithms, SOSA 2020, Salt Lake City, UT, USA, January 6-7, 2020*. SIAM, 8–14.

[6] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. 2020. OptORAMa: Optimal Oblivious RAM. In *Advances in Cryptology - EUROCRYPT 2020 (Lecture Notes in Computer Science)*, Vol. 12106. Springer, 403–432.

[7] Kenneth E. Batcher. 1968. Sorting Networks and Their Applications. In *American Federation of Information Processing Societies: AFIPS*, Vol. 32. Thomson Book Company, Washington D.C., 307–314.

[8] Gabrielle Beck, Aarushi Goel, Abhishek Jain, and Gabriel Kaptchuk. 2021. Order-C Secure Multiparty Computation for Highly Repetitive Circuits. In *Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part II (Lecture Notes in Computer Science)*, Anne Canteaut and François-Xavier Standaert (Eds.), Vol. 12697. Springer, 663–693.

[9] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. 1988. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*. ACM, 1–10.

[10] Marina Blanton and Everaldo Aguiar. 2012. Private and Oblivious Set and Multiset Operations. In *ASIACCS '12*. ACM, New York, NY, USA, 40–41. https://doi.org/10.1145/2414456.2414479

[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] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. 2021. Lightweight Techniques for Private Heavy Hitters. (2021), 762–776.

[13] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. 2018. Fast Large-Scale Honest-Majority MPC for Malicious Adversaries. In *CRYPTO 2018*. 34–64. https://doi.org/10.1007/978-3-319-96878-0_2

[14] Arka Rai Choudhuri, Aarushi Goel, Matthew Green, Abhishek Jain, and Gabriel Kaptchuk. 2021. Fluid MPC: Secure Multiparty Computation with Dynamic Participants. In *Advances in Cryptology - CRYPTO 2021 - 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16-20, 2021, Proceedings, Part II (Lecture Notes in Computer Science)*, Tal Malkin and Chris Peikert (Eds.), Vol. 12826. Springer, 94–123.

[15] Henry Corrigan-Gibbs and Dan Boneh. 2017. Prio: Private, Robust, and Scalable Computation of Aggregate Statistics. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI*. USENIX Association, 259–282.

[16] Ronald Cramer, Ivan Damgård, and Yuval Ishai. 2005. Share Conversion, Pseudorandom Secret-Sharing and Applications to Secure Computation. In *TCC (LNCS)*, Joe Kilian (Ed.), Vol. 3378. Springer, 342–362.

[17] Daniel Genkin, Yuval Ishai, Manoj Prabhakaran, Amit Sahai, and Eran Tromer. 2014. Circuits resilient to additive attacks with applications to secure computation. In *Symposium on Theory of Computing, STOC 2014*. ACM, 495–504.

[18] Oded Goldreich. 2004. *The Foundations of Cryptography - Volume 2, Basic Applications.* Cambridge University Press.

[19] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *STOC*. ACM, 218–229.

[20] Michael T. Goodrich. 2010. Randomized Shellsort: A Simple Oblivious Sorting Algorithm. In *SODA*. 1262–1277.

[21] Michael T. Goodrich. 2014. Zig-zag sort: a simple deterministic data-oblivious sorting algorithm running in O(n log n) time. In *Symposium on Theory of Computing, STOC 2014*. ACM, 684–693.

[22] Michael T. Goodrich. 2014. Zig-zag sort: a simple deterministic data-oblivious sorting algorithm running in O(n log n) time. In *Symposium on Theory of Computing, STOC 2014*. ACM, 684–693.

[23] Koki Hamada, Ryo Kikuchi, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. 2012. Practically Efficient Multi-party Sorting Protocols from Comparison Sort Algorithms. In *ICISC*. 202–216.

[24] Y. Huang, D. Evans, J. Katz, and L. Malka. 2011. Faster Secure Two-Party Computation Using Garbled Circuits. In *USENIX Security'11*. USENIX, 539–554.

[25] Mitsuru Ito, Akira Saito, and Takao Nishizeki. 1989. Secret sharing scheme realizing general access structure. *Electronics and Communications (Part III: Fundamental Electronic Science)* 72, 9 (1989), 56–64.

[26] Kristján Valur Jónsson, Gunnar Kreitz, and Misbah Uddin. 2011. Secure Multi-Party Sorting and Applications. In *ACNS*. Springer.

[27] Ryo Kikuchi, Nuttapong Attrapadung, Koki Hamada, Dai Ikarashi, Ai Ishida, Takahiro Matsuda, Yusuke Sakai, and Jacob C. N. Schuldt. 2019. Field Extension in Secret-Shared Form and Its Applications to Efficient Secure Computation. In *ACISP*, Vol. 11547. Springer, 343–361.

[28] Ryo Kikuchi, Dai Ikarashi, Takahiro Matsuda, Koki Hamada, and Koji Chida. 2018. Efficient Bit-Decomposition and Modulus-Conversion Protocols with an Honest Majority. In *ACISP 2018*. 64–82. https://doi.org/10.1007/978-3-319-93638-3_5

[29] Vladimir Kolesnikov, Naor Matania, Benny Pinkas, Mike Rosulek, and Ni Trieu. 2017. Practical Multi-party Private Set Intersection from Symmetric-Key Techniques. In *CCS 2017*. 1257–1272. https://doi.org/10.1145/3133956.3134065

[30] Sven Laur, Jan Willemson, and Bingsheng Zhang. 2011. Round-Efficient Oblivious Database Manipulation. In *ISC 2011*. 262–277.

