Fast Privacy-Preserving Top-k Queries using Secret Sharing

(Invited Paper)

Martin Burkhart
ETH Zurich
burkhart@tik.ee.ethz.ch

Xenofontas Dimitropoulos
ETH Zurich
fontas@tik.ee.ethz.ch

***Abstract*—Over the past several years a lot of research has focused on distributed top-k computation. In this work we are interested in the following privacy-preserving distributed top-k problem. A set of parties hold private lists of key-value pairs and want to find and disclose the k key-value pairs with largest aggregate values without revealing any other information. We use secure multiparty computation (MPC) techniques to solve this problem and design two MPC protocols, *PPTK* and *PPTKS*, putting emphasis on their efficiency. *PPTK* uses a hash table to condense a possibly large and sparse space of keys and to probabilistically estimate the aggregate values of the top-k keys. *PPTKS* uses multiple hash tables, i.e., sketches, to improve the estimation accuracy of *PPTK*. We evaluate our protocols using real traffic traces and show that they accurately and efficiently aggregate distributions of IP addresses and port numbers to find the globally most frequent IP addresses and port numbers.**

I. Introduction

In many cases, a set of parties would like to aggregate private key-value lists and find the keys with the highest aggregate values. This problem is ubiquitous having applications in many fields, including network monitoring, security analysis, and distributed databases. Our work is motivated by a simple network profiling application. A NetFlow collector computes top-k traffic feature reports, such as top-10 port numbers, and we would like to compare local top-k reports with aggregate reports corresponding to the data of multiple cooperating collectors. Comparing, for example, local with aggregate top-10 port numbers is useful for profiling trends and for troubleshooting. Another application is aggregating IDS alerts from multiple organizations to find the most frequent alerts without relying on a trusted third party, like DShield [1] that presently provides this service relying on organizations offering to share their alerts. Moreover, organizations maintaining reputation scores for identifiers, like AS numbers, IP addresses, or domain names, can aggregate their scores to find the overall top-scored identifiers. Unfortunately, these and other similar distributed problems lack practical solutions that preserve the privacy of the involved data.

For several years researchers have been studying MPC techniques that enable a set of parties to compute a public function over their private data in a way that the computation only reveals the final value of the public function and nothing more about the private data. Although MPC techniques provide powerful privacy guarantees, until recently they were primarily of theoretical interest as practical implementations required significant running time and communication overhead. Nevertheless, recent developments show that MPC techniques can also have practical computational overhead. For example, the SEPIA library we developed recently [2], [3] provides MPC primitives and protocols for networking applications that can efficiently process input data in near real-time.

In this work we design two MPC protocols tailored for near real-time top-k computation. With our *PPKT* protocol, parties put keys into a local hash table to produce a compact representation of a possibly large and sparse space of keys, such as the space of IP addresses. Then they use MPC primitives to aggregate the local hash tables, to resolve collisions, and to estimate the top-k key-value pairs. A main idea of our approach is that MPC operations are applied to fixed-length hash tables, which reduces computational overhead by avoiding expensive key comparison operations. The *PPTKS* protocol extends *PPTK* using multiple hash tables, i.e., sketches, to reduce the number of collisions experienced by top-k keys and to further improve the estimation accuracy of *PPTK*. Both *PPTK* and *PPTKS* are optimized for parallel invocations and use the honest-but-curious adversary model, in which parties follow the protocol, but may use learned information to invade privacy.

We evaluate the accuracy of our protocols using traffic traces from an ISP and find that they can very accurately compute top-k reports of IP addresses and port numbers. For example, in the challenging case of IP addresses, *PPTK* using hash tables with 10,000 entries aggregates 180,000 IP addresses and estimates the top-10 with 99.7% accuracy, whereas *PPTKS* using pairs of hash tables with 1,000 entries estimates the top-100 IP addresses with 98.2% accuracy. Finally, we benchmark the running time of our protocols and find that they need only few minutes to compute aggregate top-k reports. For example, 8.8 minutes are needed to accurately estimate the top-100 IP addresses, while substantial less time is required for port numbers and top-10 reports.

In the next section, we briefly discuss related work. In Section III we provide some background information on MPC and we introduce and evaluate our protocols in Sections IV and V. Finally, in Section VI we benchmark the running time of our protocols and in Section VII we conclude this paper.

II. Related Work

A lot of work [4], [5], [6], [7], [8], [9] has focused on algorithms for answering distributed one-time top-k queries in the non-privacy-preserving setting, where the main goal is to minimize communication cost. In particular, these works assume a set of parties holding local lists of items and corresponding values. A top-k query then finds the items with the top-k aggregate values using typically exact methods. For example, with the well-known algorithm by Fagin [4], parties output local items in a descending order until the output has k items in common. The candidate items in the output are then guaranteed to include the top-k. On the other hand, a number of related works, such as the paper by Babcock and Olston [10], adopt the data streaming model, where each party locally observes an online stream of items and corresponding values, and use approximate methods to answer top-k monitoring queries, which continuously report the k largest values obtained from distributed data streams.

