Secure Multi-Party Sorting and Applications

Kristján Valur Jónsson[1], Gunnar Kreitz[2], and Misbah Uddin[2]

[1] Reykjavik University
[2] KTH—Royal Institute of Technology

Abstract. Sorting is among the most fundamental and well-studied problems within computer science and a core step of many algorithms. In this article, we consider the problem of constructing a secure multi-party computing (MPC) protocol for sorting, building on previous results in the field of sorting networks.
Apart from the immediate uses for sorting, our protocol can be used as a building-block in more complex algorithms. We present a weighted set intersection algorithm, where each party inputs a set of weighted elements and the output consists of the input elements with their weights summed. As a practical example, we apply our protocols in a network security setting for aggregation of security incident reports from multiple reporters, specifically to detect stealthy port scans in a distributed but privacy preserving manner. Both sorting and weighted set intersection use $\mathcal{O}(n \log^2 n)$ comparisons in $\mathcal{O}(\log^2 n)$ rounds with practical constants.
Our protocols can be built upon any secret sharing scheme supporting multiplication and addition. We have implemented and evaluated the performance of sorting on the Sharemind secure multi-party computation platform, demonstrating the real-world performance of our proposed protocols.

Keywords. Secure multi-party computation; Sorting; Aggregation; Cooperative anomaly detection

1 Introduction

Intrusion Detection Systems (IDS) [16] are commonly used to detect anomalous and possibly malicious network traffic. Incidence reports and alerts from such systems are generally kept private, although much could be gained by cooperative sharing [30]. The primary obstacle against cooperative processing of incidence logs is privacy: organizations are typically reluctant to disclose data which potentially reveals information about themselves or their customers. Forums for sharing IDS logs already exist, as exemplified by the `www.dshield.org` service operated by the Internet Storm Center, where firewall logs can be submitted and centrally processed. However, services of this kind can leak sensitive internal information about the participating entities.

In this paper, we discuss an approach for provably secure processing of sensitive IDS logs. Specifically, we construct *secure multi-party computation* (MPC) sorting and aggregation protocols, and demonstrate their applicability to distributed anomaly detection.

1.1 Our Contribution

We make a number of contributions in this paper. First of all, we propose a privacy preserving secure sorting protocol, based on *sorting networks* [2] and generic MPC primitives, which guarantees that no single entity learns anything about other parties' inputs (contributions), apart from the length of the inputs. Specifically, we implement a sorting network using a secure comparison protocol [12,27] to construct comparison gates. We discuss some details on how to apply a data-oblivious sorting algorithm (where the sequence of comparisons does not depend on the input), giving an extension to hide the length of the inputs, albeit at considerable cost.

The observation that sorting networks could be used to implement secure sorting has previously been made [31,20]. They furthermore present a randomized version of the Shellsort algorithm, which sorts with running time $\mathcal{O}(n \log n)$ with very high probability. This is asymptotically faster than the algorithms we present. However, the constants are slightly worse, such that it would not be more efficient on the sizes we test when evaluating. We were not aware of this work when originally writing this paper.

It would be interesting to evaluate randomized Shellsort on an MPC cluster to demonstrate feasibility of MPC computations on even larger data sets than we do in this paper. Our evaluation results indicate better performance (but with weaker security guarantees) than the 2-party implementations presented in [31].

Sorting is an important building block. One recent example of this is Damgård, Meldgaard and Nielsen [15] who used sorting networks in the MPC setting to implement Oblivious RAM.

We utilize the secure sorting protocol to construct a solution to the *top-k* problem [9], that is, securely revealing the global k highest contributors to a weighted set intersection of $(key, weight)$ pairs. We consider the *symmetric case*, where all participants receive the same output. Our algorithm is an approach to solving the generalized set intersection problem, as discussed in the MPC context by Many [26]. We then discuss how the secure *top-k* algorithm can be used to securely share the k most likely suspects in a distributed intrusion detection scenario.

The feasibility of the secure sorting protocol in a real-life scenario is demonstrated by means of a prototype implementation on the Sharemind [6] MPC platform, which supports three *privacy peers*. The system is secure against a *passive* (honest-but-curious) adversary corrupting one of the three parties. We emphasize that the protocols are not specific to the Sharemind platform. Rather, our protocols are described as a sequence of computations on secret shared data, taking input and producing output in secret shared form. This allows our protocols to be easily re-used as parts of other protocols and to be implemented on other MPC platforms.

Our implementation sorts 2^{14} elements in under 4 minutes on a three machine cluster run by the Sharemind team. The performance measurements are preliminary and only intended to give a rough indication of feasibility, rather

than a basis for comparison with other solutions, as the Sharemind platform is currently being actively developed and our protocols can certainly be optimized. However, we conclude that this small experiment demonstrates the feasibility of applying MPC-based methods to the problem of secure processing of incidence reports. We reserve optimization and scaling issues for future work.

1.2 Related Work

Protocols to securely evaluate any function are given by Ben-Or, Goldwasser, and Wigderson [5], and Chaum, Crépeau, and Damgård [11]. Both these results present protocols for computing addition and multiplication (XOR and AND) on values in (verifiable) secret shared form, and with results remaining secret shared. As these primitives are complete, any function can then be evaluated gate by gate.

Adding to these primitives, Damgård *et al.* [12] presented a generic comparison protocol which can be used with most common secret sharing mechanisms. Later, a more efficient protocol for comparison was proposed by Nishide and Ohta [27]. Bogdanov *et al.* [7] gave efficient protocols specialized for the case when exactly 3 parties participate in the computation.

