Varsha Bhat Kukkala* and S.R.S Iyengar

# Identifying Influential Spreaders in a Social Network (While Preserving Privacy)

**Abstract:** In order to disseminate information in a social network, it is important to first identify the influential spreaders in the network. Using them as the seed spreaders, the aim is to ensure that the information is cascaded throughout the network. The traditional approach to identifying influential nodes is to determine the top-$r$ ranked nodes in accordance with various ranking methods such as PageRank, $k$-Shell decomposition, ClusterRank and VoteRank. In the current work, we study the problem of ranking the nodes when the underlying graph is distributedly held by a set of individuals, who consider their share of the data as private information. In particular, we design efficient secure multiparty computation (MPC) protocols for $k$-Shell decomposition, PageRank and VoteRank. For improved efficiency, we employ the oblivious RAM construct in conjunction with efficient data-oblivious graph data structures. We are the first to propose a secure variant of the VoteRank algorithm. We prove that the proposed protocols are asymptotically more efficient and have lower runtime in practice than the previous best known MPC protocols for computing $k$-Shell decomposition and PageRank centrality scores.

**Keywords:** privacy, social network analysis, secure multiparty computation

DOI 10.2478/popets-2020-0040
Received 2019-08-31; revised 2019-12-15; accepted 2019-12-16.

## 1 Introduction

Each year, millions of dollars are invested in analyzing the spread of information and diseases across a given population [1, 2]. Understanding the dissemination of information through word-of-mouth is crucial for ensuring effective advertising strategies [3]. Similarly, discerning the pattern associated with the spread of diseases can help mitigate them through targeted interventions [4]. Since the spread of rumors is known to have great potential for causing harm [5], governmental institutions have been keen on detecting as well as curbing their propagation. These are a few examples of scenarios where the broader issue of unraveling the patterns of a spreading phenomenon is addressed.

An approach often used as a solution to the above-stated problem is to model the interactions among the members of the population as a social network. Here, we denote each entity (individual/organization) as a node and the interaction between two entities by an edge between the corresponding nodes. This allows the use of network analysis techniques (Appendix-A) to investigate the phenomenon of spreading. A more refined problem in this direction is that of identifying a set of initial adopters in the network so as to have the extent of the spread maximized, popularly termed the *influence maximization problem* [6]. The influence of each node in the overall extent of the spread varies greatly [7] based on its structural position [8, 9]. In order to efficiently utilize resources, we would require to identify the influential spreaders in the underlying network. The classical approach has been to apply various ranking techniques to determine the importance of each node. Having obtained the rank of all the nodes, we consider the top ranked nodes as the candidates for initiating the spread [3]. The popular methods used to rank nodes based on their influence include PageRank [10], $k$-Shell decomposition [11], VoteRank [3], TwitterRank [12], etc. Recently, a few heuristic algorithms have also been proposed [13, 14].

Most of the algorithms designed to rank nodes assume that the entire graph representing the social network is available as input. These algorithms discount on the fact that the graph may not always be known in its entirety. More often than not, the network information is known in bits and pieces by several data holders, such that each one of them only has a partial view of the network. One of the major reasons for the data to be present in such a *distributed setting* is due to the sensitive nature of it. Consider, as an example, the case of a sexual relationship network. The informa-

***Corresponding Author: Varsha Bhat Kukkala:** Indian Institute of Technology Ropar, E-mail: varsha.bhat@iitrpr.ac.in
**S.R.S Iyengar:** Indian Institute of Technology Ropar, E-mail: sudarshan@iitrpr.ac.in

tion of who has had a sexual relationship with whom is considered to be private information and hence each is aware of only his/her interactions. Thus, the aggregate structure of the sexual network of a given population is never known. However, in order to take preventative measures to deter the spread of sexually transmitted diseases, an investigation of the hidden sexual network is essential, as is identifying the influential nodes in the network. This theme of identifying influential spreaders in a network containing sensitive data distributedly held is recurrent in several networks such as supply chain networks [15–17], informal networks in organizations [18], financial transaction network [19] and terrorist/criminal networks [20]. Clearly, in such scenarios, algorithms designed to run in the centralized setting cannot be directly used to compute the rank of nodes, without having to compromise the privacy of the involved entities. Not only do the traditional algorithms expect that the network be known completely, the run of these algorithms also varies based on the input graph. The control flow and the memory access pattern during the run of the algorithm are dependent on the input network structure. That is, the algorithms are not designed to be *data-oblivious*. This poses an additional threat to the privacy of the concerned individuals, since the inconsistency in the run of the algorithm could in itself leak sensitive information regarding the underlying network.

The current objective is to determine influential spreaders in a network, where the data is held in a distributed setting. Furthermore, it is required to do so while preserving the privacy of the involved individuals by guaranteeing that no information apart from the set of influential spreaders is revealed. We model the above problem as an instance of *secure multiparty computation* (MPC), where multiple parties wish to collectively compute a publicly known function of their private inputs. The computation is done in such a way that each party learns nothing apart from the desired output. In our case, we assume that there are $n$ data holders, denoted $D_1, D_2 \dots D_n$, who collectively hold the graph $G(V, E)$ representing the underlying social network. We assume that the set of nodes $V$ in the graph is known to all of them. However, the set of edges representing private interactions is known distributedly. That is, each data holder $D_i$ holds a partial view $G_i(V, E_i)$ of the graph where $E_i \subset E$, such that $G(V, E) = \bigcup_{i=1}^{n} G_i(V, E_i) = G(V, \bigcup_{i=1}^{n} E_i)$. In the current work, we rely on ranking the nodes by using secure variants of the $k$-Shell decomposition, PageRank and the VoteRank algorithms.

## 1.1 Contributions

The current work largely builds on the tools and techniques of secure computation as well as those of social network analysis. We aim to bridge the gap in the use of cryptographic tools for performing social network analysis. The contributions of the current work can be summarized as follows:

- We identify the need to determine influential spreaders in distributedly held networks consisting of sensitive information. In particular, we design MPC protocols for evaluating $k$-Shell decomposition (Section 4) , PageRank (Section 5) and VoteRank (Section 6) algorithms.
- In order to facilitate an efficient solution, we use an alternative graph representation known as the *edgelist* representation [21]. We provide a detailed description of how to securely construct the edgelist.
- We consider a two-party scenario and design protocols that are secure in the semi-honest setting. When the network data is distributed among more than two data holders, we reduce the problem to two representative parties securely evaluating the protocols on behalf of the multiple data holders. Further details of how the data is distributed among the two parties is described in Section 3.
- In order to guarantee the security of the designed protocols, we prove their correctness. We also prove that the designed protocols are data-oblivious and thereby show that the run of the protocol is independent of the input, except on the size of the input graph (i.e., the size of the edgelist).
- We evaluate the performance of the designed protocols by implementing them on a secure framework called Obliv-C [22]. We benchmark the performance of the protocols for varying graph sizes. We individually account for the cost of running the protocol for the edgelist construction and that of the actual network measure protocol.
- In the case of $k$-Shell decomposition and PageRank algorithms, we draw comparisons of the performance of the current protocols with those proposed previously in the literature. We show that the current protocols outperform the ones previously designed and are hence more practical.

# 2 Background

## 2.1 Secure Multiparty Computation

Introduced by Yao [23], secure multiparty computation is a branch of cryptography which deals with facilitating two or more parties to *securely* evaluate a function, involving private data items supplied as input by each of them. The notion of security here must guarantee that each of the involved parties learn nothing other than the output of the function as well as ensure the correctness of the computed result. The process of evaluating the function in itself must not leak any additional information other than the output. However, it is important to note that anything that a party can infer from the output itself cannot be avoided. MPC protocols have been designed as a solution for a variety of problems including secure election [24], auction [25], genomics [26], etc. The literature reports several generic protocols that allow the secure evaluation of any computable function, described either as an arithmetic circuit or as a boolean circuit, under various settings [27–30]. Although the protocols designed in the current work can be evaluated using any of the generic approaches, we rely on the Yao's garbled circuit approach [30] to implement them.