Most related to our work, Vaidya and Clifton [9], [11] study the problem of finding the top-k matching items over vertically-distributed private data, where each party holds different item attributes. They use privacy-preserving data mining and secure multiparty computation techniques to extend Fagin's algorithm [4]. Although Vaidya's algorithm can solve our problem, the number x of disclosed candidate items can be very large if item sets are disjoint to some degree. The assumption made in [9] is that all items are reported by all sites. However, this is not the case with our type of data, i.e., port and IP address distributions. In our experiments, on average only 114 out of 124,000 IP addresses are common in all 6 sites. Therefore, for finding the top say 120 IP addresses, Vaidya's algorithm would disclose the complete set of IP addresses ($x = 124,000$), since the output never has 120 items in common. Furthermore, also the running time of the algorithm scales with x. The authors recently implemented their algorithm [11] with Fairplay [12], which for $x = 10,000$ requires more than 10 hours for a two-party computation.

The work by Xiong [13] *et al.* introduced an algorithm for finding the k largest values among parties holding private sets of values. The protocol preserves privacy in a probabilistic way by randomizing values before distributing them among parties and avoids using cryptographic primitives. In contrast, our work focuses on the more challenging problem of aggregating items and finding the top-k aggregate items. Finally, Roughan and Zhang [14] identified several networking problems, for which MPC is useful and highlighted that the sketch data structure can be easily integrated with MPC due to its linearity property.

Our approach 1) answers distributed one-time top-k queries probabilistically, 2) uses sketches to enable very efficient MPC protocols and to accurately estimate the top-k items, and 3) provides strong privacy guarantees. In addition, we use standard secret sharing techniques that are supported by a number of available frameworks, such as SEPIA [2], VIFF [15], FairplayMP [16], and Sharemind [17], and avoid applying expensive homomorphic encryption.

III. Preliminaries

Following [16] and [2], we assume two types of parties: n input nodes (IN) and m computation nodes (CN). The INs are the parties that want to compute the top-k items over their datasets. The CNs help the INs by performing private computations on shared secrets. The level of privacy and running time scale with the number of CNs m. Hence, the separation into INs and CNs allows to tune the privacy-performance tradeoff independently of the number of input nodes. In practice, a CN can be hosted together with an IN in the same network.

We assume a secret sharing scheme offering the primitives listed in table I. A common way of realizing these basic primitives is Shamir's secret sharing scheme [18]. In order to share a secret, a random polynomial over a prime field is generated and the shares correspond to different points on this polynomial. A secret is reconstructed from shares by interpolating the polynomial. Private addition of secrets is done by simply adding individual shares, whereas private multiplication requires an intermediate step of information exchange between the CNs [19]. Private comparison operations are typically very expensive and are built by composing additions and multiplications, e.g., [20].

Several frameworks offer these primitives [16], [15], [17], [2] and can be used directly to implement the protocol presented in this paper. The performance of individual primitives depends on the design and implementation in the specific framework. Recently, we developed the SEPIA library [2] with the goal of optimizing performance for networking applications, where input data is typically voluminous and computation should finish in near real-time, i.e., fully processing an x-minute interval of traffic data does not take longer than x minutes. For parallel invocations of operations, SEPIA proves to be very efficient. For instance, with 5 CNs, it performs 82,730 multiplications, 2,070 equality checks, and 86 less-than comparisons per second. The $lessThan$ operation is by far the most expensive in terms of computation and communication cost. For this reasons, in the design of our protocol we aim at using as few $lessThan$s as possible and at parallelizing as many operations as possible, i.e., increasing the number of operations executed between consequative sychronization rounds.

The privacy guarantees of the protocol depend on the secret sharing scheme and the adversary model. There are two basic adversary models: The *semi-honest* model (supported by all frameworks) and the *malicious* model (only supported by VIFF [15]). In the semi-honest model, adversarial CNs follow the protocol, but try to learn as much as possible from the information they receive, also by combining their local information. In the malicious model, adversarial nodes may arbitrarily deviate from the protocol. For instance, in Shamir's secret sharing scheme, which is the basis of SEPIA [2] and VIFF [15], with semi-honest adversaries, input data remains