A line of research has focused on developing efficient MPC protocols for specific functions. Among these are more specific applications, such as auctions [8], and comparing gene sequences [21], but also more generic primitives such as set operations [18,23], top-k queries [9], and weighted set intersection [26]. Finding efficient algorithms for private sorting has been stated as an open problem by Du and Atallah [17], and more recently by Burkhart and Dimitropoulos [9]. Goodrich [20] has proposed the randomized Shellsort algorithm which runs in time$\mathcal{O}(n \log n)$ with high probability addressing the sorting problem.

A number of frameworks and specialized programming languages to implement and run secure multi-party computation protocols have been created. These include FairplayMP [4], Sharemind [6], SEPIA [10], and VIFF [13]. FairplayMP builds on the idea of "garbled circuits" [34,3], Sharemind uses additive sharing over a ring, and the latter two systems build on Shamir's secret sharing [28]. Despite these three different approaches to implementing a generic MPC framework, they all support a similar set of primitives, including addition, multiplication, comparisons and equality testing. However, the performance properties of FairplayMP are different from the others, and we will focus on the latter three. Programming on these platforms either uses a specialized language, or a standard programming language and library calls, depending on the platform.

One of our proposed applications is a protocol for cooperative, but private, processing of security-related information, shared between mutually distrustful organizations. Secure sharing of server logs and incidents reports is discussed by Lincoln *et al.* [25], Slagell *et al.* [30] and Xu *et al.* [32]. The common thread is that sharing of incident logs can aid in early detection of anomalies, but that naive merging of logs is not a feasible solution, as this may reveal sensitive internal and customer information. The authors cited propose a variety of log sanitization

techniques to protect potentially sensitive information. In contrast, we propose to use MPC protocols for privacy preserving cooperative log processing.

Katti *et al.* [22] perform empirical analysis on a large dataset obtained from 1700 IDS systems. They observe that coordinated attacks on multiple targets are a large fraction of the total attacks observed. Further, such attacks are seen to target small clusters of victims with relatively constant size and membership. The clustering phenomenon is most likely due to the combination of software run and services offered on the platforms under attack. The observations of Katti *et al.* support the feasibility of cooperative anomaly detection using MPC protocols, such as is the focus of our work.

Building on their SEPIA platform, Burkhart and Dimitropoulos [9] have proposed an algorithm for aggregation and top-k queries, using hash tables and secret sharing. Their motivation is secure aggregation of network flow records to cooperatively reveal anomalies. In this paper, we present an alternate algorithm for top-k queries based on sorting. We remark that their solution computes an approximately correct value, while our proposed solution computes exact results.

Also building on the SEPIA platform, Many [26] in his Master's thesis studied a problem called weighted set intersection. In this problem, each participant gives as input a list of (value, weight)-pairs, and only keys reported by some number of peers and which have weights with sum greater than some threshold are output. For this problem, we propose an alternate solution based on sorting.

We apply techniques from sorting networks in our primary building block, the MPC sorting protocol. A sorting network is a circuit which uses a fixed number of comparison gates to sort its inputs. A comparison gate is a gate with two inputs a, b and two outputs, which output $\min(a, b)$ and $\max(a, b)$, respectively. Another perspective is that a sorting network is a sorting algorithm with the property that which elements it compares is independent of the input.

Two performance metrics are of importance in the sorting network literature, as well as in our application: the number of comparison gates, and the depth of the circuit. Batcher [2] presented two famous sorting networks, odd-even merge sort and bitonic sort. Both algorithms have depth $\mathcal{O}\big(\log^2 n\big)$ using $\mathcal{O}\big(n \log^2 n\big)$ gates and are efficient in practice. The Shell sort [29] algorithm can also be implemented as a sorting network with the same asymptotic performance. Ajtai, Komlós, and Szemerédi [1] constructed the AKS sorting network which achieves the theoretically optimal $\mathcal{O}(\log n)$ depth with $\mathcal{O}(n \log n)$ gates, but the constant hidden in the ordo notation makes the algorithm inefficient for practical input sizes. Leighton and Plaxton [24] have proposed a sorting network with optimal asymptotic performance and practical constants, but which do not sort correctly for a small fraction of inputs.

2 Preliminaries and Notation

We denote a sequence of elements by a bold letter, like $\boldsymbol{a}$, and use the same letter with an index to denote the elements in the sequence, so a_i is the i^{th} element of the sequence $\boldsymbol{a}$. When we treat sequences of pairs, we explicitly associate

two new letters with the first and second item of the pair. Thus, for a vector $\boldsymbol{b}$ consisting of pairs (x_i, y_i), we write the element $b_j = (x_j, y_j)$.

2.1 Secure Multi-Party Computation

The aim of secure multi-party computation techniques is to enable joint computation of some function f by n mutually distrustful entities, in such a manner that no collusion (below some threshold) of parties learns anything beyond what they can deduce from their own inputs and observing the output. MPC techniques were first introduced by Yao [33] in 1982 and have been extensively studied since.

The most common paradigm for MPC is that of computing on shared secrets [5,11]. In this paradigm, parties use secret sharing to share their private inputs among all participants in the computation. They then execute protocols which operate on secret shared inputs and return the output in secret shared form. Finally, the shares of the output are combined to recover the output of the function.

Our protocols are based on a small number of primitives, which we assume to be securely implemented. These operations are addition, multiplication, comparison and equality testing on secret shared inputs. The values operated on are in a finite ring or field, which is known to all parties. We denote the ring or field by $\mathbb{Z}_k$. In Table 1, we list the primitives and the notation we use. All arithmetic operations are in $\mathbb{Z}_k$.