## 2.2 Garbled Circuits

One of the most preferred generic approaches to securely evaluate a function described as a Boolean circuit is the garbled circuit technique, credited to Yao [30]. It is designed to securely compute a function $f$ under the 2-party setting. One of the parties known as the *generator* encrypts the boolean circuit equivalent to $f$ by associating each input $i$ of the circuit with a pair of keys. The key pair, denoted by $k_i^0$ and $k_i^1$, are representative of the possible input bits, 0 and 1 correspondingly. Using these keys, the generator creates encrypted tables for the possible output across all the gates in the circuit in such a way that the correct output can be obtained only when one has keys corresponding to the actual inputs. This garbled circuit is passed onto the party termed the *evaluator*, who would require the keys corresponding to each of the inputs to evaluate the circuit. The keys corresponding to the inputs associated with the generator are directly passed on to the evaluator. The parties use an *oblivious transfer (OT)* protocol for the evaluator to obtain the keys associated with his inputs. An OT protocol allows the evaluator to learn precisely one of the keys $k_j^0$ or $k_j^1$, without the generator knowing which of the two keys were transferred. The output of the circuit is then generated and collectively decoded.

## 2.3 Oblivious RAM

Consider a scenario where a client stores an array of length $N$ on an untrusted server, such that each entry of the array is protected from the server. The client wishes to access the array in such a way that the server remains oblivious to the access pattern and thus any sensitive data is prevented from being leaked to the server. Goldreich and Ostrovsky were the first to propose the cryptographic primitive of *Oblivious* RAM to address the above problem [31]. An ORAM scheme allows the server to maintain a memory structure whose entries can be manipulated obliviously by the client using essentially two protocols. The *initialization* protocol allows the client to set up the memory structure on the server by initializing the entries of the array. Once initialized, the *access* protocol translates a given logical address of the array to a sequence of physical addresses to the memory structure, using which an element of interest can be retrieved. The security guarantees of the ORAM primitive ensures that the access pattern during neither the initialization nor the access protocol leaks any sensitive data to the server. That is, while initializing data elements of two different arrays of the same size, the physical addresses revealed to the server in both the cases are indistinguishable. Similar is the case for the access protocol initiated on two different logical addresses. Since its inception, several ORAM schemes have been developed [32–35], each aiming to optimize different performance metrics such as bandwidth, client storage, server memory overhead, etc. Apart from the traditional client server setting, the applicability of ORAMs to other scenarios such as secure processors [36, 37], outsourced storage [33, 38] and multiparty computation [39–41] have been considered.

## 2.4 RAM Secure Computation

In the generic circuit based approaches to MPC, oblivious access to a specific private memory location involves reading all the entries of the memory. Thus, an $\mathcal{O}(1)$ memory access in the RAM model essentially requires $\mathcal{O}(N)$ memory accesses in the circuit model, for a memory of size $N$. This reveals the inefficiency of the generic

circuit model approach to MPC. The RAM model of secure computation (RAM-SC) counters this by utilizing the ORAM primitive in conjunction with the circuit based approach to facilitate oblivious memory accesses in sub-linear time. As in the case of generic protocols, all the operations apart from memory accesses are evaluated securely using the circuit based approach even in RAM-SC. Each read/write to a memory location is performed by circuits that emulate ORAM operations. Traditional ORAM schemes are designed to provide single-sided privacy guarantees, that is they ensure only client privacy from the server. However, in an MPC setting, privacy guarantees must extend to all the involved parties. Thus, all the client side computation is performed using circuit based secure computation protocols. That is, we construct circuits on the fly that take the secret logical address as input and output the translated sequence of physical addresses. The security guarantees of the ORAM scheme in itself allows for the physical locations to be revealed to the parties. The protected data items retrieved from these locations are further processed by circuits which return encoded output that can be used to update the contents of the accessed physical locations. This way, the memory accesses are also made oblivious. Gordon et al. [40] was the first to describe how any ORAM scheme in the client server model can be adapted for use with any generic two-party secure computation protocol. Following this, several works in the literature have considered improving ORAMs for the specific case of secure computation [39, 41–44]. The Circuit ORAM scheme [43] is known to be the most efficient for use in MPC with respect to its asymptotic complexity. However, the recent contribution by Zahur et al. [44] shows the Square-Root ORAM scheme to be better than Circuit ORAM for practical data sizes. Furthermore, the Square-Root ORAM scheme offers a more efficient approach to initializing the oblivious memory structure. Hence, in the current work, we rely on the Square-Root ORAM scheme for implementing the designed protocols and refer the reader to [44] for further details.

## 2.5 Security Model

While designing MPC protocols, we must define the computational capabilities of the parties and the extent of corruption, under which the security guarantees hold. In the current work, we design protocols under the 2-party setting that are guaranteed to be secure in the *semi-honest* model. That is, we assume that the two parties neither collude nor do they deviate from the protocol. Most of the sensitive social interactions currently investigated use trusted third parties, which requires that the data holders disclose their private information completely to a third party. For example, an external agency such as Keyhubs [45] is hired to perform network analytics based on the informal interactions between the employees of the consulting organization. In comparison to this, considering a semi-honest threat model is clearly advantageous since the disclosure of private data can be completely avoided. Additionally, the semi-honest model fits well with scenarios where the parties are inherently known to be honest, as in the case of governmental institutions investigating criminal networks [20], and are bound by statutory laws from disclosing their data to one another. Since the private data considered is that of a social network, we must also cater to the possibility that the underlying network is collectively known among multiple data holders. In such a case, each of them secret shares (e.g. secure XOR shares) their view of the network among two representative parties, who now hold the network data distributedly. The two representative parties, here on referred to as $P_1$ and $P_2$, run the designed 2-party protocols on behalf of all the data holders to compute the desired network measure. Further details of how the network data is distributed and stored is described in Section 3. Determining the ideal candidates for the role of parties $P_1$ and $P_2$ is application scenario specific. Hence, further details of the same lie beyond the scope of the current work. Additionally, the stronger threat model of a *malicious* setting where the parties are capable of colluding and deviating from the protocol is not addressed here.

## 2.6 Notation

In order to increase the readability of the protocols as well as to keep them compact, we follow a few conventions. First is to note the distinction between the two types of variables used. Some variables store publicly known data values while the others store sensitive and private data. In order to distinguish between them, only those with private data are specified within angular brackets ($\langle\ \rangle$). It is indicative of the fact that the value of the variable is distributedly held (e.g. secret shared) among parties $P_1$ and $P_2$. An array can be stored in the ORAM when we want to access its entries in an oblivious manner without revealing the array indexes. The ORAM protocols discussed previously are used in this case. We distinguish the read write

operations of an ORAM scheme through separate subroutines of OramRead() and OramWrite(). The routine OramRead() takes two input parameters, the first is the name of the ORAM array, while the second is a private variable that stores the logical address in the memory that needs to be read. The value read at the input location is returned as a private value. On the other hand, the OramWrite() routine takes three input parameters and has no return value. The first parameter denotes the name of the ORAM array, the second and third are private variables that store the logical address of the memory location and the new content to be written correspondingly. Another aspect of the designed protocols is the use of oblivious *if-else* construct, whose functionality differs from a regular *if-else* construct. Typically, the construct allows for selectively executing either the statements within the body of *if* or those within *else*, based on the truth/falsity of the associated conditional statement. This leaks sensitive information if the construct is conditioned on private variables. Hence, to avoid this, an oblivious *if-else* construct executes statements within both the branches in such a way that the effect of only one of the branches is reflected [22]. In the designed protocols, an oblivious *if-else* construct is denoted within angular brackets (i.e. $\langle if \rangle$ and $\langle else \rangle$).

In general, we consider the data items to consist of integer values. However, in some cases such as in the PageRank protocol, we might have to deal with fractional values as well. In order to distinguish these, variables/arrays storing floating point values are appended with a tilde over their labels (e.g.. $\widetilde{X}$ where $X$ can be either a variable or an array name). We use the fixed point notation to deal with real valued data items as it is easy and efficient to implement. Each real value is mapped to an integer by scaling and then rounding off. That is, for a real value $r$, the integer equivalent would be $int(r) = \lceil (r \times S) \rceil$, where $S$ is the scaling factor. Performing arithmetic operations on the real values followed by obtaining the integer equivalent of the result can be directly adapted to performing the operations on the integer representations itself. The real valued end result is retrieved by mapping the obtained integer result to its equivalent real value.