Name	Syntax	Output	Description
sharing	$share(s)$	$[s]$	A secret value s held by an IN is split up in m shares s_i. Each s_i is then distributed to CN i. The ensemble of all distributed shares $\{s_1, s_2, \cdots, s_m\}$ is called a *sharing* of s and denoted by $[s]$.
reconstruction	$recon([s])$	s	The individual shares of a sharing $[s]$ are combined to reconstruct the secret value s.
addition	$[a]+[b]$, $[a]+b$	$[a+b]$	Adds two sharings (or a sharing and a public value) to get a sharing of the sum. This also includes subtraction ('-').
multiplication	$[a]*[b]$, $[a]*b$	$[a*b]$	Multiplies two sharings (or a sharing and a public value) to get a sharing of the product.
equal	$equal([a],[b])$, $equal([a],b)$	if $(a == b)$ then [1] else [0]	Two sharings (or a sharing and a public value) are compared for equality. The output remains secret.
less-than	$lessThan([a],[b])$, $lessThan([a],b)$	if $(a < b)$ then [1] else [0]	Two sharings (or a sharing and a public value) are compared for size. The output remains secret.

TABLE I: MPC primitives required for our protocol.

secret as long as the majority of CNs is honest. With malicious adversaries (only supported by VIFF), correctness and privacy of input data is guaranteed as long as less than $1/3$ of the CNs is malicious [21].

IV. TOP-k PROTOCOL PPTK

In this section we introduce the privacy-preserving top-k protocol *PPTK* and evaluate its accuracy with real traffic data.

A. Input data

Each IN locally holds a set of *items*. An item is defined by an identifying key and a corresponding value. The goal of the protocol is to compute the k items with the biggest aggregate values over all input sets, without disclosing information about non-top-k items. The aggregate value of an item is simply the sum of its local values. Also, the protocol should not reveal which INs contribute to a top-k item.

First, the INs store keys and values of their items in one-dimensional hash arrays of size H, which we call hash tables. Hash values are generated using a public hash function h with $h(x) \in [0, \dots, H-1]$. The precise choice of the table size H involves a tradeoff between accuracy and performance. The smaller H is chosen, the more collisions occur and the less accurate are the computations. The bigger H is chosen, the more MPC operations need to be performed on vectors of size H.

The INs generate a vector of keys $\mathbf{k} = \{k_0, k_1, \cdots, k_{H-1}\}$ and values $\mathbf{v} = \{v_0, v_1, \cdots, v_{H-1}\}$, initialized to zero. Then, for each item, they store key and value at the position given by the hash of the key. If local collisions occur, the node only reports key and value of the item with the bigger value. That is, for each item a the INs do:

$$\left.\begin{array}{r} v_{h(key(a))} = value(a) \\ k_{h(key(a))} = key(a) \end{array}\right\} \quad \text{if} \quad value(a) > v_{h(key(a))}$$

Note that we use hashes solely for storing items from a typically large and sparse space of keys in a more compact form, which makes private aggregation much more efficient as items with the same key fall into the same bucket. We do *not* rely on the privacy properties of hash functions. While it is generally hard to infer x from $h(x)$, it can be practically feasible for small domains. For instance, if x is an IPv4 address, it is relatively easy to brute-force the entire address range and find x from $h(x)$. Hence, in our protocol, the privacy of input values is solely protected by the secret sharing scheme.

B. Aggregation

Each IN shares the vectors $\mathbf{k}$ and $\mathbf{v}$ among the CNs. Let $[k_j^i]$ and $[v_j^i]$ be the sharings at position j in the key and value vectors of IN i, respectively. Then, the CNs aggregate the values for all items simply by building the sum over all n inputs:

$$[V_j] = [v_j^1] + [v_j^2] + \cdots + [v_j^n]$$

The value V_j is an approximation of the aggregate value of the largest item that is hashed at position j. The vector $\mathbf{V}$ contains the aggregated values for all positions. Local and global collisions introduce error. If local collisions occur, a node selects the bigger item. Thus, the dropped item loses support in the global aggregation and its value is underestimated. Global collisions occur if the keys of two INs, say nodes 1 and 2, at position j differ, i.e., $k_j^1 \neq k_j^2$. We use a collision resolution algorithm (see Alg. 2) to detect global collisions for the top-k items and to correct the introduced error. As we show in the evaluation section, these errors can be easily manipulated to become arbitrarily small.

C. Finding the top-k items

The CNs now hold sharings of all aggregate item values. The next step is to find the k biggest values in $[\mathbf{V}]$. The keys of these values represent the sought top-k items. The basic idea following [9] and [22] is to identify a threshold value τ that separates the k-th from the $(k+1)$-th item by performing a binary search over the range of values. For each candidate threshold, the number of values above the threshold are privately computed and compared to k. The threshold is increased if more than k items are larger, otherwise it is decreased. Once the correct threshold τ is found, all values greater than τ are reconstructed. For sake of simplicity, we assume that all values are different. We denote the maximum expected value by M. Then, the binary search for τ is guaranteed to finish in $\log_2 M$ rounds. If we have an estimate of τ based on past observations, we can, of course, reduce average search time.