Notation	Operation
$[x]$	The value x shared between the parties
$[\boldsymbol{a}]$	The sequence $\boldsymbol{a}$ shared element-wise between the parties
$[x] + [y]$	Computation of the value $x + y$ shared by the parties
$c \cdot [x]$	Computation of x multiplied by a public constant
$[x] \cdot [y]$	Computation of the product of two secret shared values
$[x < y]$	The value 1 if x is strictly smaller than y, 0 otherwise
$[x = y]$	The value 1 if x is equal to y, 0 otherwise

Table 1. List of MPC primitives used

Several of our algorithms operate on sequences. When sharing a sequence, it is shared element-wise. Thus, the notation $[\boldsymbol{a}]$ for $\boldsymbol{a}$ of length m means that the parties have shares $[a_1], [a_2], \ldots, [a_m]$. We remark that sharing a sequence reveals the length of the sequence to all parties. Similarly, when sharing pairs we share them element-wise. Thus, for a sequence $\boldsymbol{b}$ of length m consisting of pairs (x_i, y_i), $[\boldsymbol{b}]$ means that parties have shares $[x_1], [y_1], [x_2], [y_2], \ldots, [x_m], [y_m]$.

The computational and communication cost of the different primitives varies significantly. Addition of two shares, as well as multiplication by a constant, can be performed locally by all participants in the protocol. Computing the less-than function or equality is done by computing a large number of multiplications,

the exact number depending on the protocol used and on the bit length of the elements. For instance, the protocol by Nishide and Ohta [27] uses 15 rounds and $279\ell + 5$ multiplications to compare two elements where ℓ is the number of bits needed to represent an element in $\mathbb{Z}_k$, i.e. $\lceil \log_2 k \rceil$. All operations in Table 1 use a constant number of rounds. For fixed k, the communication is also $\mathcal{O}(1)$.

As in other areas of computer science, MPC protocols benefit from parallelization. Therefore, when designing and implementing a protocol, it is important to keep not only the number of operations, but also the number of rounds as low as possible. In Section 6, we demonstrate the importance of parallelization by comparing the run time of a parallelized and non-parallelized version of bubble sort.

In the context of MPC, different notions of what a party is has been discussed. Traditionally, the model has been that all parties contributing input also participate in the secure computation. However, as performance of MPC protocols does not scale well to a large number n of participants, a different model has been proposed [14]. In this model, a large number of *peers*[3] collect data and send it in secret shared form to a relatively small number of *privacy peers*. The privacy peers then run the protocol to perform the secure computation on the inputs. Most often when designing protocols, the distinction between the two models is not important. However, as we discuss in Section 3, for one of our sorting protocols, it does make a difference.

The exact security guarantees achieved, as well as the constraints on the execution environment, depend on the MPC framework on which our protocol is implemented. The frameworks based on Shamir's secret sharing are secure against a *passive* adversary who can corrupt up to $\lfloor (n-1)/2 \rfloor$ nodes. These protocols require each pair of computing nodes to be connected by *private channels*. The security guarantee still holds even if the adversary has unlimited computational power (*information-theoretic security*). The Sharemind framework is implemented for $n = 3$ parties and provides security against a passive (honest-but-curious) adversary corrupting any single node. By implementing the protocols using verifiable secret sharing, security against up to $\lfloor (n-1)/3 \rfloor$ *active* adversaries can be achieved. Of the sharing-based frameworks, only VIFF has implemented verifiable secret sharing.

Informally, the security guarantees in the passive case say that no collusion of parties (for collusions smaller than the security threshold) learn more than what they do from their inputs and the output of the function. We refer to [19] for the formal definitions.

2.2 Sorting Networks

A sorting network is a circuit with m inputs and m outputs[4]. The outputs take the same values as the inputs, but in sorted order. The gates used to construct the

[3] Here, we follow the terminology of [10]

[4] More formally, it is a family of circuits for all sizes m, but we will ignore this distinction.

circuit are so called *comparison gates*, which compare and sort a pair of elements. We may view a sorting network as a sorting algorithm with the property that the comparisons made are independent from the values to be sorted. Among the well-known traditional sorting algorithms, bubble sort and Shell sort can both be implemented in such a way that they have this property.

The viewpoint of a sorting network as a sorting algorithm with a fixed sets of comparisons illuminates why such constructions can function as MPC sorting algorithms. This means that given a design for a comparison gate, which takes two inputs in secret shared form and sorts them (keeping them in secret shared form), a sorting network can be implemented as an MPC computation. Furthermore, the performance metrics of a sorting network are the same as those we are interested in optimizing in MPC; the circuit depth corresponds to the number of rounds and the number of comparison gates used to the number of comparisons.

A comparison gate is a gate with two inputs a, b and two outputs, h, l such that the output h takes value $\max(a, b)$ and the output l takes value $\min(a, b)$. A comparison gate can be implemented as a multi-party computation by using the following construction:

$$[h] = [a < b] \cdot [b] + (1 - [a < b]) \cdot [a]$$
$$[l] = [a < b] \cdot [a] + (1 - [a < b]) \cdot [b]$$

We define the function Compare-exchange(a_i, a_j) and assume sequences are sorted in-place. This function compares the two elements a_i, a_j and swaps them if needed such that after its execution, $a_i \leq a_j$. The Compare-exchange operation is implemented using the construction given above. We present the algorithm later in Algorithm 3.

Sorting networks are comparison based, so the general $\Omega(m \log m)$ lower bound on the number of comparisons applies. The AKS sorting network [1] matches this bound, but with such large constants that it is inefficient for practical input sizes. There are several sorting networks [2,29] with slightly worse asymptotic performance, $\mathcal{O}\big(m \log^2 m\big)$ comparisons and depth $\mathcal{O}\big(\log^2 m\big)$, but with good constants. Of these, we will focus on the odd-even merge sort algorithm by Batcher [2]. There is a also fast sorting network by Leighton and Plaxton [24] with $\mathcal{O}(m \log m)$ comparisons, but which does not correctly sort the output for a very small fraction of the possible inputs. In the context of MPC, we feel that correctness outweighs performance. As we show in Section 6, an MPC implementation of odd-even merge sort achieves good performance even on reasonably large data sets.