# 3 Graph Representation

An important factor contributing to the efficiency of the designed protocols is the manner in which the input graph is stored and dealt with. Most of the previous works that deal with securely computing over network data prefer to store the underlying graph using the *adjacency matrix* representation [46–48]. When each entry of the matrix is protected (i.e. through secret sharing or encryption), this representation only reveals the number of nodes ($|V|$) in the graph. This representation is well suited for data-oblivious settings and its space complexity is $\mathcal{O}(|V|^2)$. However, considering real world networks that are known to be very sparse [49], where the number of edges is of the order of number of nodes, the adjacency matrix representation is space inefficient. As a space efficient alternative for real world graphs, we utilize the *edgelist* representation [21]. The edgelist stores the input graph $G(V, E)$ as a 2-tuple $(\mathcal{E}, Idx)$, where $\mathcal{E}$ is an ordered list of edges while $Idx$ stores a list of indexes to particular entries of $\mathcal{E}$. Each node is assigned a label starting from 0 to $|V| - 1$. The list $\mathcal{E}$ stores all the neighbors of node 0, followed by the neighbors of node $1, 2, 3 \ldots (|V| - 1)$, in the same order. The last entry of $\mathcal{E}$ is $\{-1\}$ signifying a null entry and is used for the purposes of implementation. Thus, the space complexity of $\mathcal{E}$ is $|E| + 1$. In order to delimit the neighbors of one node from the other, the $i^{th}$ entry of list $Idx$ stores the first index of $\mathcal{E}$ from where the enumeration of the neighbors of node $i$ begins. The last entry of $Idx$ points to the last entry of $\mathcal{E}$ and the space complexity of it is $|V| + 1$. Thus, the overall space complexity of the edgelist $(\mathcal{E}, Idx)$ is precisely $|V| + |E| + 2$, which in the case of real world graphs is of $\mathcal{O}(|V|)$. Unlike the adjacency matrix representation, the edgelist reveals not only the number of nodes but also the total number of edges, by virtue of the length of lists $Idx$ and $\mathcal{E}$ respectively.

## 3.1 Edgelist Construction

We first consider the scenario where the network data is collectively known by two data holders and thus they themselves are the computing parties $P_1$ and $P_2$. Each of the parties $P_1$, $P_2$ holds a subset of edges $E_1, E_2$ correspondingly as their private data, such that the set of all edges $E = E_1 \cup E_2$. For each edge $(u, v) \in E_i$, party $P_i$ creates shares of $(\langle u \rangle, \langle v \rangle)$ and distributes a share each to the other party. Thus, the parties distributedly hold shares of the set of all edges, which they obliviously sort [50–54] based on the first entry $\langle u \rangle$ of each tuple $(\langle u \rangle, \langle v \rangle)$. After obtaining the sorted list of shares, only the second entry $\langle v \rangle$ of each tuple is retained, which in turn constitutes entry wise shares of the list $\mathcal{E}$. Appending the corresponding shares of the value $-1$ to this list concludes the construction of the list $\mathcal{E}$. Let the entries

of the list $Idx$ be denoted by $Idx[0], Idx[1], \ldots Idx[|V|]$. By the definition of $Idx$, we can conclude that the first entry of $Idx$ is always 0 and hence the parties $P_1$ and $P_2$ distribute shares of 0 among themselves. In general, $Idx[i] = Idx[i-1]+$ degree of node $i-1$. For each node $i$, let $d_{i1}$ represent the number of edges emanating from $i$ reported by party $P_1$ and $d_{i2}$ represent those reported by $P_2$. Degree of node $i$, denoted by $deg_i$, can be computed as $deg_i = d_{i1} + d_{i2}$. Each party $P_j$ distributes a share of $\langle d_{ij} \rangle$ to the other. Since all the distributed shares are additive in nature, each party obtains shares of $\langle deg_i \rangle$ by locally computing the addition of the shares $\langle d_{i1} \rangle + \langle d_{i2} \rangle$. This in turn allows the parties to compute shares of each entry $Idx[i]$. As in the case of $\mathcal{E}$, the parties $P_1$ and $P_2$ obtain the shares of the last entry $Idx[|V|]$ by distributing shares of the value $|E|$ among themselves. This concludes the construction of the edgelist, with parties $P_1$ and $P_2$ each having a share of its entries. These shares are further used to initialize the edgelist as ORAM arrays if required by the secure protocol for the corresponding network measure.

The above construction can be extended to the case where there are more than 2 data holders. The computing parties $P_1$ and $P_2$ can be picked from the set of data holders itself or the computation can be outsourced to two semi-honest third parties. The privacy of the data holders is guaranteed in either case since the computing parties are assumed to be semi-honest and provide the shares of the output to the required data holders. The changes required in the construction are as listed below. Firstly, it is required that the number of edges reported by each data holder is kept hidden. The data holders fix on a constant $k$ that represents an upper bound on the number of edges reported by them individually. An ideal approach to determine $k$ is to securely compute the maximum number of edges reported across all the data holders. Thus, each data holder distributes $k$ shares of data items representing their private edges $(\langle u \rangle, \langle v \rangle)$, padded by dummy entries symbolically denoted as $(\langle \infty \rangle, \langle \infty \rangle)$. As in the previous case, $P_1$ and $P_2$ obliviously sort the list of obtained shares. The largest value $k$ can attain is $|V|-1$, as a node could be neighbors with every other node in the graph. In such a case, the complexity of constructing the edgelist would boil down to the cost of performing oblivious sort on a list with $(|V|^2 - |V|)$ entries. Assuming a data-oblivious sorting technique such as the Batcher algorithm [51], this would amount to $\mathcal{O}(|V|^2 \log^2 |V|)$ operations. Despite having a high worst case time complexity, the cost of constructing the edgelist can be seen as a one-time process and is hence discounted when reporting the asymptotic complexity of the designed protocols. Unlike in the previous case, the data holders additionally perform secure addition to determine the total number of edges $|E|$ and the result is made public. This allows parties $P_1$ and $P_2$ to retain only the first $|E|$ entries of the sorted list as the shares of valid edges. The shares of the list $Idx$ can be constructed as described before. The only difference would be that all the data holders would be involved in performing secure addition to determine the degree of each node. We assume that the data holders are also semi-honest. Thus, it is expected that each of the data holders not only report a fixed number of edges (valid + dummy) but also provide correct shares of the number of valid edges reported.

# 4 $k$-Shell Decomposition

As an alternative to measuring cohesiveness of a network through its density, Seidman proposed the idea of *k-cores* that recursively decomposes the network into cohesive subgraphs known as cores [55]. The process fragments the network into a sequence of subgraphs such that each is a subgraph of the preceding graph and has higher cohesiveness. It was later observed that decomposing the graph into cores also helps in better identifying the most influential nodes in the network [11, 56]. The innermost core nodes are known to be influential spreaders of information, diseases, etc. The formal definition of $k$-cores and the algorithm to identify them is discussed further.

**Definition 1.** *Given a graph $G(V, E)$, consider the subgraph $G_k$ induced by the largest vertex set $V_k$, such that $V_k \subseteq V$ and $\forall v \in V_k$ the degree of $v$ in $G_k$ is at least $k$. Such an induced subgraph $G_k$ is called the k-core of graph $G$.*

We can thus construct a sequence of subgraphs satisfying the above definition, as shown below, where $G_i \leq G_j$ denotes that the graph $G_i$ is a subgraph of graph $G_j$:

$$G_k \leq G_{k-1} \leq \ldots G_2 \leq G_1 \leq G_0 = G(V, E)$$

This process of decomposing the graph into cores assigns each node a unique shell number.

**Definition 2.** *Given the $i$ and $i+1$ cores of a graph $G(V, E)$ as $G_i$ and $G_{i+1}$, for every vertex $v \in V$ we define shell$(v) = i$ if $v \in G_i - G_{i+1}$.*

Batagelj and Zaversnik were the first to propose an efficient technique known as the $k$-Shell decomposition algorithm [57] to determine the cores of a given network. The algorithm follows an iterative approach to prune few nodes in each iteration. All nodes with degree less than $i$ are pruned in the $i^{th}$ iteration and assigned shell value $i$. The process repeats by considering the graph that remains, i.e. the $(i+1)$-core. The pseudocode for the same is given in Figure 1. It has two nested *for* loops, in steps 2 and 4, of which the number of iterations of the inner *for* loop is dependent on the node currently set to be pruned in the outer loop. This shows that the algorithm is input graph dependent. We provide a data-oblivious equivalent in Figure 2 by coalescing the loops [58] such that, in each iteration we either access an element of the sorted vertex list or access a neighbor of the current node in list $\mathcal{E}$.