[31] Phi Hung Le, Samuel Ranellucci, and S. Dov Gordon. 2019. Two-party Private Set Intersection with an Untrusted Third Party. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM, 2403–2420. https://doi.org/10.1145/3319535.3345661

[32] Frank Thomson Leighton, Yuan Ma, and Torsten Suel. 1995. On Probabilistic Networks for Selection, Merging, and Sorting. In *7th Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '95*. ACM, 106–118.

[33] Wei-Kai Lin, Elaine Shi, and Tiancheng Xie. 2019. Can We Overcome the n log n Barrier for Oblivious Sorting?. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019*, Timothy M. Chan (Ed.). SIAM, 2419–2438.

[34] B. Pinkas, T. Schneider, G. Segev, and M. Zohner. 2015. Phasing: Private Set Intersection Using Permutation-based Hashing. In *USENIX Security'15*. USENIX, 515–530.

[35] Benny Pinkas, Thomas Schneider, Christian Weinert, and Udi Wieder. 2018. Efficient Circuit-Based PSI via Cuckoo Hashing. In *EUROCRYPT*. 125–157.

[36] Adi Shamir. 1979. How to share a secret. *Commun. ACM* 22, 11 (1979), 612–613.

A IMPROVING CONCRETE EFFICIENCY

In this section, we introduce several optimization techniques to improve concrete efficiency of our basic sorting protocol described in Section 4. The techniques in Appendix A.1 are applicable to our protocols with both semi-honest security and malicious security while those in Appendix A.2 are applicable to only our protocol with semi-honest security.

A.1 An Optimized Sorting Protocol

A.1.1 Extended Multiplication Protocol. The multiplication protocol can be extended to an inner-product protocol that on input $([a_1], \ldots, [a_\ell])$ and $([b_1], \ldots, [b_\ell])$ outputs $[c]$, where $c = \sum_{j=1}^{\ell} a_i b_i$ [13]. We present the extended functionality $\mathcal{F}_{\ell\text{-mult}}$ in Functionality A.1. We can obtain the protocol for $\mathcal{F}_{\ell\text{-mult}}$ by extending the efficient three party semi-honest multiplication protocol of [3]. The communication and round complexities are the same as the multiplication protocol. We use $\mathcal{F}_{\ell\text{-mult}}$ in our optimized protocol.

Functionality A.1 ($\mathcal{F}_{\ell\text{-mult}}$ – Extended Multiplication)**:**

Let P_i be the corrupted party. Upon receiving from the honest parties their shares of $\vec{a} = (a_1, \ldots, a_\ell)$ and $\vec{b} = (b_1, \ldots, b_\ell)$, and an input (c_i, c_{i+1}) from P_i, $\mathcal{F}_{\ell\text{-mult}}$ reconstructs $(\vec{a}, \vec{b})$, computes $c = \sum_i^{\ell} = 1 a_i b_i$, computes $c_{i-1} = c - (c_i + c_{i+1})$ and send the honest parties their shares.

A.1.2 Batch Processing of Multiple Bits. Our first optimization is to improve both communication and round complexities of GenPerm (Protocol 4.4) by processing multiple bits of keys at a time. In Protocol 4.4, recall that we iteratively compute a secret-shared permutation at step 4 to 7, where the permutation $\vec{\sigma}_j$ obtained after the j-th iteration represents the stable sort by the lower j bits of the key. Roughly speaking, the number of the iterations of computing the permutation can be reduced to $1/L$ by processing L bits at a time.

We extend Protocol 4.1, which can handle only single bit keys, to an algorithm that can handle L-bit keys as shown in Protocol A.2. In the algorithm description, $\vec{1}$ denotes m-dimensional vector with all elements being 1, and $\bigodot$ denotes element-wise multiplication. The number of calls of $\mathcal{F}_{\text{mult}}$ to compute $[\![\vec{f}^{(j)}]\!]$ for $0 \le j \le 2^L - 1$ can be $m(2^L - L - 1)$ by precomputing all terms appearing in $\vec{f}^{(j)}$.[5] Since the $\mathcal{F}_{\ell\text{-mult}}$ requires the same communication cost as $\mathcal{F}_{\text{mult}}$, the total communication complexity is the same as $m(2^L - L)$ invocations of multiplications.

Although the round complexity gets better for larger L, too large L causes worse communication complexity. We evaluate the communication complexity and show that $L = 3$ was reasonable in Section A.3.

A.1.3 Reusing permutation in shuffling. In each iteration of step 5 to 7 in Protocol 4.4, the parties first generate a random permutation π and apply it to $\vec{\sigma}_{j-1}$ in both ApplyPerm and Compose. Our observation is that we can use the same random permutation π in ApplyPerm and Compose for each iteration. Then, we can skip Steps 1 to 2 in Protocol 4.3. To use this optimization, we modify these algorithms as OptApplyPerm (Protocol A.3) and OptCompose (Protocol A.4).

A.1.4 Permuting Small Shares. Finally, recall that the reason we use a large ring $\mathcal{R}$ in Protocol 4.4 to secret-share a bit string $\vec{k}^{(i)}$ is to compute a ring element s in GenBitPerm (Protocol 4.1). On the other hand, observe that ApplyPerm works even if the input secret shares of $\vec{k}^{(i)}$ are generated in $\mathbb{Z}_2$. Hence, we can reduce the communication complexity of ApplyPerm by applying it to shares in $\mathbb{Z}_2$. This changes the communication complexity of the corresponding shuffling protocol in ApplyPerm from $2m||\mathcal{R}||$ to $2m$ bits. After that, we can convert the shares in $\mathbb{Z}_2$ to those in $\mathcal{R}$ before running GenMultiBitSort. This optimization is reflected in Protocol A.3. In the description, we denote secret shares of a in $\mathbb{Z}_2$ by $\langle a \rangle$ and use a modulus conversion protocol [28] as an ideal

[5]Concretely, the parties prepare all the terms with degree greater than 1 that appear in $\vec{f}^{(j)}$. This can be done by $m(2^L - L - 1)$ multiplications since all terms that appear in $\vec{f}^{(j)}$ can be obtained by $\bigodot_{k' \in S} \vec{k}^{(k')}$ for some $S \subseteq \{1, \ldots, L\}$. Then, the parties can compute $\vec{f}^{(j)}$ from a linear combination of the prepared terms.