As the name implies, odd-even merge sort is based on the well-known merge sort algorithm. The overall structure is the same, splitting the input in two halves, recursively sorting each half, and then merging. The difference compared to the standard merge sort algorithm lies in the merge step, which we describe in Algorithm 1. For completeness, we give a full description of the whole algorithm in Algorithm 2. With some abuse of notation, we write Function$(a_1, a_2, \ldots, a_m)$ to call Function with the sequence $\boldsymbol{a} = a_1, a_2, \ldots, a_m$ when Function takes

a single sequence as argument. We describe the algorithms as operating on a sequence, the length of which is a power of 2, but it is easy to modify to arbitrary input lengths by omitting some comparisons. We conclude this section by restating the correctness and performance analysis from [2] of the odd-even merge sort algorithm as a theorem.

Algorithm 1 ODD-EVEN MERGE [2]

Input: Sequence $\boldsymbol{a}$ whose two halves $a_1, a_2, \ldots a_{m/2}$ and $a_{m/2+1}, a_{m/2+2}, \ldots, a_m$ are sorted. Length m is a power of 2.
Output: Sequence $\boldsymbol{a}$ is modified in-place to be sorted.
if $m > 2$ **then**
  ODD-EVEN MERGE$(a_1, a_3, a_5, \ldots, a_{m-1})$
  ODD-EVEN MERGE$(a_2, a_4, a_6, \ldots, a_m)$
  for $i \in \{2, 4, \ldots, m-2\}$ **do**
    COMPARE-EXCHANGE$(a_i,\ a_{i+1})$
  end for
else
  COMPARE-EXCHANGE$(a_1,\ a_2)$
end if

Algorithm 2 ODD-EVEN MERGE SORT [2]

Input: Sequence $\boldsymbol{a}$ of length m (a power of 2).
Output: Sequence $\boldsymbol{a}$ is modified in-place to be sorted.
if $m > 1$ **then**
  ODD-EVEN MERGE SORT$(a_1, a_2, \ldots, a_{m/2})$
  ODD-EVEN MERGE SORT$(a_{m/2+1}, a_{m/2+2}, \ldots, a_m)$
  ODD-EVEN MERGE$(\boldsymbol{a})$
end if

Theorem 1 ([2]). *The* ODD-EVEN MERGE *algorithm uses* $\mathcal{O}(m \log m)$ *comparisons with depth* $\mathcal{O}(\log m)$. ODD-EVEN MERGE SORT *correctly sorts using* $\mathcal{O}\big(m \log^2 m\big)$ *comparisons with depth* $\mathcal{O}\big(\log^2 m\big)$.

3 Secure Multi-Party Sorting

We construct a secure sorting network on secret shares by using a MPC protocol for COMPARE-EXCHANGE. This allows us to leverage the long line of research in sorting networks to construct efficient MPC sorting protocols. However, there are a number of details which remain in constructing an MPC sorting algorithm related to the input to the protocol which we proceed to discuss.

3.1 Problem Definition

When we discuss MPC sorting, there are actually two slightly different problems which we tackle. The first is sorting as a stand-alone functionality, or as the first step of an algorithm. In this usage, we may need to hide the number of items which are contributed by each party. The second is using sorting as a step within an algorithm, where the number of elements to be sorted is known. We give a definition to capture this distinction.

Definition 1 (Multi-party sorting). *n parties execute a multi-party sorting protocol. Each party P_i gives as input a sequence $\boldsymbol{a}^i$ of length m_i. After the protocol has executed, the parties learn the sorted sequence $\boldsymbol{a}$, whose elements are the concatenations of the sequences $\boldsymbol{a}^i$. We say that a sorting protocol is* composable *if the parties learn the sorted output $\boldsymbol{a}$ in secret shared form and the total length $|\boldsymbol{a}| = \sum_{i=1}^{n} m_i$ openly. If the parties learn the input lengths contributed by other parties, we say that the protocol has* public input lengths*, otherwise we say it has* private input lengths*.*

Our protocols are given as standard MPC operations on secret shares, and are thus composable. The intuition for the notion of public input lengths comes from the fact that it is difficult to hide the size of inputs in the context of MPC protocols. For sorting, we give a general, but somewhat costly, transformation which transforms a protocol with public input lengths into a protocol with private input lengths. We stress that even with public input lengths, which party contributed what element of the sorted output remains secret.

3.2 Public and Private Input Lengths

Our transformation from public to private input lengths begins by the parties computing the length of the output. This is easily done by each party secret sharing the length of their input. The lengths are summed, and the result is opened and revealed to all parties. Let the sum be m and 0 be the smallest possible input. Each party then locally appends 0's to their input until their input is of length m. They then participate in sorting using this new, longer input. The cost is that the sorting algorithm must be run on a sequence of length mn (where n is the number of parties) instead of length m. One could consider a middle ground between public and private input lengths where the parties compute the maximum input length of any single party, and pad their input to that length instead of the output length. This would increase the performance, at the cost of revealing the maximum input length.

Here, we would like to tie back into the discussion on the roles of different parties in the MPC context. This transformation is slightly at odds with the privacy-peer view of MPC, as it requires one round of interaction between the data reporting peers and the privacy-peers executing the MPC protocol, as the peers must first contribute their number of shares, wait to receive the sum of these numbers, and then contribute their actual input (appropriately padded). However, we believe this limited interaction is acceptable in most applications

of MPC in a privacy-peer setting as the interaction is quick, very limited and not computationally demanding for the data reporting peers.