**Input:** Graph $G(V, E)$
**Output:** $shell(v)$, for each $v \in V$
1: $V \leftarrow$ sorted list of vertices based on their degree
2: **for** each $v \in V$ in the order **do**
3: $shell(v) \leftarrow degree(v)$
4: **for** each $u$ neighbor of $v$ **do**
5: **if** degree($u$) > degree($v$) **then**
6: $degree(u) \leftarrow degree(u) - 1$
7: Re-order the vertices in $V$ accordingly
8: **end if**
9: **end for**
10: **end for**

**Fig. 1.** Pseudocode for $k$-Shell Decomposition Algorithm

**Input:** Graph $(\mathcal{E}, Idx)$
**Output:** $deg(v)\ \forall v \in V$, denoting the shell value of node $v$
1: Initialize $deg$ and $pos$ each as an ORAM array of length $|V|$ with entries set to 0
2: Initialize $bin$ as an ORAM array of length $|V|-1$ with each entry set to 0
3: Initialize $vert$ as an ORAM array of length $|V|$ with every $i^{th}$ entry set as $i$
4: **for** $i = 0$ to $|V|-1$ **do**
5: $\langle temp1\rangle \leftarrow$ OramRead($Idx$,$\langle i+1\rangle$)
6: $\langle temp2\rangle \leftarrow$ OramRead($Idx$,$\langle i\rangle$)
7: $\langle temp3\rangle \leftarrow \langle temp1\rangle - \langle temp2\rangle$
8: OramWrite($deg$,$\langle i\rangle$,$\langle temp3\rangle$)
9: $\langle temp2\rangle \leftarrow$ OramRead($bin$,$\langle temp3\rangle$)+1
10: OramWrite($bin$,$\langle temp3\rangle$, $\langle temp2\rangle$)
11: **end for**
12: $Oblivious\_sort(vert, deg)$
13: $\langle start\rangle \leftarrow 0$
14: **for** $i = 0$ to $|V|-1$ **do**
15: $\langle temp1\rangle \leftarrow$ OramRead($bin$,$\langle i\rangle$)
16: OramWrite($bin$,$\langle i\rangle$, $\langle start\rangle$)
17: $\langle start\rangle \leftarrow \langle start\rangle + \langle temp1\rangle$
18: $\langle temp2\rangle \leftarrow$ OramRead($vert$,$\langle i\rangle$)
19: OramWrite($pos$,$\langle temp2\rangle$, $\langle i\rangle$)
20: **end for**
21: $\langle i\rangle \leftarrow 0$
22: $\langle v\rangle \leftarrow$ OramRead($vert$, $\langle i\rangle$)
23: $\langle j\rangle \leftarrow$ OramRead($Idx$, $\langle v\rangle$)
24: **for** $iter = 1$ to $|V| + |E| - 1$ **do**
25: $\langle$**if**$\rangle$ ($\langle j\rangle ==$ OramRead($Idx$, $\langle v+1\rangle$)) **then**
26: $\langle i\rangle \leftarrow \langle i\rangle + 1$
27: $\langle v\rangle \leftarrow$ OramRead($vert$, $\langle i\rangle$)
28: $\langle j\rangle \leftarrow$ OramRead($Idx$, $\langle v\rangle$)
29: $\langle$**else**$\rangle$
30: $\langle u\rangle \leftarrow$ OramRead($\mathcal{E}$, $\langle j\rangle$)
31: $\langle temp1\rangle \leftarrow$ OramRead($deg$, $\langle u\rangle$)
32: $\langle temp2\rangle \leftarrow$ OramRead($deg$, $\langle v\rangle$)
33: $\langle$**if**$\rangle$ $\langle temp1\rangle > \langle temp2\rangle$ **then**
34: $\langle du\rangle \leftarrow$ OramRead($deg$, $\langle u\rangle$)
35: $\langle pu\rangle \leftarrow$ OramRead($pos$, $\langle u\rangle$)
36: $\langle pw\rangle \leftarrow$ OramRead($bin$, $\langle du\rangle$)
37: $\langle w\rangle \leftarrow$ OramRead($vert$, $\langle pw\rangle$)
38: $\langle$**if**$\rangle$ ($\langle u\rangle ! = \langle w\rangle$) **then**
39: OramWrite($pos$, $\langle u\rangle$, $\langle pw\rangle$)
40: OramWrite($vert$, $\langle pu\rangle$, $\langle w\rangle$)
41: OramWrite($pos$, $\langle w\rangle$, $\langle pu\rangle$)
42: OramWrite($vert$, $\langle pw\rangle$, $\langle u\rangle$)
43: **end** $\langle$**if**$\rangle$
44: $\langle temp\rangle \leftarrow$ OramRead($bin$, $\langle du\rangle$)+1
45: OramWrite($bin$, $\langle du\rangle$, $\langle temp\rangle$)
46: $\langle temp\rangle \leftarrow$ OramRead($deg$, $\langle u\rangle$)$-1$
47: OramWrite($deg$, $\langle u\rangle$, $\langle temp\rangle$)
48: **end** $\langle$**if**$\rangle$
49: $\langle j\rangle \leftarrow \langle j\rangle + 1$
50: **end** $\langle$**if**$\rangle$
51: **end for**

**Fig. 2.** Protocol for computing $k$-Shell decomposition securely

## 4.1 Protocol Description

The protocol expects that the input graph in the edge-list representation is made available, with $\mathcal{E}$ and $Idx$ stored as individual ORAM arrays. We initialize other ORAM arrays namely *deg*, *vert*, *pos* and *bin* that are used further in the protocol. The array *deg* maintains the degree of all nodes. The array *vert* maintains the list of nodes in the sorted order of their degrees throughout the run of the protocol. As the position of a node in this array may be affected by the pruning of its neighbors, a separate array named *pos* is maintained to keep track of the position of each vertex in the sorted list *vert*. Additionally, we maintain certain indexes of the *vert* array that allows us to group nodes having the same degree. For this purpose, the array *bin* maintains a list of pointers such that $i^{th}$ entry of the array stores the index of the first node in *vert* having degree $i$. The protocol begins by computing the degree of each node and updates the *deg* array accordingly (steps 4-11). We simultaneously keep track of the number of nodes encountered of a particular degree which is temporarily stored in *bin*. We then sort the list of vertices in *vert* with respect to their degree (step 12) using Batcher sort [51]. In steps 14-20, the intended content of the arrays *pos* and *bin* are generated. The temporary information currently stored in the *bin* array is itself used to compute the index of the first node in *vert* having a particular degree. This ends the initialization phase of the protocol.

The pruning of nodes to securely determine the shell number of each node is performed in the *for*-loop between steps 24-51. In each iteration of the loop, we either determine the next node to be pruned by scanning through *vert* array or prune the current node by updating the degree of its neighbors. The private variable $\langle i\rangle$ stores the index of the current node being accessed from the *vert* array, while $\langle j\rangle$ is used to scan over the list of neighbors in array $\mathcal{E}$. The oblivious *if* branch at step 25 scans over *vert* array and the *else* branch between steps 29-50 scans over the array $\mathcal{E}$. The private variable $\langle v\rangle$ denotes the vertex that is currently being pruned, while its neighbor currently being processed is stored in $\langle u\rangle$.

In steps 33-48, we process the neighbor $u$ of node $v$ by reducing the degree of $u$ in case it is of a higher degree than $v$. We accordingly adjust the position of node $u$ to maintain the sorted ordering in *vert*. This is done by initially swapping node $u$ with node $w$, where $w$ denotes the first node in *vert* having degree equal to that of $u$. Further, the degree of $u$ is reduced and thereby included as a node in the previous bin. The corresponding index of the *bin* array is updated, and the changes to the positions of node $u$, $w$ are reflected in the *pos* array. The updated degree of each node at the end of the protocol itself denotes the shell value assigned to the node.