Protocol A.4 (Optimized composition of two permutations)**:**

Notation: $[\![\vec{\sigma}']\!] \leftarrow$ OptCompose($[\![\vec{\sigma}]\!], [\![\vec{\rho}]\!], \langle\!\langle \pi \rangle\!\rangle, \sigma''$).
Input: Secret-shared two permutations ($[\![\vec{\sigma}]\!], [\![\vec{\rho}]\!]$).
Output: The secret-shared permutation $[\![\vec{\sigma}']\!]$, where $\sigma'^{-1} = \sigma^{-1} \circ \rho^{-1}$.

```
1: The parties apply $\sigma''$ to $[\![\vec{\rho}]\!]$ and obtain $[\![\vec{\rho}']\!]$.
2: $[\![\vec{\sigma}']\!] \leftarrow$ Unshuffle($\langle\!\langle \pi \rangle\!\rangle; [\![\vec{\rho}']\!]$).
3: return $[\![\vec{\sigma}']\!]$.
```

Protocol A.5 (Optimized permutation generation of stable sort)**:**

Notation: $[\![\vec{\sigma}]\!] \leftarrow$ OptGenPerm($\langle \vec{k} \rangle$).
Input: Secret-shared keys $\langle \vec{k} \rangle = (\langle k_1 \rangle, \ldots, \langle k_m \rangle)$ where $\langle k_i \rangle = (\langle k_i^{(1)} \rangle, \ldots, \langle k_i^{(\ell_k)} \rangle)$ and $\langle \vec{k}^{(j)} \rangle = (\langle k_1^{(j)} \rangle, \ldots, \langle k_m^{(j)} \rangle)$.
Output: The secret-shared permutation $[\![\vec{\sigma}]\!]$ such that σ is the stable sorting of $\vec{k}$.

```
1: The parties send $(\langle \vec{k}^{(1)} \rangle, \ldots, \langle \vec{k}^{(L)} \rangle)$ to $\mathcal{F}_{\text{modconv}}$, and receive ($[\![\vec{k}^{(1)}]\!], \ldots, [\![\vec{k}^{(L)}]\!]$).
2: $[\![\vec{\rho}_1]\!] \leftarrow$ GenMultiBitSort($[\![\vec{k}^{(1)}]\!], \ldots, [\![\vec{k}^{(L)}]\!]$).
3: $[\![\vec{\sigma}_1]\!] := [\![\vec{\rho}_1]\!]$.
4: for $j = 2$ to $\widehat{\ell}$ do
5:     $\left([\![\vec{k}'^{((j-1)L+1)}]\!], \ldots, [\![\vec{k}'^{(jL)}]\!], \langle\!\langle \pi \rangle\!\rangle, \sigma''_{j-1}\right) \leftarrow$ OptApplyPerm$\left([\![\vec{\sigma}_{j-1}]\!]; \langle \vec{k}^{((j-1)L+1)} \rangle, \ldots, \langle \vec{k}^{(jL)} \rangle\right)$
6:     $[\![\vec{\rho}_j]\!] \leftarrow$ GenMultiBitSort($[\![\vec{k}'^{((j-1)L+1)}]\!], \ldots, [\![\vec{k}'^{(jL)}]\!]$).
7:     $[\![\vec{\sigma}_j]\!] \leftarrow$ OptCompose($[\![\vec{\sigma}_{j-1}]\!], [\![\vec{\rho}_j]\!], \langle\!\langle \pi \rangle\!\rangle, \sigma''_{j-1}$).
8: return $[\![\vec{\sigma}_{\widehat{\ell}}]\!]$.
```

Protocol A.2 (Generating permutation of stable sort for multiple bits)**:**

Notation: $[\![\vec{\rho}]\!] \leftarrow$ GenMultiBitSort($[\![\vec{k}^{(1)}]\!], \ldots, [\![\vec{k}^{(L)}]\!]$).
Input: Secret-shared L vectors of keys $[\![\vec{k}^{(1)}]\!], \ldots, [\![\vec{k}^{(L)}]\!]$.
Output: Shares of the permutation $[\![\vec{\rho}]\!]$ of the stable sorting by $(\vec{k}^{(1)}, \ldots, \vec{k}^{(L)})$

```
1: for $j = 0$ to $2^L - 1$ do
2:     Regard $j$ as an $L$-bit element $j = b^{(L)} || \cdots || b^{(1)}$.
3:     for $k' = 1$ to $L$ do
4:         The parties locally compute $[\![\vec{d}_{k'}]\!] := [\![b^{(k')}\vec{k}^{(k')} + (1 - b^{(k')})(\vec{1} - \vec{k}^{(k')})]\!]$.
5:     The parties compute $[\![\vec{f}^{(j)}]\!] := [\![\bigodot_{i=1}^{L} \vec{d}_i]\!]$ by using $\mathcal{F}_{\text{mult}}$.
6: $[\![s]\!] := [\![0]\!]$
7: for $j = 0$ to $2^L - 1$ do
8:     for $i = 1$ to $m$ do
9:         $[\![s]\!] := [\![s]\!] + [\![f_i^{(j)}]\!]$.
10:        $[\![s_i^{(j)}]\!] := [\![s]\!]$.
11: for $1 \le i \le m$ do in parallel
12:     Compute $[\![\rho(i)]\!]$ by sending $([\![f_i^{(0)}]\!], \ldots, [\![f_i^{(2^L-1)}]\!])$ and $([\![s_i^{(0)}]\!], \ldots, [\![s_i^{(2^L-1)}]\!])$ to $\mathcal{F}_{\ell\text{-mult}}$ with $\ell = 2^L$.
13: return $[\![\vec{\rho}]\!] = ([\![\rho(1)]\!], \ldots, [\![\rho(m)]\!])$.
```

functionality that change secret shares in $\mathbb{Z}_2$ to those in $\mathcal{R}$. We denote the functionality by $\mathcal{F}_{\text{modconv}}$.