Algorithm 1 Find top-k elements and their indices in a vector of sharings.

```
Require: A shared vector $[\mathbf{V}]$ with elements $[V_0]$ to $[V_{H-1}]$
1: $match = 0$
2: $lbound = 0$ {lower bound}
3: $ubound = M$ {upper bound}
4: while $match == 0$ do
5:     $\tau = \lceil (lbound + ubound)/2 \rceil$
6:     $[biggercount] = share(0)$
7:     for $j = 0$ to $H - 1$ do
8:         $[lt_j] = lessThan([V_j], \tau)$
9:         $[biggercount] = [biggercount] + (1 - [lt_j])$
10:    end for
11:    $match = recon(equal([biggercount], k))$
12:    if $match == 0$ then
13:        $toohigh = recon(lessThan([biggercount], k))$
14:        if $toohigh == 1$ then
15:            $ubound = \tau$
16:        else
17:            $lbound = \tau$
18:        end if
19:    end if
20: end while
21: $count = 0$ {start item reconstruction}
22: for $j = 0$ to $H - 1$ do
23:     if $recon([lt_j]) == 0$ then
24:         $topi_{count} = j$
25:         $topv_{count} = recon([V_j])$
26:         $count = count + 1$
27:     end if
28: end for
29: return $(\mathbf{topi}, \mathbf{topv})$
```

The detailed algorithm is given in Alg. 1. The algorithm does not reconstruct the intermediate threshold value, but only decides whether the threshold is too high or not. The algorithm requires $(H+1)\log_2 M$ invocations of $lessThan$ and $\log_2 M$ invocations of $equal$. The vast majority of these operations are independent and can be performed in parallel.

D. Resolving global collisions

Once we have the top k values and their indices, we resolve global collisions and reconstruct the top-k keys. To compute a top-k value $[V_j]$, we aggregated n local values from the INs that correspond to up to n different keys. Which of these keys should be the key assigned to $[V_j]$? In this step we find the key with the biggest aggregate value and adjust $[V_j]$ by subtracting all contributions coming from colliding keys. Alg. 2 gives the details of our global collision resolution protocol.

Algorithm 2 Reconstruct key with the biggest contribution to the aggregate value in position j (j is fixed for one run).

```
Require: Shared keys $[k_j^1], [k_j^2], \cdots, [k_j^n]$, and corresponding values
         $[v_j^1], [v_j^2], \cdots, [v_j^n]$.
1: for $x = 1$ to $n$ do
2:     for $y = x + 1$ to $n$ do
3:         $[e_{xy}] = equal([k_j^x], [k_j^y])$ {Note that $e_{xy} == e_{yx}$}
4:         $[sum] = [v_j^x] + [v_j^y]$
5:         $[v_j^x] = [e_{xy}] * [sum] + (1 - [e_{xy}]) * [v_j^x]$
6:         $[v_j^y] = [e_{xy}] * [sum] + (1 - [e_{xy}]) * [v_j^y]$
7:     end for
8: end for
9: $[maxv] = [v_j^1]$ {store the max value}
10: $[maxk] = [k_j^1]$ {store key with max value}
11: for $x = 2$ to $n$ do
12:     $[comp] = lessThan([maxv], [v_j^x])$
13:     $[maxv] = [comp] * [v_j^x] + (1 - [comp]) * [maxv]$
14:     $[maxk] = [comp] * [k_j^x] + (1 - [comp]) * [maxk]$
15: end for
16: return $(maxk, maxv)$
```

The algorithm requires $n(n-1)/2$ invocations of $equal$ and $n - 1$ invocations of $lessThan$. All the computations of the intermediate values $[e_{xy}]$ in line 3 are independent and can be performed in parallel. The algorithm needs to be called once for each of the k top-k items. All of these calls are also independent and can be executed in parallel. Note that although the algorithm needs $O(n^2)$ invocations of $equal$, the number of INs n will typically be small, i.e., much smaller than H or the number of items aggregated.

A refinement of the collision resolution protocol using sorting and binary search for aggregating keys would reduce the complexity from $O(n^2)$ to $O(n \log n)$. However, designing an efficient private sorting algorithm is a challenge on its own and left for future work. Another variation, that reconstructs all of the potentially conflicting keys k_j^i (without their values), allows to dismiss the private $equal$ and multiplication operations in the nested for-loop in lines 1-8 and removes the quadratic dependency on n. However, this variant leaks all the keys that collide with a specific key and also the IN that reported these keys.