## 4.2 Analytical Evaluation

### 4.2.1 Complexity

From the pseudocode described in Figure 1, we can conclude that the complexity of performing $k$-Shell decomposition is $\mathcal{O}(|E|)$. The two nested *for*-loops contribute to a total of $|E|$ iterations. Further, assuming that the re-ordering of nodes in step 7 can be achieved using a constant number of swaps, the overall complexity of the algorithm remains $\mathcal{O}(|E|)$. Since the described algorithm is not inherently data-oblivious, a direct translation of the same into an MPC protocol using generic MPC constructs is not feasible. Considering the protocol in Figure 2, the major contribution to its complexity is by the *for*-loop in step 24. The instructions within the loop are iterated over $|V| + |E| - 1$ number of times, which is also an upper bound for the number of iterations of other loops in the protocol. Each iteration of the loop consists of performing a constant number of ORAM array accesses. Thus, the complexity of executing $|V| + |E|$ iterations of the loop is $\mathcal{O}((|V|+|E|)f(\mathcal{O}(|E|)))$ operations. Henceforth, $f(n)$ denotes the overhead of performing an ORAM access over an array of length $n$. Since the underlying graphs represent real world networks that are known to be sparse, the length of each of the ORAM arrays can be bounded by $\mathcal{O}(|E|)$. Assuming the complexity of performing data-oblivious sorting is $\mathcal{O}(n(\log(n))^2)$ over an $n$ length array [51], the overall complexity of the protocol is given by $\mathcal{O}((|V| + |E|)f(\mathcal{O}(|E|)) + (|V| \log^2 |V|))$. Although there are other alternatives with lower asymptotic complexity to oblivious sort an array [50, 52–54], we use the Batcher sorting technique throughout our work due to its efficiency in practice as well as its deterministic approach to sorting.

### 4.2.2 Security

Since the protocols are designed in the semi-honest adversarial model, proving that the protocol is secure entails to guaranteeing its correctness and privacy. Correctness of a protocol ensures that the output generated by the protocol is the same as the intended result. The correctness of the protocol in Figure 2 follows from the instruction sequence directly. It can be ascertained by the protocol description provided in Section 4.1. A protocol is said to preserve privacy if it ensures that no information other than the desired output is revealed. Since each of the designed protocols performs computation on protected data and no intermediary values are released apart from the desired output, the security of the protocol boils down to what is leaked by the access pattern during the run of the protocol in itself. This is the reason that we focus on proving the data-obliviousness of the protocols to establish that the access pattern of the protocol is independent of the underlying graph structure and hence preserves privacy. Translating a data-oblivious algorithm into its equivalent MPC protocol is feasible when the primitive operations assumed in the algorithm can be performed securely using MPC constructs. The primitive operations assumed in our case are limited to arithmetic operations of addition, subtraction, multiplication and division. Since these operations can be securely performed using the generic construct of garbled circuits, we prove the data-obliviousness of the designed protocols as follows. All the arrays - *deg*, *bin*, *vert*, *pos*, $\mathcal{E}$, *Idx* are stored in the ORAM. Hence, the access pattern of these arrays do not leak any information about the elements being accessed. Throughout the protocol, apart from accessing elements of the arrays in the ORAM, we perform addition and subtraction operations that are assumed as primitive operations and hence are oblivious. In step 10, we sort the array *vert*, which can be performed data-obliviously. All the *for* loops in the protocol are executed a publicly known number of times. As described in Section 2, the *if-else* construct can be realized data-obliviously. Hence the sequence of operations remain the same for a fixed-size input and therefore the overall protocol remains data-oblivious. Moreover, none of the designed protocols modifies these underlying primitives in any way that has an effect on their security properties, and hence we claim that the privacy guarantees of the designed protocols also hold.

# 5 PageRank

The World Wide Web can be visualized as a network, where web pages are denoted by nodes and the hyperlinks between them are represented by edges. Page et al. [10] proposed an algorithm to rank these web pages based on the underlying hyperlink structure, popularly known as the PageRank algorithm. The algorithm associates a centrality value to each node that quantifies its importance. The PageRank algorithm recursively defines the centrality of a node to be proportional to the sum of the centralities of those nodes having incoming links to the considered node. Since its inception, it has been used to find influential nodes in networks including citation networks and social networks [59, 60]. We adopt the *iterative method* to compute PageRank centrality. The method is briefly described here. All the nodes begin with an equal share of the PageRank centrality value. Every vertex distributes a predetermined fraction of its PageRank value equally among all of its neighbors (i.e., through the outgoing links). In the case of no outgoing links, the node retains this fraction of the PageRank with itself. The remaining fraction of its PageRank value is uniformly redistributed to all nodes in the network. This constitutes an iteration of the PageRank algorithm. Therefore, in each iteration, the PageRank value of a node is computed as the sum total of all the shares it receives through its in-links and those received through the uniform redistribution. The uniform redistribution of the PageRank values among all nodes is performed to ensure that nodes having zero outgoing links do not end up accumulating the overall centrality measure. It is well established that the centrality values will converge as the number of iterations increase [10]. In order to simplify the process of computing the PageRank centrality, we define the notion of an *update matrix* and an *update rule*. Let $G(V, E)$ represent the graph of interest. The out-degree of a node $u \in V$ is defined as the number of its outgoing links, denoted as $od(u)$.

**Definition 3.** *The update matrix for a graph $G(V, E)$ is denoted by $N = [N_{ij}]$ with size $(|V| \times |V|)$. The entries of the matrix are defined as follows:*

$$N_{ij} = \begin{cases} \frac{s}{od(i)} + \frac{(1-s)}{|V|} & \text{if } od(i) \neq 0 \text{ and } \{i,j\} \in E \\ s + \frac{(1-s)}{|V|} & \text{if } od(i) = 0 \text{ and } i == j \\ \frac{(1-s)}{|V|} & \text{otherwise} \end{cases}$$

The entry $N_{ij}$ of the update matrix denotes the fraction of the PageRank value that node $i$ contributes to the overall centrality of node $j$. The definition includes a *redistribution parameter* $s$ that determines the fraction of PageRank to be reserved for outgoing links. Thus, the remaining $(1 - s)$ fraction of each node's PageRank value is uniformly distributed among all nodes in the network.

**Definition 4.** *Let $r_i^{(k)}$ represent the PageRank value of node $i$ after $k$ iterations. The PageRank update rule is defined as follows:*

$$r_i^{(k)} \leftarrow \sum_{j=1}^{n} N_{ji} r_j^{(k-1)}$$

The repeated application of the PageRank update rule allows us to iteratively compute the PageRank centralities of all the nodes in the network. To support a space efficient graph representation, an algorithm to compute PageRank for the case when the input graph is in the adjacency list format is provided in Figure 3. There are several aspects of this algorithm that render it non-oblivious. The branching of the *if-else* construct in steps 6-12 is dependent on the input graph structure. Further, the number of times the *for*-loop in step 7 is executed depends on the length of the adjacency list of the current node. The entries of the PageRank vectors $r^{(k)}$ here are accessed in an input-dependent manner. The advantage, however, is that there is no need to maintain the matrix $N$ explicitly. An equivalent oblivious algorithm to compute PageRank using the adjacency matrix representation is described in Appendix B. We design an MPC protocol to compute the PageRank values of nodes, by incorporating modifications to the algorithm in Figure 3. Our protocol, as described in Figure 4, assumes the edgelist representation for the input graph.

## 5.1 Protocol Description

The protocol in Figure 4 takes three input parameters, consisting of the edgelist representation $(\mathcal{E}, Idx)$ of the underlying graph, the number of iterations denoted by $l$ and the PageRank redistribution parameter $s$. The lists $\mathcal{E}$ and $Idx$ are provided as individual ORAM arrays. The protocol begins by initializing three ORAM arrays $od$, $\widetilde{r}_{old}$ and $\widetilde{r}_{new}$, each of length $|V|$. The $i^{th}$ entry of $od$ array serves to store the out-degree of node $i$. Since updating the PageRank values of nodes in the current iteration requires knowledge of their previous PageRank values, we maintain two different ORAM arrays, $\widetilde{r}_{old}$ and $\widetilde{r}_{new}$, to store the PageRank values of both the it-

**Input:** Graph $G(V, E)$, $l$, $s$
**Output:** The PageRank values of all the nodes

1: Initialize $r_i^{(0)} \leftarrow 1/|V|$ for $1 \leq i \leq |V|$ (all other $r^{(k)}$ vectors are initialized as zero vectors for $k \geq 1$)
2: **for** $k = 1$ to $l$ **do**
3: $pool \leftarrow 0$
4: **for** $i = 0$ to $|V| - 1$ **do**
5: $od_i \leftarrow$ out degree of node $i$
6: **if** $od_i \neq 0$ **then**
7: **for** each $j$ neighbor of $i$ **do**
8: $r_j^{(k)} \leftarrow r_j^{(k)} + \{s/od_i \times r_i^{(k-1)}\}$
9: **end for**
10: **else**
11: $r_i^{(k)} \leftarrow r_i^{(k)} + \{s \times r_i^{(k-1)}\}$
12: **end if**
13: $pool \leftarrow pool + \{(1-s) \times r_i^{(k-1)}\}$
14: **end for**
15: **for** $i = 0$ to $|V| - 1$ **do**
16: $r_i^{(k)} \leftarrow r_i^{(k)} + \{pool/|V|\}$
17: **end for**
18: **end for**