A.1.5 Putting it All Together. Applying these optimizations described above to GenPerm (Protocol 4.4), we obtain our optimized sorting protocol (OptGenPerm) as described in Protocol A.5, which uses Protocols A.2 to A.4 as a subroutine. Note that $\widehat{\ell} := \lceil \frac{\ell_k}{L} \rceil$ in the description of OptGenPerm.

Protocol A.3 (Optimized application of a permutation)**:**

Notation: $\left([\![\vec{k}'^{(1)}]\!], \ldots, [\![\vec{k}'^{(L)}]\!], \langle\!\langle \pi \rangle\!\rangle, \sigma''\right) \leftarrow$ OptApplyPerm$\left([\![\vec{\sigma}]\!]; \langle \vec{k}^{(1)} \rangle, \ldots, \langle \vec{k}^{(L)} \rangle\right)$.
Input: A secret-shared permutation $[\![\vec{\sigma}]\!]$ and a secret-shared vector $\langle \vec{k}^{(1)} \rangle, \ldots, \langle \vec{k}^{(L)} \rangle$.
Output: The secret-shared vector $[\![\vec{k}'^{(1)}]\!], \ldots, [\![\vec{k}'^{(L)}]\!]$ such that $\vec{k}'^{(i)} = \sigma(\vec{k}^{(i)})$ for $1 \le i \le L$, a shared permutation $\langle\!\langle \pi \rangle\!\rangle$, and a permutation σ'' such that $\pi \circ \sigma$.

```
1: The parties call $\mathcal{F}_{\text{rand}}$ and receive $\langle\!\langle \pi \rangle\!\rangle$.
2: $[\![\vec{\sigma}'']\!] \leftarrow$ Shuffle($\langle\!\langle \pi \rangle\!\rangle; [\![\vec{\sigma}]\!]$).
3: The parties reveal $[\![\vec{\sigma}'']\!]$ and obtain $\vec{\sigma}''$.
4: for $1 \le i \le L$ (in parallel) do
5:     $\langle \vec{k}''^{(i)} \rangle \leftarrow$ Shuffle($\langle\!\langle \pi \rangle\!\rangle; \langle \vec{k}^{(i)} \rangle$).
6:     The parties send $\langle \vec{k}''^{(i)} \rangle$ to $\mathcal{F}_{\text{modconv}}$, and receive $[\![\vec{k}^{(i)}]\!]$.
7:     The parties apply $\sigma''$ with $[\![\vec{k}''^{(i)}]\!]$ and obtain $[\![\vec{k}'^{(i)}]\!]$.
8: return $\left([\![\vec{k}'^{(1)}]\!], \ldots, [\![\vec{k}'^{(L)}]\!], \langle\!\langle \pi \rangle\!\rangle, \sigma''\right)$.
```

A.2 Further Optimization for Sorting with Semi-honest Security

In the semi-honest security model, we can further optimize shuffling protocols. The shuffling protocol (Protocol 3.2) invokes the resharing protocol three times, i.e., the communication complexity is $6m||\mathcal{R}||$ in total. We show the optimized shuffling protocol in Protocol A.6 whose communication complexity is $4m||\mathcal{R}||$ bits in total.

Recall that the shuffling protocol repeatedly permutes shares by π_i three times. The idea is that P_1 knows π_2 and π_1 that used for the first and second shuffling steps, and the randomness used in resharing also can be obtained via $\mathcal{F}_{\text{rand}}$ beforehand. Therefore, P_1 can permute $\vec{a}$ using π_2 and π_1 at once.

Protocol A.6 (Optimized shuffling protocol)**:**

Notation: $[\![\vec{a}']\!] \leftarrow$ OptShuffle($\langle\!\langle \pi \rangle\!\rangle; [\![\vec{a}]\!]$)
Input: A secret-shared vector $[\![\vec{a}]\!]$ and a permutation $\langle\!\langle \pi \rangle\!\rangle$.
Output: The secret-shared shuffled vector $[\![\vec{a}']\!] = [\![\pi\vec{a}]\!]$.

```
1: Let $\langle\!\langle \pi \rangle\!\rangle_i = (\pi_i, \pi_{i+1})$ and $[\![\vec{a}]\!]_i = (\vec{a}_i, \vec{a}_{i+1})$.
2: The parties call $\mathcal{F}_{\text{rand}}$ $m$ times and obtain $(\vec{\alpha}_i, \vec{\alpha}_{i+1})$ for $P_i$.
3: $P_3$ computes $\vec{\gamma} := \pi_1(\vec{a}_3) + \vec{\alpha}_1$ and sends it to $P_2$.
4: $P_1$ computes $\vec{\delta} := \pi_2(\pi_1(\vec{a}_1 + \vec{a}_2) - \vec{\alpha}_1) - \vec{\alpha}_2$ and sends it to $P_3$.
5: $P_2$ computes $\vec{a}''_2 := \pi_3(\pi_2(\vec{\gamma}) + \vec{\alpha}_2)$.
6: $P_3$ computes $\vec{a}''_1 := \pi_3(\vec{\delta})$.
7: Let $\vec{a}''_3 := \vec{0}$ and $[\![\vec{a}']\!] \leftarrow$ reshare($[\![\vec{a}'']\!]$, 1).
8: return $[\![\vec{a}']\!]$
```