E. Accuracy

Depending on H and the item distribution, local and global hash collisions can introduce error. In this section, we use real traffic data to evaluate the accuracy of the top-k items found by *PPTK*. We ran simulations on TCP destination port and IP address distributions of the six biggest customer networks of SWITCH [23]. In the scenario, the six customers want to compute the aggregate top-k items over their datasets. We used traffic from a whole day in August 2007, resulting in 96 timeslots of 15 minutes. For each timeslot, a new random seed for the hash function was used. We computed the correct set σ of aggregate top-k items and the approximate set $\widetilde{\sigma}$ as calculated by our protocol. We then compared the two to evaluate the accuracy of *PPTK* using the following metrics:

1) Percentage of correct top-k items: $|\sigma \cap \widetilde{\sigma}|/k$.
2) The average rank distortion of items in $\sigma \cap \widetilde{\sigma}$, where the rank distortion of an item is the absolute difference between its true and approximate rank. For instance, if the rank 1 item is reported in rank 4, then its rank distortion is 3.
3) Percentage of items in $\sigma \cap \widetilde{\sigma}$ with *exact* values.
4) Average relative error of values that are in $\sigma \cap \widetilde{\sigma}$ but are not *exact*.

All metrics except the average rank distortion take values in the range $[0, 1]$. For metrics 1 and 3, a value of 1 is best whereas for metrics 2 and 4 a value of 0 is optimal.

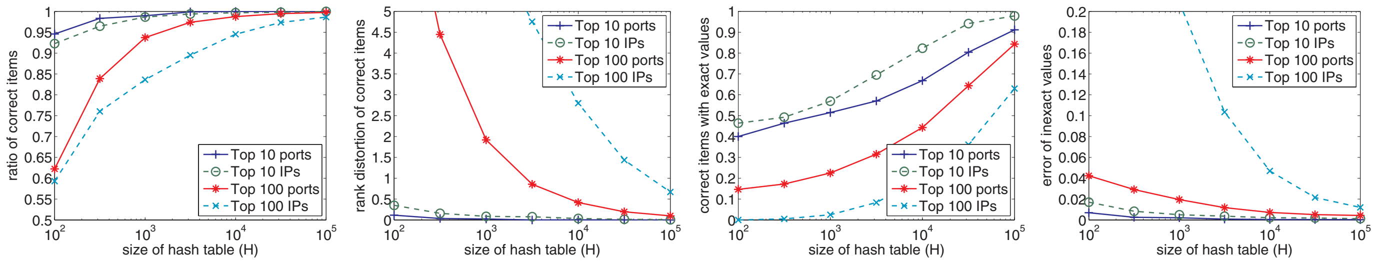


Fig. 1: Statistics for top 10/100 ports and top 10/100 IP addresses.

With port distributions, the number of distinct items N is smaller or equal to 65,535. Figure 1 shows the accuracy metrics for the computation of top-10/100 ports and IP addresses. The size of the hash table H is varied between 100 and $100,000$. For top-10 ports, rather small choices of H produce very good results. For example, for $H = 1,000$, the ratio of correctly identified top-k items is 98.95% and the rank distortion of these items is only 0.023. 51.4% of the items have an exact values while the remaining have an average relative error of only 0.2%. This is surprising as H is very small compared to N in this case. If we raise H to $10,000$, the ratio of correct items goes to 1.

Accurately reconstructing the top-100 ports requires bigger choices of H. In a heavy-tailed distribution, the top-10 items are much more stable with values usually further apart from each other compared to items with ranks between 90 and 100. Therefore, the fluctuation in ranks 90 to 100 is much higher and small inaccuracies have more impact. Still, a choice of $H = 10,000$ reconstructs 98.7% of the top-100 ports correctly, with 41.7% exact values and only 0.7% error on the non-exact values. The rank distortion is expected to be higher than for top-10 computation, as the maximum distortion per item is 100 instead of 10. However, the average rank distortion of 0.41 in the top-100 is acceptable, especially as most of this distortion is due to the lower ranked items.

The number of distinct IP addresses in the aggregate distribution varies between 70,000 during the night and approximately 180,000 during the day. As with ports, relatively small values of H lead to very good results for the top-10 items. For $H = 10,000$, 99.7% of the top-10 IP addresses are reconstructed correctly. For those with non-exact values, the average error is only 0.2%. However, for the top-100 addresses, only $H = 31,600$ (or higher) leads to acceptable results, i.e., 97.3% correct items with 36.1% correct values and 2.1% error in the non-exact values.