**Fig. 3.** Pseudocode for iterative PageRank Algorithm using adjacency list

erations. To begin with, all the nodes have the same centrality value $1/|V|$ as initialized in the array $\widetilde{r}_{old}$. The *for*-loop in steps 3-8 determines the out-degree of each of the nodes using the entries of the $Idx$ array, and stores the result in the array $od$. The *for*-loop in steps 9-42 is the heart of the protocol that computes the PageRank values through the repeated application of the PageRank update rule (Definition 4). Each iteration of the *for*-loop requires that a few private variables be maintained. The private variable $\langle cur\_node \rangle$ maintains the current node whose neighbors are being explored, while $\langle cur\_vote \rangle$ stores its PageRank value available for distribution through the outgoing links. The variable $\langle cur\_deg \rangle$ stores the number of neighbors of $\langle cur\_node \rangle$. The variable $\langle cur\_indx \rangle$ stores the index of the current edge in $\mathcal{E}$ that is to be processed. Lastly, the variable $\langle pool \rangle$ accumulates the fraction of PageRank value contributed by all the nodes intended to be redistributed uniformly across all of them. The private variables $\langle temp \rangle$, $\langle temp1 \rangle$, $\langle temp2 \rangle$ and $\langle temp3 \rangle$ are just a few temporary variables used for increasing the readability of the protocol. The *for*-loop between steps 15-35 either updates the next node to be explored or scans over the list of edges of the current node stored in the array $\mathcal{E}$, one after the other, while simultaneously updating the PageRank value of the nodes accordingly. The oblivious *if*-construct in step 17 determines whether all the neighbors of the current node have been explored. This in turn would imply that the current node must be updated with the next node and update the other private variables accordingly (steps 25-27). Prior to updating the current node, the $\langle pool \rangle$ variable accounts for its contribution of PageRank value intended for uniform distribution across all nodes (step 24). Additionally, the special case when the current node has no neighbors is

**Input:** Graph $(\mathcal{E}, Idx)$, $l$, $s$
**Output:** The PageRank values of all the nodes

1: Initialize $od$ as an $|V|$ length ORAM array
2: Initialize $\widetilde{r}_{old}$ and $\widetilde{r}_{new}$ as $|V|$ length ORAM arrays with each entry set as $\frac{1}{|V|}$ and 0 respectively.
3: **for** $i = 0$ to $|V| - 1$ **do**
4: $\langle temp1 \rangle \leftarrow$ OramRead$(Idx, \langle i \rangle)$
5: $\langle temp2 \rangle \leftarrow$ OramRead$(Idx, \langle i + 1 \rangle)$
6: $\langle temp3 \rangle \leftarrow \langle temp2 \rangle - \langle temp1 \rangle$
7: OramWrite$(od, \langle i \rangle, \langle temp3 \rangle)$
8: **end for**
9: **for** $k = 1$ to $l$ **do**
10: $\langle cur\_node \rangle \leftarrow 0$
11: $\langle cur\_vote \rangle \leftarrow$ OramRead$(\widetilde{r}_{old}, \langle cur_node \rangle)$
12: $\langle cur\_deg \rangle \leftarrow$ OramRead$(od, \langle cur_node \rangle)$
13: $\langle cur\_indx \rangle \leftarrow 0$
14: $\langle pool \rangle \leftarrow 0$
15: **for** $i = 0$ to $|E| + |V| - 1$ **do**
16: $\langle temp \rangle \leftarrow$ OramRead$(Idx, \langle cur_node + 1 \rangle)$
17: **if** $\{\langle cur\_indx \rangle == \langle temp \rangle\}$ **then**
18: **if** $\langle cur\_deg \rangle == 0$ **then**
19: $\langle temp1 \rangle \leftarrow$ OramRead$(\widetilde{r}_{new}, \langle cur_node \rangle)$
20: $\langle temp2 \rangle \leftarrow \langle s \rangle \times \langle cur\_vote \rangle$
21: $\langle temp3 \rangle \leftarrow \langle temp1 \rangle + \langle temp2 \rangle$
22: OramWrite$(\widetilde{r}_{new}, \langle cur_node \rangle, \langle temp3 \rangle)$
23: **end if**
24: $\langle pool \rangle \leftarrow \langle pool \rangle + (\langle 1 - s \rangle \times \langle cur\_vote \rangle)$
25: $\langle cur\_node \rangle \leftarrow \langle cur\_node \rangle + 1$
26: $\langle cur_vote \rangle \leftarrow$OramRead$(\widetilde{r}_{old}, \langle cur_node \rangle)$
27: $\langle cur\_deg \rangle \leftarrow$ OramRead$(od, \langle cur_node \rangle)$
28: **else**
29: $\langle nbr \rangle \leftarrow$ OramRead$(\mathcal{E}, \langle cur_indx \rangle)$
30: $\langle temp1 \rangle \leftarrow$ OramRead$(\widetilde{r}_{new}, \langle nbr \rangle)$
31: $\langle temp2 \rangle \leftarrow \langle temp1 \rangle + \left(\frac{\langle s \rangle \times \langle cur\_vote \rangle}{\langle cur\_deg \rangle}\right)$
32: OramWrite$(\widetilde{r}_{new}, \langle nbr \rangle, \langle temp2 \rangle)$
33: $\langle cur\_indx \rangle \leftarrow \langle cur\_indx \rangle + 1$
34: **end if**
35: **end for**
36: **for** $i = 0$ to $|V| - 1$ **do**
37: $\langle temp1 \rangle \leftarrow$ OramRead$(\widetilde{r}_{new}, \langle i \rangle)$
38: $\langle temp2 \rangle \leftarrow \langle temp1 \rangle + \frac{\langle pool \rangle}{|V|}$
39: OramWrite$(\widetilde{r}_{old}, \langle i \rangle, \langle temp2 \rangle)$
40: OramWrite$(\widetilde{r}_{new}, \langle i \rangle, \langle 0 \rangle)$
41: **end for**
42: **end for**

**Fig. 4.** Protocol for computing PageRank values of nodes securely

handled (steps 18-23) by ensuring the node retains a fraction of its PageRank value with itself. The steps 29-33 handles the case where the list of neighbors of the current node are yet to be fully explored. The private variable $\langle nbr \rangle$ stores the neighbor of the current node, as read from the array $\mathcal{E}$. We update the PageRank value of node $nbr$ by including the current node's contribution to it. Further, we update the variable $\langle cur\_indx \rangle$ to point to the next edge to be processed. At the end of the *for*-loop in step 35, all the nodes in the array $\mathcal{E}$ are processed and their PageRank value received through the incoming links are accounted for. Thus, the *for*-loop in 36-41 accounts for updating the PageRank values of nodes by uniformly distributing the centrality value accumulated in the *pool* variable. The protocol concludes by repeating the above process a fixed number of times, as determined by the input parameter $l$.

## 5.2 Analytical Evaluation

### 5.2.1 Complexity

The iterative approach using the adjacency list representation (refer Figure 3) has the overall complexity of $\mathcal{O}(l(|E| + |V|))$. In each of the $l$ iterations, we require $\mathcal{O}(|E|)$ operations to scan through the list of neighbors of all the nodes and $\mathcal{O}(|V|)$ operations to distribute the PageRank value aggregated in the *pool* The non-oblivious nature of the above algorithm prevents it from being directly implemented using generic MPC constructs. Since our protocol for computing PageRank centrality (refer Figure 4) is designed to take advantage of the sparsity of real world networks (i.e. $|E| = \mathcal{O}(|V|)$), we can generalize that the ORAM arrays used in the protocol are of length $\mathcal{O}(|E|)$. The protocol essentially involves computing the PageRank values through scanning the edgelist followed by finalizing the PageRank value of each node by accounting for the uniform redistribution. This is performed over $\mathcal{O}(|E| + |V|)$ iterations of the for-loop in steps 15-35. Given that the above process is repeated $l$ times and that a constant number of ORAM accesses are made in each iteration, the overall complexity of the protocol can be given by $\mathcal{O}(l(|E| + |V|)f(\mathcal{O}(|E|)))$. Owing to the sparsity of the underlying graph, the complexity can be further reduced to $\mathcal{O}(l|V|f(\mathcal{O}(|V|)))$.