First, let us confirm completeness. Observe that

$$\begin{aligned}\vec{a}''_2 &= \pi_3 \circ \pi_2 \circ \pi_1(\vec{a}_3) + \pi_3 \circ \pi_2(\vec{\alpha}_1) + \pi_3(\vec{\alpha}_2) \\ \vec{a}''_1 &= \pi_3 \circ \pi_2 \circ \pi_1(\vec{a}_1 + \vec{a}_2) - \pi_3 \circ \pi_2(\vec{\alpha}_1) - \pi_3(\vec{\alpha}_2).\end{aligned}$$

Therefore,

$$\vec{a}''_1 + \vec{a}''_2 + \vec{a}''_3 = \pi(\vec{a}).$$

and $[\![\vec{a}']\!]$ is correct shares of $\pi(\vec{a})$.

Theorem A.7. *Protocol A.6 securely computes $\mathcal{F}_{shuffle}$ in the $\mathcal{F}_{rand}$-hybrid model against a single corruption by a passive adversary.*

Proof: All the values that P_2 and P_3 receive in the protocol are masked by $\vec{\alpha}_1$ and $\vec{\alpha}_2$, respectively. All the values that P_1 receives in the protocol are pseudorandom shares of $[\![\vec{a}']\!]$ for P_1. Therefore, the simulator can simulate all values by random elements. □

Note that we can obtain optimized unshuffling protocol similarly with the same complexity as OptShuffle. We call the protocol OptUnshuffle.

A.2.1 Shuffling with reveal. We observe that in the step 2-3 of OptApplyPerm (Protocol A.3), the output of shuffling is immediately revealed. The communication complexity of a sequential invocation of the (optimized) shuffling and reveal protocols is $4m||\mathcal{R}|| + 3m||\mathcal{R}||$ bits in total. We can reduce it to $4m||\mathcal{R}||$ bits by contriving a new protocol that directly computes the shuffling secret shares and then reveals them. The shuffling with reveal protocol appears in Protocol A.8.

Protocol A.8 (Combining shuffling and Reveal)**:**

Notation: $\vec{a}' \leftarrow$ ShuffleReveal($\langle\!\langle \pi \rangle\!\rangle; [\![\vec{a}]\!]$)
Input: A secret-shared vector $[\![\vec{a}]\!]$ and a permutation $\langle\!\langle \pi \rangle\!\rangle$.
Output: The shuffled vector $\vec{a}' = \pi\vec{a}$.

```
1: Let $\langle\!\langle \pi \rangle\!\rangle_i = (\pi_i, \pi_{i+1})$ and $[\![\vec{a}]\!]_i = (\vec{a}_i, \vec{a}_{i+1})$.
2: The parties call $\mathcal{F}_{\text{rand}}$ $m$ times and obtain $(\vec{\alpha}_i, \vec{\alpha}_{i+1})$ for $P_i$.
3: $P_3$ computes $\vec{\gamma} := \pi_1(\vec{a}_3) + \vec{\alpha}_1$ and sends it to $P_2$.
4: $P_1$ computes $\vec{\delta} := \pi_1(\vec{a}_1 + \vec{a}_2) - \vec{\alpha}_1$ and sends it to $P_2$.
5: $P_2$ computes $\vec{a}' = \pi_3 \circ \pi_2(\vec{\gamma} + \vec{\delta})$ and send it to $P_1$ and $P_3$.
6: return $\vec{a}'$
```

We can confirm completeness as

$$\vec{a}' = \pi_3 \circ \pi_2 \circ \pi_1(\vec{a}_1 + \vec{a}_2 + \vec{a}_3) = \pi(\vec{a}).$$

Theorem A.9. *Protocol A.6 securely compute $\mathcal{F}_{ShuffleReveal}$ in the $\mathcal{F}_{rand}$-hybrid model against a single corruption by a passive adversary.*

Proof: The simulator can simulate the view of P_2 by choosing random vector $\tilde{\vec{\gamma}}$ and $\tilde{\vec{\delta}}$ such that $\tilde{\vec{\gamma}} + \tilde{\vec{\delta}} = \pi_2^{-1} \circ \pi_3^{-1}(\vec{a}')$. □

A.3 Communication Complexity

In this section, we evaluate the communication complexity of the optimized sorting protocol with semi-honest security described in Sections A.1 and A.2.

- GenMultiBitSort: This protocol includes $m(2^L - L - 1)$ calls to $\mathcal{F}_{\text{mult}}$ and m calls to $\mathcal{F}_{\ell\text{-mult}}$, both involve sending one ring element. Thus, each party sends $m(2^L - L)||\mathcal{R}||$ bits.
- OptApplyPerm: This protocol includes 1 call to ShuffleReveal in $\mathcal{R}$, L calls to OptShuffle in $\mathbb{Z}_2$ and $\mathcal{F}_{\text{modconv}}$ over m secrets. $\mathcal{F}_{\text{modconv}}$ takes $(1 + ||\mathcal{R}||)m$-bit communication per party [28]. Hence, OptApplyPerm requires sending $\frac{4m}{3}||\mathcal{R}|| + \frac{4mL}{3} + L(1 + ||\mathcal{R}||)m = (\frac{7L}{3} + \frac{4+3L}{3}||\mathcal{R}||)m$ bits per party.
- OptCompose: This protocol includes 1 call to OptUnshuffle, which takes $\frac{4m}{3}||\mathcal{R}||$-bit communication per party.