V. Top-k protocol PPTKS

The running time of *PPTK* scales linearly with H, requiring $O(H)$ *lessThan* operations. Therefore, if we choose $H = 10,000$ instead of $H = 1,000$, we improve accuracy but also entail a 10-fold running time increase. In this section we improve the accuracy of *PPTK* by using sketches, leading to a better tradeoff between accuracy and running time. Instead of a single hash table of size H, we use S hash tables with pairwise independent hash functions. The advantage of using two or more hash tables is that the probability of observing the same hash collisions in multiple tables is very low. Therefore, multiple tables allow to correct errors introduced by hash collisions in single tables.

A. Protocol PPTKS

With *PPTK*, collisions always lead to an underestimation of item values. If a collision occurs locally, the IN drops the smaller key, which leads to an underestimation of this key's value. If a collision occurs globally, our collision resolution protocol reconstructs only the key with the largest value. All other keys are dropped, again leading to an underestimation of their values. Therefore, the accuracy of values can be monotonically increased by adding more hash tables and by always selecting the maximum value for each key. For example, if we have 3 tables and the aggregate counts for port 80 are 15,176, 16,002, and 15,995, then we know that 16,002 is closest to the true count of port 80 and that for the other counts some contributions to port 80 were dropped due to collisions.

PPTKS, the sketch-version of *PPTK* proceeds as follows:

1) Run *PPTK* S times using pairwise independent hash functions.
2) For each distinct key in the S top-k sets, store the maximum occurring value.
3) Sort the new items by descending value and return the largest k items.

B. Analysis of reconstructed information

Besides the final top-k items, *PPTKS* discloses some additional information. An item appearing in two or more of the S top-k lists may have different value estimates. The difference between two estimates indicates a contribution that was dropped. However, it does not reveal any other information, like the identity or number of INs that reported the item.

Assume an item that is not part of the true top-k set but appears in one or more of the S approximate top-k sets. The item could only slip in one of the S top-k sets because a true top-k item was underestimated. This means, that the false-positive item for which we learn its approximate value must be directly following the true top-k items. If we compute the top-100 items and the ratio of correct items for each top-k set is 95%, then we learn (on average) information about the top-105 items. However, if we monitor the top-100 items continuously over time, chances are high that we will learn these values

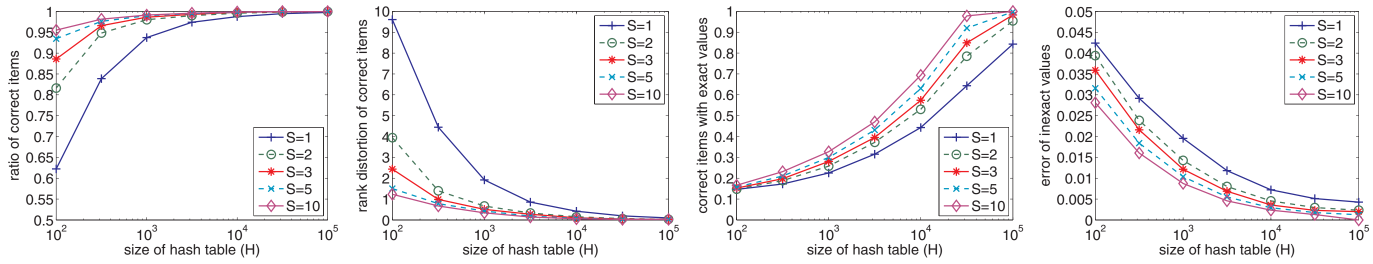


Fig. 2: Statistics for top 100 ports using sketches with S hash tables.

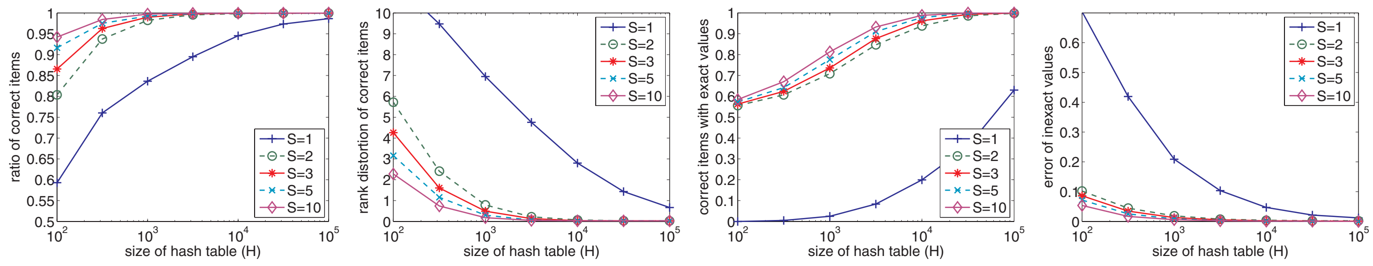


Fig. 3: Statistics for top 100 IP addresses using sketches with S hash tables.

anyway due to fluctuations around rank 100. If this additional information is considered sensitive, the aggregation of the S sets can easily be performed in MPC as well, using a slightly modified version of Alg. 2.