A similar issue arises with regard to the domain in which the elements lie. For our MPC protocol, we require the elements to be in $\mathbb{Z}_k$ for some public k. But how large is k? In most applications, an upper bound on k is known *a priori*, for instance when operating on IP addresses which are of fixed length. If no such bound is known from the applications, the parties can precede the sorting protocol with a protocol computing the maximum element in the input to some suitable degree of precision (e.g., the number of bits required to store it) and then initialize the MPC framework with a ring or field of appropriate size. In the remainder of this paper, we assume that the parties know a suitable size *a priori* to present the main ideas more clearly.

3.3 Multi-party Sorting

We are now ready to present our protocol for MPC sorting. We begin by describing a comparison gate, Compare-exchange. We present the algorithm as operating on key-value pairs in Algorithm 3. With some abuse of notation, we call this function both when sorting key-value pairs and when sorting singleton elements. In the algorithm, we use $[s]$ and $[t]$ as temporary variables for the values that will go into outputs $[x_1]$ and $[y_1]$.

Algorithm 3 Compare-exchange

Input: Two key-value pairs $([x_1], [y_1])$, $([x_2], [y_2])$ in secret shared form
Output: The pairs are swapped in-place such that the pair with the smallest key is in the first position.
Compute $[c] \leftarrow [x_1 < x_2]$
Let $[s] \leftarrow [c] \cdot [x_1] + (1 - [c]) \cdot [x_2]$
Let $[t] \leftarrow [c] \cdot [y_1] + (1 - [c]) \cdot [y_2]$
Let $[x_2] \leftarrow (1 - [c]) \cdot [x_1] + [c] \cdot [x_2]$
Let $[y_2] \leftarrow (1 - [c]) \cdot [y_1] + [c] \cdot [y_2]$
Let $[x_1] \leftarrow [s]$
Let $[y_1] \leftarrow [t]$

We include a full description of the protocol as Algorithm 4. The construction used for private input lengths is general and can be used for any MPC sorting protocol.

We remark that the users could sort their own contributed inputs locally. Some sorting networks, in particular the odd-even merge sort network that we presented as Algorithm 2, could be modified to make use of this to increase the performance. However, we omit the details of such modifications.

Theorem 2. *Algorithm 4 is correct and secure.*

Algorithm 4 MULTI-PARTY SORTING

Input: Each party P_i inputs sequence $\boldsymbol{a}^i$ of length m_i
Output: Sorted sequence $\boldsymbol{a}$ in secret shared form
if Private input lengths **then**
  Each party P_i shares $[m_i]$
  Compute $[m] = \sum_{i=1}^{n} [m_i]$
  Open $[m]$ to all parties
  Each party P_i forms $\boldsymbol{b}^i$ by padding $\boldsymbol{a}^i$ with $m - m_i$ 0 elements
else
  Each party P_i sets $\boldsymbol{b}^i \leftarrow \boldsymbol{a}^i$
end if
Each party P_i shares $\boldsymbol{b}^i$ element-wise
Let $[\boldsymbol{b}]$ be the concatenation of all $[\boldsymbol{b}^i]$
Sort $[\boldsymbol{b}]$ in-place using a sorting network
if Private input lengths **then**
  return the last m elements of $\boldsymbol{b}$
else
  return $\boldsymbol{b}$
end if

Proof (Sketch). Correctness of the algorithm follows directly from the correctness of the sorting network implemented.

The security of the protocol relies on the security of the implementation of the underlying MPC primitives for all MPC operations. In the case with private input lengths, each party performs the same sequence of MPC operations independent of their inputs. Without private input sizes, the only difference between parties in the protocol is how many elements are shared by each party. The only value opened to the parties is the size of the output, which is part of the output in Definition 1. □

4 Weighted Set Intersection and Aggregation

We now give our secure weighted set intersection protocol, which leverages our sorting protocol from Section 3.

4.1 Problem Definition

We formally define our problem and refer to it as weighted set intersection. We remark that this name was also used by Many [26] for their problem, which is closely related but not exactly the same. In light of the application, we will refer to weighted set intersection and the top-k problem as *aggregation*.

Definition 2 (Weighted Set Intersection, Top-k). *In the Weighted Set Intersection problem, each party P_i gives as input a sequence $\boldsymbol{a}^i$ of length m_i consisting of value-weight pairs, $a^i_j = (v^i_j, w^i_j)$, with positive weights. The output is*

a sequence of value-weight pairs such that element v_j is in the output if it was in the input of any party. Elements in the output are assigned weight as the sum over the input weights of all parties. If the output is truncated to only include the top k values, sorted by their aggregate weights, we say that it solves the top-k problem.

In Section 3, we defined and discussed the notion of public or private input lengths. A similar concern exists in our current setting. However, this time, the padding idea we used in sorting does not work. The key difference between sorting and aggregation is that with aggregation, it is difficult to compute the length of the output without performing the actual aggregation. By applying the same transformation as we did for sorting, we would get a protocol hiding the number of items submitted by each party, but where the difference between the length of the inputs and the output reveals how many key-value pairs were reported by more than one party. Thus, we propose a protocol for aggregation with public input lengths.

4.2 The Core Aggregation Step

In our algorithm, we sort the complete list of item-weight pairs, ordered by item. Then an aggregation step is executed, which aggregates the weights of all items. After the aggregation step, we want a single entry in the list for each distinct item, its weight being the sum of its weights in the original input. This means that some item-weight pairs need to be removed; "removal" of an item-weight pair is performed by setting its weight to 0 which ensures its exclusion from the output later in the algorithm. We begin our description by presenting in Algorithm 5 an MPC algorithm, Aggregate-if-equal, for comparing two key-value pairs and merging them if they have the same key, analogous in function to Compare-exchange.