Therefore, the complexity of OptGenPerm for each party can be bounded by $T(L) = \hat{\ell}m(\frac{7L}{3} + (2^L + \frac{8}{3})||\mathcal{R}||)$ bits where $\hat{\ell} = \lceil \frac{\ell_k}{L} \rceil$. Similar to Eq. (1), the total communication complexity per party can be bounded by

$$T(L) + 3m||\mathcal{R}|| + 2m||\mathcal{R}'||.$$

We can observe that $T(1) > T(2) < T(3) < T(4) < \cdots$. However, we found that $L = 3$ is experimentally the best setting since the difference between $T(2)$ and $T(3)$ is small, and the round complexity decreases as L increases. Hence, we use $L = 3$ for the evaluation. Note that $T(3) \approx m\ell_k(\frac{7}{3} + \frac{32}{9}||\mathcal{R}||)$, and the communication complexity of the optimized protocol is about a third of that of the basic protocol presented in Section 4 (by assuming that $1 \ll ||\mathcal{R}|| \approx ||\mathcal{R}'||$ and $1 \ll \ell_k$).

B OMITTED PROOFS

B.1 Proof of Theorem 4.6

Theorem B.1 (Theorem 4.6, restated)**.** *Protocol (GenPerm, ApplyPerm) securely computes $\mathcal{F}_{sort}$ in the $(\mathcal{F}_{rand}, \mathcal{F}_{mult})$-hybrid model in the presence of semi-honest adversaries controlling a single party.*

Proof: In our protocol, there are two sources of interaction that needs to be simulated by the simulator $\mathcal{S}$: revealing and resharing of secrets. Revealing takes place after shuffling using a random permutation π. In this case, the data seen by the adversary is completely random and uniformly distributed, and therefore the simulator $\mathcal{S}$ can simply send a random element to the adversary as its incoming message when running reveal. Resharing takes place inside the shuffling protocol, when two parties permute the data and then reshare it to the third party. When the corrupted party is the third party, the simulator needs to simulate the two shares sent to him by the honest parties. Recall that in the shuffling protocol, each pair of parties P_i and P_{i+1} choose a random permutation π_{i+1} and a random sharing of 0. Denote the vector of data by $\vec{a}$ and denote the random sharing of 0 by $((r_i, r_{i+1}))_{i=1}^{3}$ such that $r_1 + r_2 + r_3 = 0$. Then, for item k in the vector, P_i sends $a'_{k,i} + r_i$ to P_{i-1}, whereas P_{i+1} sends him $a'_{k,i-1} + r_{i-1}$, where a' is the vector after applying the random permutation π_i. Since both π_i and (r_{i-1}, r_i) are random and independent of the input vector, it follows that the messages sent to P_{i-1} are completely random. Therefore, $\mathcal{S}$ can simulate these messages by choosing random messages for the corrupted party's view. The simulation is therefore perfect. This concludes the proof. □

B.2 Proof of Lemma 5.3

Lemma B.2 (Lemma 5.3, restated)**.** *Each iteration i ($i \in [3]$) in our Shuffling protocol (Protocol 3.2) securely computes $\mathcal{F}^{add}_{reshare\text{-}shuffle}$ in the presence of a single malicious party.*

Proof: Let $\mathcal{S}$ be the ideal-world simulator and let P_i be the corrupted party. $\mathcal{S}$ receives the input shares of the P_i from the functionality. Then, we have two cases:

- *P_i is one the resharing parties.* In this case, $\mathcal{S}$ receives also π and the random $\vec{r}_1, \vec{r}_2$ and $\vec{r}_3$ from the functionality. Thus, it can compute P_i's message and therefore extract the additive error $\vec{\Delta}$, by taking the difference between the actual message and the expected message.
- *P_i is the receiving party.* In this case, $\mathcal{S}$ receives the output shares of P_i from the ideal functionality, and so it can perfectly simulate the honest parties sending their messages to P_i

Observe that in both cases the simulation is perfect and the output is identical to a real execution. This concludes the proof. □

B.3 Proof of Lemma 5.5

Lemma B.3 (Lemma 5.5, restated). *If the corrupted party sends any $\Delta \neq 0$ in any of the calls to $\mathcal{F}^{\text{add}}_{\ell-\text{mult}}$ or $\mathcal{F}^{add}_{reshare\text{-}shuffle}$ before the verification step, then $\beta =$ accept in the verification (Protocol 5.4) with probability at most $\frac{2}{|\mathbb{F}|}$.*

Proof: For each output of $\mathcal{F}^{\text{add}}_{\ell-\text{mult}}$ and $\mathcal{F}^{\text{add}}_{\text{reshare-shuffle}}$, let z_k be the actual value and z'_k be the MAC (i.e., if the parties acted honestly then $z'_k = r \cdot z_k$). Thus, we have

$$w = r \cdot u - v = r \cdot \sum_{k=1}^{n} \alpha_k \cdot z_k - \sum_{k=1}^{n} \alpha_k \cdot z'_k$$

Let Mult and reshare be the set of outputs of multiplications and resharing operations in our protocol, respectively. Then, we can write

$$w_{\text{Mult}} = r \cdot \sum_{k \in \text{Mult}} \alpha_k \cdot z_k - \sum_{k \in \text{Mult}} \alpha_k \cdot z'_k \tag{2}$$

and so

$$w = r \cdot \sum_{k \in \text{reshare}} \alpha_k \cdot z_k - \sum_{k \in \text{reshare}} \alpha_k \cdot z'_k + w_{\text{Mult}}$$

Next, assuming that cheating took place and errors were added to the outputs, we have

$$w = r \cdot \left(\sum_{k \in \text{reshare}} (\alpha_k \cdot z_k) + \epsilon_1 \right) + \epsilon_2 - \sum_{k \in \text{reshare}} \alpha_k \cdot (r \cdot z_k + \delta_k) + \epsilon_3 + w'_{\text{Mult}} \tag{3}$$

where:

- ϵ_1, ϵ_2 and ϵ_3 are the errors added in the verification protocol itself, when calling $\mathcal{F}^{\text{add}}_{\ell-\text{mult}}$ to compute u, $r \cdot u$ and v respectively.
- w'_{Mult} is the value in Eq. (2) after introducing the additive errors from the calls to $\mathcal{F}^{\text{add}}_{\ell-\text{mult}}$ into w_{Mult}.
- δ_k is the accumulated error in the calls to $\mathcal{F}^{\text{add}}_{\text{reshare-shuffle}}$. Specifically, assume that z_k and z'_k were obtained when shuffling the vector $\vec{x}$ and $\vec{x}'$ with the permutation π. Thus, $z_k = x_{\pi^{-1}(k)} + \Delta_k$ and $z'_k = x'_{\pi^{-1}(k)} + \Delta'_k$ where Δ_k and Δ'_k are the errors that were added by the corrupted party. Now, observe that $x'_{\pi^{-1}(k)} = r \cdot x_{\pi^{-1}(k)} + \varepsilon_k$, where ε_k comes from errors in previous operations. Therefore, we have

$$\begin{aligned} \delta_k &= z'_k - r \cdot z_k = r \cdot x_{\pi^{-1}(k)} + \varepsilon_k + \Delta'_k - r \cdot (x_{\pi^{-1}(k)} + \Delta_k) \\ &= \varepsilon_k + \Delta'_k - r \cdot \Delta_k \end{aligned} \tag{4}$$

Next, we compute the probability $\beta =$ accept at the end of the protocol, i.e., the probability that $w = 0$. Observe that

$$w = r \cdot \epsilon_1 + \epsilon_2 - \sum_{k \in \text{reshare}} \alpha_k \cdot \delta_k + \epsilon_3 + w'_{\text{Mult}}. \tag{5}$$

We have the following cases:

- *Case 1: Cheating took place in some call to $\mathcal{F}^{\text{add}}_{\ell-\text{mult}}$.* In this case, if all $\delta_k = 0$, then $w = r \cdot \epsilon_1 + \epsilon_2 + \epsilon_2 + w`_{\text{Mult}}$. By Theorem 5.2 in [13], we have that $w = 0$ with probability of at most $\frac{2}{|\mathbb{F}|}$. Otherwise, there exists $\delta_k \neq 0$, say δ_{k_0}. Then, $w = 0$ if and only if

$$\alpha_{k_0} = (r \cdot \epsilon_1 + \epsilon_2 - \sum_{k \in \text{reshare}, k \neq k_0} \alpha_k \cdot \delta_k + \epsilon_3 + w'_{\text{Mult}}) \cdot (\delta_{k_0})^{-1}$$

which happens with probability $\frac{1}{|\mathbb{F}|}$.
- *Case 2: No cheating took place in any call to $\mathcal{F}^{\text{add}}_{\ell-\text{mult}}$.* In this case, $w'_{\text{Mult}} = 0$ (note that all calls to $\mathcal{F}^{\text{add}}_{\ell-\text{mult}}$ are done in GenBitPerm before any shuffling and so there are no previous errors to take into account here). Thus, we have

$$w = r \cdot \epsilon_1 + \epsilon_2 - \sum_{k \in \text{Shuffle}} \alpha_k \cdot \delta_k + \epsilon_3 \tag{6}$$

Now, if there exists $\delta_k \neq 0$, then as in the previous case, $w = 0$ with probability $\frac{1}{|\mathbb{F}|}$. However, it might happen that errors were added when calling $\mathcal{F}^{\text{add}}_{\text{reshare-shuffle}}$ and still $\delta_k = 0$ for all ks. Let k_0 be the *first* call to $\mathcal{F}^{\text{add}}_{\text{reshare-shuffle}}$ for which an attack took place. This implies that in Eq. (4) $\varepsilon_{k_0} = 0$, and so $\delta_{k_0} = \Delta'_k - r \cdot \Delta_k$. However, since r is uniformly chosen from $\mathbb{F}$, it follows that $\delta_{k_0} = 0$ with probability $\frac{1}{|\mathbb{F}|}$. It follows that the probability that $w = 0$ in Eq. (6) is $\frac{1}{|\mathbb{F}|} + \left(1 - \frac{1}{|\mathbb{F}|}\right) \cdot \frac{1}{|\mathbb{F}|} \leq \frac{2}{|\mathbb{F}|}$.

□

B.4 Proof of Theorem 5.6

Theorem B.4 (Theorem 5.6, restated). *Let s be a statistical security parameter, and let $\mathbb{F}$ be a finite field such that $\frac{2}{|\mathbb{F}|} \leq 2^{-s}$. Then, our protocol (as described in the text) securely computes $\mathcal{F}_{sort}$ with abort in the ($\mathcal{F}_{rand}$, $\mathcal{F}^{\text{add}}_{\ell-\text{mult}}$, $\mathcal{F}^{add}_{reshare\text{-}shuffle}$)-hybrid model, in the presence of one malicious party.*

Proof: Let $\mathcal{S}$ be the ideal world simulator and let P_i be the corrupted party. In the simulation, $\mathcal{S}$ plays the role of $\mathcal{F}_{\text{rand}}$, $\mathcal{F}^{\text{add}}_{\ell-\text{mult}}$, $\mathcal{F}^{\text{add}}_{\text{reshare-shuffle}}$ and the honest parties interacting with P_i. Playing the role of these three functionalities, the simulator receives from P_i its shares of the output of each generation of a random secret, multiplication and shuffling, as well as the additive attacks. Note that $\mathcal{S}$ knows the corrupted party P_i's share throughout the execution. It remains to show how to simulate the verification protocol and the revealing that take place after each verification.