C. Accuracy

Figures 2 and 3 illustrate the accuracy improvements achieved by applying *PPTKS* for top-100 ports and IP addresses, respectively. Generally, *PPTKS* improves accuracy significantly for small values of H, whereas the improvement is smaller for higher H. This is intuitively clear, because for higher values of H accuracy is already high and the room for further improvement is small. Another observation is that the first additional table, i.e., increasing $S = 1$ to $S = 2$, improves accuracy significantly, while using $S > 3$ does not help as much. Particularly striking is the case of top-100 IP addresses shown in Figure 3. For $H = 1,000$, using $S = 2$ instead of $S = 1$ increases the ratio of correct items from 83.6% to 98.2%. At the same time, the rank distortion of correct items is greatly reduced from 7.0 to 0.8. Furthermore, while the ratio of exact values was as low as 2.5% for $S = 1$, it is boosted to 70.8% for $S = 2$. Last but not least, the average error of inexact values is reduced from 20.8% to 1.9%. To achieve a similar accuracy improvement with a single hash table, H would have to be raised to $100,000$, leading to 100 times longer running time. With $H = 1,000$ and 2 tables, the same improvement is achieved by merely doubling running time.

VI. Estimated running time

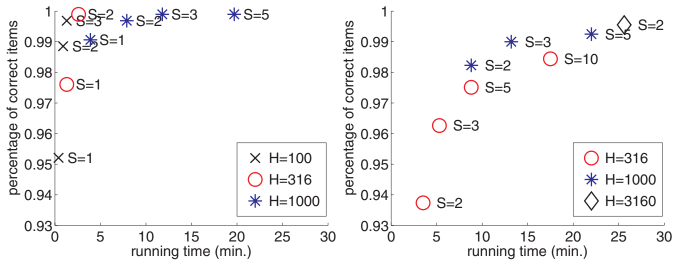
To estimate the running time of *PPTKS*, we use benchmarking results for the parallel execution of basic MPC primitives with SEPIA [2]. These results were obtained by using 5 CNs and 32-bit secrets. CNs were run on separate 2x Dual Core AMD Opteron 275 machines with 1Gb/s LAN connections. When many parallel invocations of operations are needed, SEPIA significantly outperforms other frameworks. While Sharemind is comparable regarding performance, it only supports 3 CNs, i.e., it only tolerates a single semi-honest node. For further details please refer to [2].

SEPIA is able to perform 82,730 private multiplications, 2,070 $equal$s, and 86 $lessThan$s per second. We do not consider the number of private additions because they only require local computation without communication among the CNs. Therefore, private additions are even faster than private multiplications and pose no performance bottleneck. Based on the complexity analysis in Section IV we derive the number of operations needed. Each $equal$ and $lessThan$ in Alg. 2 is followed by 4 private multiplications that use the comparison result to conditionally add/select sharings. Thus, one complete run of *PPTK* requires:

$lessThan$s:	$(H + 1)\log_2 M + k(n - 1)$
$equal$s:	$kn(n - 1)/2 + \log_2 M$
multiplications:	$4(kn(n - 1)/2 + k(n - 1))$

For *PPTKS*, these numbers have to be multiplied by S. In our settings, running time is dominated by H and S. The parameters n and k dominate the number of $equal$s, which are very fast in SEPIA and thus have only a small impact on the running time.

Figure 4 shows the estimated running time versus the accuracy, i.e., percentage of correct items, for different configurations of H and S. Note that the estimated running time is based on the assumption that operations can be performed in parallel, which is true for the vast majority of them, as different instances of *PPTK* run by *PPTKS* are completely independent and can be performed in parallel. We use the average percentage of correct items obtained in the previous sections and estimate running time for $n = 20$ INs. The



(a) Top 10 ports. (b) Top 100 IP addresses.

Fig. 4: Tradeoff of accuracy vs. running time with 20 input and 5 computation nodes. M was set to 2^{20}.

maximum item value encountered during our simulations with real traffic data was approximately $660K$ flows for port 80, therefore we used $M = 2^{20}$.

The plot shows that it is possible to achieve very good accuracy with short running times. For top-10 ports, an accuracy of 99.89% is achieved in 2.6 mins with $H = 316$ and $S = 2$. For top-100 IP addresses, an accuracy of 98.2% requires 8.8 mins with $H = 1,000$ and $S = 2$. In the same setting, raising n from 20 to 50 results in a running time of 11.7 mins, while a reduction of k from 100 to 10 reduces the running time to 7.9 mins. Note that the running time is independent of the number of items processed[1]. This nicely demonstrates that *PPTKS* is scalable with regard to the number of INs n and the number of items processed.