Algorithm 5 Aggregate-if-equal

Input: Two key-value pairs $([x_1], [y_1])$, $([x_2], [y_2])$ in secret shared form
Output: If the keys are equal, the values are merged in-place
Compute $[c] \leftarrow [x_1 = x_2]$
Let $[y_1] \leftarrow [y_1] + [c] \cdot [y_2]$
Let $[y_2] \leftarrow (1 - [c]) \cdot [y_2]$

Returning to the original question: given a sorted list, how to aggregate the weights for an item? A naive solution would be to use $\mathcal{O}(m^2)$ calls to Aggregate-if-equal, comparing each pair of items and summing weights appropriately. However, we can do better. We present an algorithm which aggregates m items using $\mathcal{O}(m \log m)$ equality tests with a construction similar to Batcher's Odd-even merge algorithm. We call this algorithm Odd-even aggregation and give the pseudo-code in Algorithm 6 and proof of correctness

in Theorem 3. As with sorting, we describe the algorithm for an input lengths which is a power of 2, and the algorithm can easily be adapted to arbitrary input lengths.

Algorithm 6 ODD-EVEN AGGREGATION

Input: Sequence $\boldsymbol{a}$ of key-value pairs (x_i, y_i) which are sorted by the key values. The length m of $\boldsymbol{a}$ is a power of 2.
Output: Sequence $\boldsymbol{a}$ is modified in-place to contain one key-value pair with non-zero weight for each unique key. "Removed" entries have weight set to 0.

```
if m > 2 then
  ODD-EVEN AGGREGATION(a_1, a_3, a_5, ..., a_{m-1})
  ODD-EVEN AGGREGATION(a_2, a_4, a_6, ..., a_m)
  for i ∈ {1, 2, 3, ..., m − 1} do
    AGGREGATE-IF-EQUAL(a_i, a_{i+1})
  end for
else
  AGGREGATE-IF-EQUAL(a_1, a_2)
end if
```

Theorem 3. *Algorithm 6 correctly aggregates its input using $\mathcal{O}(m \log m)$ equality tests in $\mathcal{O}(\log m)$ rounds.*

Proof (Sketch). We observe that on inputs of the same length, ODD-EVEN AGGREGATION runs AGGREGATE-IF-EQUAL twice as many times as ODD-EVEN MERGE runs COMPARE-EXCHANGE. The performance follows from Theorem 1.

The proof of correctness is by induction on the input length. We claim that the aggregated entry for a key x that occurs in the input with non-zero weight will be stored at the first position where x occurred in the input, i.e. the smallest i such that $x_i = x$. The base case of input length 2 is easily verified.

We observe that since AGGREGATE-IF-EQUAL compares the keys of elements, calling it on items with different keys does not cause problems. As it overwrites the weight with 0 when merging, calling it on a position which has already been aggregated is also harmless. Thus, our main concern is proving that all entries with the same key will indeed be aggregated into the first entry with that key.

The two recursive calls (on the odd and even sub-sequences) are on sorted sequences, so by induction the two sub-sequences are correctly aggregated. For a key x occurring in the input, let i be the least even i such that $x_i = x$, and let j be the least odd j such that $x_j = x$ in the input. By induction, the weights for x are aggregated into positions i and j. As the input was sorted by key, we have $|i - j| = 1$, and thus the two entries are merged by the for-loop calling AGGREGATE-IF-EQUAL. As this holds for any x, it concludes the proof. □

4.3 Algorithm for Weighted Set Intersection

Given the algorithm for the aggregation step on a sorted list, the key building blocks for the weighted set intersection algorithm are in place. To handle the "re-

moved" elements in the output after the aggregation step, we sort the aggregated list and count the number of non-zero items in the list, which is the number of elements to keep as output. We present the full algorithm in Algorithm 7.

Algorithm 7 MPC Weighted Set Intersection

Input: Party P_i contributes a list $\boldsymbol{a}^i$ of item-weight pairs (x_j, y_j) of length m_i.
Output: Sequence of item-weight pairs, sorted by value
Each party P_i shares its input element-wise, $[\boldsymbol{a}^i] = [x_1^i], [y_1^i], [x_2^i], [y_2^i], \ldots [x_{m_i}^i], [y_{m_i}^i]$
Let $\boldsymbol{a}$ be the concatenation of the sequences $\boldsymbol{a}^i$
Odd-even merge sort($\boldsymbol{a}$), sorting on x_j
Odd-even aggregation($\boldsymbol{a}$)
Odd-even merge sort($\boldsymbol{a}$), sorting on y_j
$[m] \leftarrow 0$
for $i = 1$ to $|\boldsymbol{a}|$ **do**
  $[m] \leftarrow [m] + (1 - [x_i = 0])$
end for
Open $[m]$ to all parties
return The m last elements of $\boldsymbol{a}$

Theorem 4. *Algorithm 7 correctly and privately (with public input lengths) solves the weighted set intersection problem using* $\mathcal{O}\big(m \log^2 m\big)$ *comparisons and* $\mathcal{O}(m \log m)$ *equality tests in* $\mathcal{O}\big(\log^2 m\big)$ *rounds.*

Proof (Sketch). Correctness of the algorithm follows from Theorems 1 and 3 (correctness of sorting and aggregation).

The security of the protocol relies on the security of the implementation of the underlying MPC primitives for all MPC operations. The number of elements shared by each party reveals the length of their input. After contributing their input, all parties perform the same operations. The only value opened to the parties is the size of the output, which is part of the output. □

5 Cooperative Anomaly Detection

Our motivating application is that of cooperative processing of anomaly logs produced by intrusion detection systems (IDS). We consider a scenario where a number of parties (organizations) collaborate on distributed intrusion detection by contributing anomaly indications based on local IDS logs. The anomaly indications are *(IP,weight)* pairs of suspected attackers, where the weight indicates the confidence in that rating. In our sample application, the weights are simple counts that can indicate anomalies, such as the number of TCP SYN packets observed from a given IP. More complex weights can be constructed from combinations of multiple criteria. The cooperating parties are mutually distrustful and reluctant to reveal any part of the characteristics of their IP traffic. However, they recognize the benefits of sharing their anomaly data. Our *top-k* protocol

from Section 4 can be utilized to produce an indication of the k system-wide most suspicious source IP addresses, while preserving the privacy of their inputs.