### 5.2.2 Security

The protocol instructions, as described in Section 5.1, account for all the three cases of PageRank value distribution - to the neighboring nodes through the outgoing links, the uniform redistribution to all nodes in the graph and the special case of retaining the PageRank value when the node has no outgoing links. Thus, the output generated by the protocol is the same as the pseudocode described in Figure 3. All of the sensitive data are stored in private variables/arrays, which are distributedly held, and is ensured that their values are not disclosed in the open. Furthermore, to prevent the leakage of information through memory accesses, those arrays with input-dependent memory accesses are stored as ORAM arrays. The protocol replaces the input-dependent loops of the algorithm with equivalent loops having a fixed number of iterations. The two nested *for* loops in steps 4-14 of the pseudocode for PageRank (Figure 3) are now coalesced into a single *for* loop as given in step 15 of the protocol. Although this results in an increase in the total number of iterations from $|E|$ to $|E| + |V| - 1$, the additional $|V| - 1$ iterations are required to obliviously update the next node to be processed. However, it is important to note that the overall functionality of the nested loops is preserved. The input-dependent branch statements are also replaced by oblivious *if-else* constructs. The above mentioned reasons justify the data-obliviousness of the protocol, which in turn guarantees privacy.

# 6 VoteRank

There exists a possibility that the sphere of influence of the top-$k$ influential nodes picked from a given network overlaps. In order to overcome this undesirable effect, Zhang et al. [3] proposed a simple yet effective method to identify most influential nodes in the network. Similar to PageRank, the authors propose an iterative approach in which the most influential nodes are elected one by one, while considering only local information.

**Definition 5.** *Voting score of a node $u \in V$, denoted by $s_u$, is the total votes obtained by $u$ through its neighbors. Voting ability of $u$ on the other hand, denoted by $va_u$, is the vote that $u$ contributes to each of its neighbors. Thus, given the input graph $G(V, E)$, each node $u \in V$ is associated with a tuple $(s_u, va_u)$.*

The pseudocode for computing VoteRank and determining the top $r$ influential nodes is as described in Figure 5. It begins by initializing the tuple $(s_u, va_u)$ for each node $u$ as $(0, 1)$. In every iteration (Steps 2-22), we compute the score of each node by aggregating the votes provided by its neighbors, as given in Step 3-7. The node with the highest votes (i.e. node $v_{max}$) is picked as the most influential node in Steps 8-13. From here on, its voting ability as well as its score is set to zero (Step 14). In Steps 15-21, the scores and the voting ability of nodes are updated. The voting ability of only those nodes which have $v_{max}$ as one of its neighbors is reduced by a predetermined factor $fact$, restricting other nodes in the neighborhood from being elected. Further, to disallow the participation of $v_{max}$ in the up-coming iterations, it is deleted from the list of neighbors of nodes in step 18. In step 20, the scores of all the nodes are reset to 0 in order to ensure a fresh start to voting in the next iteration. This entire process is repeated $r$ times From the pseudocode described above, we can conclude the following regarding the data-oblivious aspects of it. The *for*-loop

in step 4 is not data-oblivious as the number of times it is executed depends on the number of neighbors that the current node being explored has. Determining the node with maximum score is part of the desired output, however, the evaluation of the conditional in step 11 can reveal the relative ordering of the scores among some of the nodes in the network. Such conclusions can be drawn each time the $v_{max}$ is updated. Similarly, the *if* conditional in step 16 is also input-dependent, possibly revealing the network structure. In order to counter the above mentioned issues, we design a secure MPC protocol for computing VoteRank, as described in Figure 6.

```
Input: Graph G(V, E)
Output: Computing the top-r ranked nodes based on VoteRank
 1: Initialize the (s_u, va_u) tuple for all u ∈ V as (0,1)
 2: for i = 1 to r do
 3:     for each node v ∈ V do
 4:         for u neighbor of v do
 5:             s_u ← s_u + va_v
 6:         end for
 7:     end for
 8:     v_max ← 0
 9:     for each node v ∈ V do
10:         if s_v ≥ s_{v_max} then
11:             v_max ← v
12:         end if
13:     end for
14:     Set the tuple (s_{v_max}, va_{v_max}) as (0,0)
15:     for each node v ∈ V do
16:         if v_max neighbor of v then
17:             va_v ← va_v − fact
18:             Delete v_max from the list of neighbors of v
19:         end if
20:         s_v ← 0                                  ▷ resetting all scores
21:     end for
22: end for
```

**Fig. 5.** Pseudocode for VoteRank Algorithm

## 6.1 Protocol Description

The protocol in Figure 6 takes as input the edgelist representation of the graph $(\mathcal{E}, Idx)$, such that $\mathcal{E}$ and $Idx$ are stored as ORAM arrays. The protocol begins by initializing three ORAM arrays of size $|V|$ each, named $Vote$, $Rank$ and $Mark$. The $Vote$ array tracks the voting abilities of all the nodes, while $Rank$ maintains the score of each node. The $Mark$ array ensures that a node once picked as an influential node is prohibited from participating in the voting process further. The voting ability of all the nodes is set to 1 at the start of the protocol. In each iteration of the *for*-loop between steps 3-48, we pick the top scoring node as one of the influential nodes in the network. We maintain the following private variables: $\langle cur\_indx \rangle$ to store the index of the neighbor stored in $\mathcal{E}$ currently being processed; $\langle cur\_node \rangle$ to store the current node whose neighbors are being explored; $\langle cur\_vote \rangle$ to store the voting ability of the current node; $\langle vmax \rangle$ to store the node with the maximum score so far and $\langle vmax\_score \rangle$ to store its score. The *for*-loop between steps 9-27 performs the first pass over the entries of the edgelist. In steps 10-13, the oblivious *if* construct determines whether all the neighbors of the current node are explored or not. The value of the current node is obliviously updated when the condition is evaluated to be true. Else, the private variable $\langle nbr \rangle$ stores the neighbor of the current node being explored. We update the score of node $\langle nbr \rangle$ by accounting for the vote of $\langle cur\_node \rangle$ in its score (steps 16-19) provided that the node $\langle nbr \rangle$ is unmarked. We obliviously determine if this update resulted in the $\langle nbr \rangle$ having the highest score (steps 20-23). It is reflected in the variable $\langle vmax \rangle$ accordingly. We also obliviously update the value in $\langle cur\_indx \rangle$ to hold the index to the next neighbor to be processed. At the end of the first pass over the edgelist (step 27), the variable $\langle vmax \rangle$ stores the node with the maximum score. Thus, in steps 28-30, its voting ability and score is set to zero. Further, the entry corresponding to $\langle vmax \rangle$ in array $Mark$ is set to 1. The *for*-loop between steps 33-46 performs the second scan over the elements of edgelist. Similar to the previous scan, in each iteration of the loop, we are either updating the next node to be explored or are processing a neighbor of the current node. We update the current node as and when we finish exploring all of its neighbors (steps 34-37). Additionally, we update the scores of all nodes to zero, in preparation for the voting to be performed in the next iteration. Else, if $\langle vmax \rangle$ is found in the list of neighbors, we obliviously reduce the voting ability of $\langle cur\_node \rangle$ in steps 40-43. After both the passes over the edgelist terminate, we reveal the top scoring node of the current iteration in step 47. This is repeated $r$ times to determine the top-$r$ influential nodes in the network.