Simulating the verification protocol. The simulator $\mathcal{S}$ receives the P_i's shares for the output of the invocations of $\mathcal{F}^{\text{add}}_{\ell-\text{mult}}$ when computing u, v and $r \cdot u$ and the additive errors ϵ_1, ϵ_2 and ϵ_3. To simulate the opening of w, there are several cases:

- *Case 1: No Additive attacks were sent.* In this case, everyone acted honestly and so $w = 0$.
- *Case 2: For each multiplication or resharing operation it holds that* $\Delta'_k - r \cdot \Delta_k = 0$. Recall that Δ'_k is the error when computing the MAC and Δ_k is the error when computing the actal output of each operation (see Eq. (4)). In this case, all the errors cancel each other, and therefore there are no accumulated errors that propagates to the next layer of operations in the protocol. Therefore, both Δ_k and w'_{Mult} in Eq. (5) are 0, and so the simulator $\mathcal{S}$ sets: $w = r \cdot \epsilon_1 + \epsilon_2 + \epsilon_3$.

- *Case 3: there exists k such that $\Delta'_k - r \cdot \Delta_k = 0$.* In this case, we have that Δ_k or w'_{mult} in Eq. (5) are not 0. Now, since the random coefficients are secret and distributed uniformly, this implies that w is distributed randomly over $\mathbb{F}$. Thus, $\mathcal{S}$ sets w to be a random element in $\mathbb{F}$.

Now, in the above last two cases, if despite an additive attack was carried-out before the verification protocol $w = 0$, the simulator outputs fail and halts. Otherwise, given the corrupted party's shares w_i and w_{i+1}, the simulator sets $w_{i-1} = w - w_i - w_{i+1}$ and simulates the honest parties sending w_{i-1} to P_i in the procedure reveal($[\![w]\!]$). If the corrupted party P_i sent inconsistent shares or if $w \neq 0$, the simulator $\mathcal{S}$ sends `abort` to the trusted party computing $\mathcal{F}_{\text{sort}}$ and simulates the honest parties aborting the protocol.

Simulating the revealing procedure. The simulation proceed to this procedure, only after a verification protocol has ended successfully, with the honest parties outputting accept. In this case, $\mathcal{S}$ can work as in the semi-honest setting, i.e., for each opened value a, the simulator $\mathcal{S}$, who knows the corrupted party's shares a_i and a_{i+1}, simply chooses a random a and sets $a_{i-1} = a - a_i - a_{i+1}$. As before, if P_i sends inconsistent shares in the reveal procedure, then $\mathcal{S}$ sends `abort` to the trusted party computing $\mathcal{F}_{\text{sort}}$ and simulates the honest parties aborting the protocol.

The simulation above is distributed identically to a real execution, except for the case that $\mathcal{S}$ outputs fail. However, note that this event happens when the corrupted party added errors and yet $w = 0$, which corresponds to event where errors are not detected by the honest parties in the real execution. Thus, by Lemma 5.5, it follows that $\Pr[\mathcal{S} \text{ outputs fail}] \leq \frac{2}{|\mathbb{F}|}$, which is exactly the statistical error allowed by the theorem. This concludes the proof. □

C COMPUTING VARIANTS OF THE HEAVY-HITTERS PROBLEM

Protocol C.1 describes another method for securely computing heavy hitters using a sorting protocol. The protocol uses two sorting operations, one using the values as a key, and one using a single bit as the key. Crucially, the second sorting operation must be implemented as a stable sort. In addition to sorting, the protocol computes simple operations that can implemented as a logarithmic depth circuit of binary gates.

Protocol C.1 (Computing heavy hitters)**:**

```
Input: A secret-shared vector $[\![\vec{v}]\!]$ of the values of the clients. A threshold $t$.
Output: All values which appear more than $t$ times.
1: Sort the elements of $[\![\vec{v}]\!]$. (As a result, all identical values are adjacent to each other.)
2: Compute a shared vector of bits $[\![\vec{f}]\!]$, where $f_i = 1$ iff $v_i \neq v_{i+1}$.
   ($f_i$ is 1 iff $v_i$ is the last in a sequence of identical values.)
3: Generate a secret-shared vector $[\![\vec{t}]\!]$, where $t_i = i$.
4: For each index $i$ consider the concatenation $f_i||t_i||v_i$ as a single element. Sort these elements based on the single-bit key $f_i$. (As a result, the element with $f_i = 1$ appear first. Since the sorting is stable, they are ordered by their location in the previous order, $t_i$.)
5: For each $i$, if $f_i = f_{i-1} = 1$, and $t_i - t_{i-1} \geq t$, output $v_i$ and $t_i - t_{i-1}$.
   (Identify elements which appear at least $t$ times and output them together with their count.)
```

As described, Protocol C.1 outputs the identities of values appearing at least t times, as well as the counts of appearances of each item (Protocol 6.1 does not compute these counts). Of course, Protocol C.1 can be slightly changed to not output the counts, or to output noisy counts in order to ensure differential privacy. Protocol C.1 can also be easily changed to support the computation of other variants of the heavy hitters problem. The overhead of these variants is similar to that of the original protocol:

- If it is only required to output a histogram of all values, the protocol can end after the first sort operation (Step 1).
- If it is required to output the top s heavy hitters, instead of outputting all values which appear more than t times, then instead of checking in Step 5 whether $t_i - t_{i-1} \geq t$, the protocol should sort the items according to $t_i - t_{i-1}$ and output the top s results.
- Of course, the protocol can easily be adapted to output the top s values which appear more than t times.