We use as our working example a *slow port scan* attack against multiple victims (our cooperating parties). In a slow scan, a relatively small set of attackers port scans a large set of potential victims in a pattern, which is likely to evade detection by the IDS systems of any single one, at least in terms of a strongly positive result. However, the attack may well become obvious when observing traffic aggregates.

We assume a system consisting of three privacy peers, consistent with the Sharemind MPC platform. Each cooperating party submits a list of potentially suspicious source addresses – *(IP,weight)* pairs – in secret shared form to the three privacy peers. The processing is performed on the MPC platform, i.e. by the three privacy peers, preserving the privacy of individual contributions. The final top-k potential attackers are thus not traceable to any particular contributor, but can be utilized to set policies to respond to the possible threats, e.g. by pro-actively blocking suspicious IP addresses.

MPC protocols traditionally require highly connected networks and a high messaging overhead. Thus, techniques of this sort have generally been dismissed as being impractical for scalable real-world applications, such as the one considered here. However, the privacy peer paradigm, as well as the development of general-purpose MPC platforms, such as Sharemind, mean that truly secure cooperative log processing is becoming feasible. The phenomenon of victim clustering into relatively well-defined and static groups, as observed by Katti *et al.* [22], further supports the feasibility of MPC techniques in real-life network security applications.

6 Performance Evaluation

We demonstrate the feasibility of our secure sorting protocol from Section 3, vectorized odd-even merge sort, in a real-life setting by a proof-of-concept implementation on the Sharemind [6] secure multi-party computation platform. Sharemind allows privacy preserving processing of inputs from three *privacy peers*, secure against a corruption of any single one by a passive adversary.

The Sharemind framework has its own programming language, *SecreC*, which we used to implement our sorting algorithm. For comparison purposes, we also implemented MPC sorting based on bubble sort and evaluated its performance on the same cluster. Both vectorized and serialized bubble sort implementations were produced. Our implementations are fully unrolled, meaning that the SecreC code contains the full program sequence without iterations or recursive calls.

Experiments were performed on a cluster operated by the Sharemind team, consisting of 3 computers running version 2.0 of the Sharemind framework. The computers were equipped with dual Intel X5670 2.93 GHz CPUs and 48 GB RAM, and were interconnected by gigabit links.

In Figure 1, we show the total execution time of the three algorithms as a function of n, the total number of items sorted. Our results show that 2^{14} 32-bit

values can be securely sorted in little more than 3.5 minutes using the algorithm from Section 3. The implementation is only moderately optimized, allowing us to conclude that the performance can be improved. The baseline implementation of the non-vectorized bubble sort algorithm can sort 2^6 elements in a few minutes. Parallelization of operations increases performance considerably, as expected, allowing sorting of 2^{10} elements within a similar time frame.

As expected, a faster sorting algorithm allows us to consider larger input sizes. The performance of our odd-even merge sort implementation in this proof-of-concept test leads us to conclude that MPC-based cooperative processing of several thousands of reported events per minute can indeed be practical in a real-world distributed IDS application.

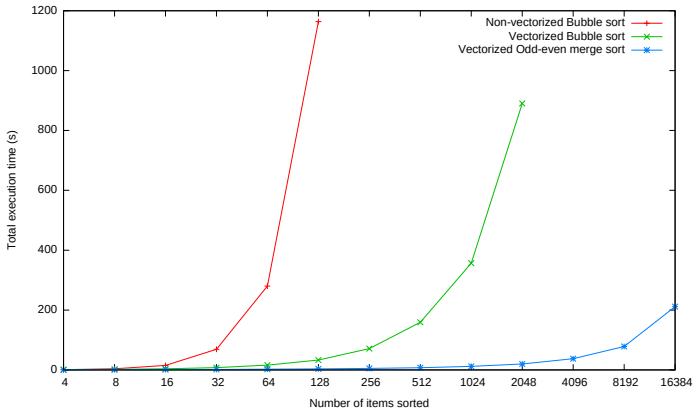

Fig. 1. Execution time (wall time) of the three compared sorting implementations. Number of elements on x-axis on log-scale.

We only present experimental results of our sorting algorithm, but the results are indicative for the feasibility of our proposed approach to cooperative anomaly detection. In particular, the run-time of our weighted set intersection protocol is dominated by sorting (twice), so its performance is closely connected to that of sorting.

7 Conclusion

We give a MPC sorting algorithm, based on the principles of sorting networks and generic MPC primitives, which preserves the privacy of individual contributors. The performance and practicality of the algorithm was assessed using a proof-of-concept implementation on the Sharemind MPC platform. Our results indicate that our MPC sorting algorithm is practical for large inputs.

We describe an algorithm which utilizes MPC sorting to produce *top-k* aggregates of weighted inputs. Further, we describe the application of the *top-k*

algorithm to distributed anomaly detection – the problem of secure aggregation of anomaly reports from multiple cooperating, but mutually suspicious, parties.

In the context of intrusion detection, there is much to be gained by collaborating. However, the sensitive nature of alerts from IDS systems makes it difficult to do so. MPC protocols offer very strong security guarantees, and as we have shown in our evaluation, are approaching practical performance for distributed intrusion detection applications.