VII. Conclusion

In this work we designed two protocols, *PPTK* and *PPTKS*, that use MPC techniques and sketches to aggregate local key-value lists and to find the keys with the top-k aggregate values. Our protocols balance computational efficiency and estimation accuracy. In particular, we use the sketch data structure to estimate the values of the top keys and to restrict MPC operations on fixed-length vectors. Our evaluation with real traffic traces shows that our protocols achieve very good accuracy in estimating top IP addresses and port numbers. In addition, our benchmarking highlights that our protocols need less than 10 minutes to process 15-minute traffic intervals. Besides implementing our top-k protocols using the SEPIA library, in the future we plan to investigate further techniques for improving the efficiency of our protocols.

Acknowledgment

We are grateful to SWITCH for providing their traffic traces.

References

[1] DShield, "The Internet Storm Center," www.dshield.org.

[2] M. Burkhart, M. Strasser, D. Many, and X. Dimitropoulos, "SEPIA: Privacy-Preserving Aggregation of Multi-Domain Network Events and Statistics," in *19th USENIX Security Symposium*, August 2010.

[3] SEPIA, "Project webpage," http://www.sepia.ee.ethz.ch.

[4] R. Fagin, "Combining fuzzy information from multiple systems," in *ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, 1996.

[5] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," in *ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS)*, 2001.

[6] R. Akbarinia, E. Pacitti, and P. Valduriez, "Best position algorithms for top-k queries," in *international conference on Very large data bases (VLDB)*, 2007.

[7] K. Chang and S. Hwang, "Minimal probing: Supporting expensive predicates for top-k queries," in *ACM SIGMOD international conference on Management of data*, 2002.

[8] A. Marian, N. Bruno, and L. Gravano, "Evaluating top-k queries over web-accessible databases," *ACM Trans. Database Syst.*, vol. 29, no. 2, pp. 319–362, 2004.

[9] J. Vaidya and C. Clifton, "Privacy-preserving top-k queries," in *IEEE International conference on data engineering (ICDE)*, 2005.

[10] B. Babcock and C. Olston, "Distributed top-k monitoring," in *ACM SIGMOD international conference on Management of data*, 2003.

[11] J. Vaidya and C. Clifton, "Privacy-preserving kth element score over vertically partitioned data," *IEEE Trans. on Knowl. and Data Eng.*, vol. 21, no. 2, pp. 253–258, 2009.

[12] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella, "Fairplay-a secure two-party computation system," in *13th USENIX Security Symposium*, 2004.

[13] L. Xiong, S. Chitti, and L. Liu, "Topk queries across multiple private databases," in *IEEE International conference on distributed computing systems (ICDCS)*, 2005.

[14] M. Roughan and Y. Zhang, "Secure distributed data-mining and its application to large-scale network measurements," *Computer Communication Review (CCR)*, vol. 36, no. 1, pp. 7–14, 2006.

[15] I. Damgård, M. Geisler, M. Krøigaard, and J. Nielsen, "Asynchronous multiparty computation: Theory and implementation," in *Conference on Theory and Practice of Public Key Cryptography (PKC)*, 2009.

[16] A. Ben-David, N. Nisan, and B. Pinkas, "FairplayMP: a system for secure multi-party computation," in *Conference on Computer and communications security (CCS)*, 2008.

[17] D. Bogdanov, S. Laur, and J. Willemson, "Sharemind: A Framework for Fast Privacy-Preserving Computations," in *European Symposium on Research in Computer Security (ESORICS)*, 2008.

[18] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[19] R. Gennaro, M. Rabin, and T. Rabin, "Simplified VSS and fast-track multiparty computations with applications to threshold cryptography," in *ACM symposium on Principles of distributed computing (PODC)*, 1998.

[20] T. Nishide and K. Ohta, "Multiparty computation for interval, equality, and comparison without bit-decomposition protocol," in *Conference on Theory and Practice of Public Key Cryptography (PKC)*, 2007.

[21] M. Ben-Or, S. Goldwasser, and A. Wigderson, "Completeness theorems for non-cryptographic fault-tolerant distributed computation," in *ACM symposium on Theory of computing (STOC)*, 1988.

[22] G. Aggarval, N. Mishra, and B. Pinkas, "Secure Computation of the kth-Ranked Element," in *EUROCRYPT*, 2004.

[23] SWITCH, "The Swiss Education and Research Network," http://www.switch.ch.

[1]Local computation scales with the number of items, but compared to the cost of MPC operations it is negligible.