Acknowledgments

We would like to thank the Sharemind team for their valuable feedback on our work, and for the help with the experiments. We would like to thank Marina Blanton for informing us of several additional references.

References

1. Miklós Ajtai, János Komlós, and Endre Szemerédi. An O(n log n) sorting network. In *STOC*, pages 1–9. ACM, 1983.
2. Kenneth E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, volume 32 of *AFIPS Conference Proceedings*, pages 307–314. Thomson Book Company, Washington D.C., 1968.
3. Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *STOC*, pages 503–513. ACM, 1990.
4. Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: a system for secure multi-party computation. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *ACM Conference on Computer and Communications Security*, pages 257–266. ACM, 2008.
5. Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *STOC*, pages 1–10. ACM, 1988.
6. Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In Sushil Jajodia and Javier López, editors, *ESORICS*, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206. Springer, 2008.
7. Dan Bogdanov, Margus Niitsoo, Tomas Toft, and JanWillemson. Improved protocols for the Sharemind virtual machine. Technical Report T-4-10, Cybernetica, 2010.
8. Peter Bogetoft, Ivan Damgård, Thomas P. Jakobsen, Kurt Nielsen, Jakob Pagter, and Tomas Toft. A practical implementation of secure auctions based on multiparty integer computation. In Giovanni Di Crescenzo and Aviel D. Rubin, editors, *Financial Cryptography*, volume 4107 of *Lecture Notes in Computer Science*, pages 142–147. Springer, 2006.
9. Martin Burkhart and Xenofontas Dimitropoulos. Fast privacy-preserving top-k queries using secret sharing. In *19th International Conference on Computer Communications and Networks (ICCCN)*, Zurich, Switzerland, August 2010.
10. Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas Dimitropoulos. SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In *19th USENIX Security Symposium*, Washington, DC, USA, August 2010.

11. David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In *STOC*, pages 11–19. ACM, 1988.
12. Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In Shai Halevi and Tal Rabin, editors, *TCC*, volume 3876 of *Lecture Notes in Computer Science*, pages 285–304. Springer, 2006.
13. Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous multiparty computation: Theory and implementation. In Stanislaw Jarecki and Gene Tsudik, editors, *Public Key Cryptography*, volume 5443 of *Lecture Notes in Computer Science*, pages 160–179. Springer, 2009.
14. Ivan Damgård and Yuval Ishai. Constant-round multiparty computation using a black-box pseudorandom generator. In Victor Shoup, editor, *CRYPTO*, volume 3621 of *Lecture Notes in Computer Science*, pages 378–394. Springer, 2005.
15. Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly secure oblivious RAM without random oracles. In Yuval Ishai, editor, *TCC*, volume 6597 of *Lecture Notes in Computer Science*, pages 144–163. Springer, 2011.
16. Hervé Debar, Marc Dacier, and Andreas Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 31:805?822, 1999.
17. Wenliang Du and Mikhail J. Atallah. Secure multi-party computation problems and their applications: A review and open problems. In Victor Raskin, Steven J. Greenwald, Brenda Timmerman, and Darrell M. Kienzle, editors, *NSPW*, pages 13–22. ACM, 2001.
18. Michael J. Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In Christian Cachin and Jan Camenisch, editors, *EUROCRYPT*, volume 3027 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2004.
19. Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, New York, NY, USA, 2004.
20. Michael T. Goodrich. Randomized shellsort: A simple oblivious sorting algorithm. In Moses Charikar, editor, *SODA*, pages 1262–1277. SIAM, 2010.
21. Somesh Jha, Louis Kruger, and Vitaly Shmatikov. Towards practical privacy for genomic computation. In *IEEE Symposium on Security and Privacy*, pages 216–230. IEEE Computer Society, 2008.
22. Sachin Katti, Balachander Krishnamurthy, and Dina Katabi. Collaborating against common enemies. In *Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*, IMC '05, pages 34–34, Berkeley, CA, USA, 2005. USENIX Association.
23. Lea Kissner and Dawn Xiaodong Song. Privacy-preserving set operations. In Victor Shoup, editor, *CRYPTO*, volume 3621 of *Lecture Notes in Computer Science*, pages 241–257. Springer, 2005.
24. Frank Thomson Leighton and C. Greg Plaxton. Hypercubic sorting networks. *SIAM J. Comput.*, 27(1):1–47, 1998.
25. Patrick Lincoln, Phillip Porras, and Vitally Shmatikov. Privacy-preserving sharing and correction of security alerts. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, pages 17–17, Berkeley, CA, USA, 2004. USENIX Association.
26. Dilip Many. Privacy-preserving collaboration in network security. Master's thesis, ETH Zürich, 2009.
27. Takashi Nishide and Kazuo Ohta. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In Tatsuaki Okamoto and

Xiaoyun Wang, editors, *Public Key Cryptography*, volume 4450 of *Lecture Notes in Computer Science*, pages 343–360. Springer, 2007.
28. Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.
29. Donald L. Shell. A high-speed sorting procedure. *Commun. ACM*, 2(7):30–32, 1959.
30. Adam J. Slagell and William Yurcik. Sharing computer network logs for security and privacy: A motivation for new methodologies of anonymization. *CoRR*, cs.CR/0409005, 2004.
31. Guan Wang, Tongbo Luo, Michael T. Goodrich, Wenliang Du, and Zutao Zhu. Bureaucratic protocols for secure two-party sorting, selection, and permuting. In Dengguo Feng, David A. Basin, and Peng Liu, editors, *ASIACCS*, pages 226–237. ACM, 2010.
32. Dingbang Xu and Peng Ning. Privacy-preserving alert correlation: a concept hierarchy based approach. In *21st Annual Computer Security Applications Conference*, 2005.
33. Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164. IEEE, 1982.
34. Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, pages 162–167. IEEE, 1986.