## 6.2 Analytical Evaluation

### 6.2.1 Complexity

The overall complexity of the algorithm using the adjacency list representation is $\mathcal{O}(r(|E|+|V|))$ (refer Figure 5). Each iteration essentially requires $\mathcal{O}(|E|)$ operations to compute the scores of all the nodes as well as to update the voting abilities of nodes by scanning over the edges. The complexity of determining the node with the maximum score is $\mathcal{O}(|V|)$. The overall complexity results from repeating the above process $r$ times. However, this representation does not allow for a data-oblivious execution of the algorithm. Alternatively, assuming the adjacency matrix representation, the overall complex-

```
Input: Graph (E, Idx)
Output: The top-r ranked nodes based on VoteRank
 1: Initialize Vote as an ORAM array of size |V|, with each entry initialized to 1
 2: Initialize Rank and Mark as an ORAM arrays of size |V|, with each entry initialized to 0
 3: for iter = 1 to r do
 4:     ⟨cur_indx⟩ ← ⟨0⟩
 5:     ⟨cur_node⟩ ← ⟨0⟩
 6:     ⟨cur_vote⟩ ←OramRead(Vote,⟨cur_node⟩)
 7:     ⟨vmax⟩ ← ⟨0⟩
 8:     ⟨vmax_score⟩ ← ⟨0⟩
 9:     for i = 0 to |E| + |V| − 1 do
10:         ⟨temp1⟩ ← OramRead(Idx, ⟨cur_node + 1⟩)
11:         ⟨if⟩ ⟨cur_indx⟩ == ⟨temp1⟩ then
12:             ⟨cur_node⟩ ← ⟨cur_node⟩ + 1
13:             ⟨cur_vote⟩ ←OramRead(Vote,⟨cur_node⟩)
14:         ⟨else⟩
15:             ⟨nbr⟩ ← OramRead(E,⟨cur_indx⟩)
16:             ⟨if⟩ OramRead(Mark, ⟨nbr⟩) == 0 then
17:                 ⟨temp1⟩ ← OramRead(Rank,⟨nbr⟩)
18:                 ⟨temp2⟩ ← ⟨temp1⟩ + ⟨cur_vote⟩
19:                 OramWrite(Rank,⟨nbr⟩, ⟨temp2⟩)
20:                 ⟨if⟩ ⟨temp2⟩ > ⟨vmax_score⟩ then
21:                     ⟨vmax⟩ ← ⟨nbr⟩
22:                     ⟨vmax_score⟩ ← ⟨temp2⟩
23:                 end ⟨if⟩
24:             end ⟨if⟩
25:             ⟨cur_indx⟩ ← ⟨cur_indx⟩ + 1
26:         end ⟨if⟩
27:     end for
28:     OramWrite(Vote,⟨vmax⟩, ⟨0⟩)
29:     OramWrite(Rank,⟨vmax⟩, ⟨0⟩)
30:     OramWrite(Mark,⟨vmax⟩, ⟨1⟩)
31:     ⟨cur_indx⟩ ← ⟨0⟩
32:     ⟨cur_node⟩ ← ⟨0⟩
33:     for i = 0 to |E| + |V| − 1 do
34:         ⟨temp1⟩ ← OramRead(Idx, ⟨cur_node + 1⟩)
35:         ⟨if⟩ ⟨cur_indx⟩ == ⟨temp1⟩ then
36:             OramWrite(Rank,⟨cur_node⟩, ⟨0⟩)
37:             ⟨cur_node⟩ ← ⟨cur_node⟩ + 1
38:         ⟨else⟩
39:             ⟨nbr⟩ ← OramRead(E,⟨cur_indx⟩)
40:             ⟨if⟩ ⟨nbr⟩ == ⟨vmax⟩ then
41:                 ⟨temp⟩ ← OramRead(Vote,⟨cur_node⟩)
42:                 OramWrite(Vote,⟨cur_node⟩, ⟨temp − f⟩)
43:             end ⟨if⟩
44:             ⟨cur_indx⟩ ← ⟨cur_indx⟩ + 1
45:         end ⟨if⟩
46:     end for
47:     Reveal ⟨vmax⟩ to all parties
48: end for
```

**Fig. 6.** Protocol for computing $r$-top ranked nodes based on VoteRank securely

ity would amount to $\mathcal{O}(r|V|^2)$. The first pass to compute the scores of all the nodes would require access to all the entries of the adjacency matrix once, amounting to $\mathcal{O}(|V|^2)$ operations. Determining the node with the maximum score continues to have the complexity of $\mathcal{O}(|V|)$. In order to update the voting ability of all the neighbors of the top ranked node $v_{max}$, a single scan of the column representing $v_{max}$ in the adjacency matrix would require $\mathcal{O}(|V|)$ operations. Owing to the data-oblivious representation of the underlying graph, implementing the above algorithm using generic MPC constructs would also incur $\mathcal{O}(r|V|^2)$ complexity. Considering the protocol described in Figure 6, the *for*-loop in steps 3-48 is the major contributor to the overall asymptotic complexity. This loop is executed $r$ times, and within each iteration, we perform 2 scans over the list of edges in $\mathcal{O}(|E| + |V|)$ iterations each. Additionally, we perform a constant number of ORAM accesses within each of the scans. Since we consider the input graphs to be sparse (i.e. $|E| = \mathcal{O}(|V|)$), each ORAM array is of length $\mathcal{O}(|E|)$. Considering the overhead of performing ORAM read/write operations to be $f(n)$ over an array of length $n$, the overall complexity of the designed protocol boils down to $\mathcal{O}(r(|E| + |V|)f(\mathcal{O}(|E|)))$.

### 6.2.2 Security

The correctness of the protocol follows directly from the instruction sequence and protocol description provided in Section 6.1. The proof of privacy follows along the same lines as described for the previous protocols, including loops with fixed iterations, ORAM arrays, oblivious *if-else* constructs and non-disclosure of intermediary private values.

# 7 Related Work

As discussed previously, the problem of identifying influential nodes in a network are well investigated [3, 7, 11, 12]. Each of these approaches computes the influence score of nodes based on different network characteristics. However, all of them consider that the structure of the underlying network is known centrally and are not designed to cater to the privacy of the involved individuals. There are only a few works that have addressed the problem of determining the top-$r$ influential nodes when the underlying network structure is private and/or is not known centrally. Tassa and Bonchi [61] propose MPC protocols to compute the strength of each edge and the influence score of each node in a social network. Unlike the current work, the authors in [61] consider the scenario where one of the parties (i.e. host) has the entire social network network $G(V, E)$ on a set of users as its private data. Other parties (i.e. service providers) have individual activity logs as their private data, that captures the various actions performed by the users. The objective is for the host and the service providers to quantify the influence capabilities of edges as well as nodes in the graph without each disclosing his private data. Thus, the data model considered in [61] is different from the current work. The computed measures are specific to the application scenario considered, unlike the generic protocols designed in the current work. Additionally, the protocols designed in [61] require that the host sends across a set of edges $E'$ to each of the service providers. Since this edge set $E'$ is a superset of the edges of the true graph (i.e. $E' \supset E$), it allows the service providers to conclude that if an edge $(v_i, v_j) \notin E'$, then they are sure that $(v_i, v_j) \notin E$ as well. This constitutes leaking structural

information of the underlying graph.

The problem of computing network measures securely when the entire graph is distributedly known to a set of parties has been previously addressed by Kukkala et al. [46]. In particular, they design MPC protocols to compute the overall degree distribution of the network, closeness centrality of each node, securely implementing the PageRank and the $k$-Shell decomposition algorithms. Although some of the problems addressed in [46] overlap with the current work, we would like to briefly highlight some of the differences. Firstly, the authors in [46] design protocols for the multiparty setting while we design protocols for the two-party setting. Moreover, the solution proposed in [46] is designed in the circuit model, whose drawbacks have been discussed earlier. To counter this, the current work designs protocols using RAM-SC which facilitates more efficient solutions. Secondly, the use of the adjacency matrix representation in [46] also adds to the inefficiency of the protocols, when considering real world networks. We improve this by using an alternative graph representation known as the edgelist representation which fares better. Thirdly, the centrality values computed in both the works vary in their algorithmic approach. That is, the PageRank algorithm designed in [46] is based on the *random-walk* model while the protocol in the current is designed based on the *iterative* approach. Similarly in the case of $k$-Shell decomposition, the approach to pruning nodes significantly varies. These variations in the underlying algorithm also add to the asymptotic complexity of the protocols designed based on them. We implement the protocols designed in both the works, the results of which are discussed in Section 8.

# 8 Experimental Results

## 8.1 Setup

The performance of the protocols in practice is gauged by implementing them on the popular framework called *Obliv-C* [22]. It is a C-language based framework that allows for compiling and executing secure computation protocols using garbled circuits at the back-end. The framework incorporates most of the recent optimizations such as the use of free XOR gates [62], efficient OT and its extensions [63], the use of half gates [64] and fixed-key block ciphers [65]. We also use the ACK library [66] in conjunction with Obliv-C since it provides efficient implementations of several ORAM constructions. All the protocols were executed on a 64-bit machine with Intel Xeon E7-8870 v3 CPU clocking at 2.10 GHz and 16 GB RAM under Ubuntu 16.04 operating system. Further, the programs were compiled using version 5.4.0 of gcc compiler executed on a single thread.

The dataset considered as input to the protocols consists of synthetically generated sparse graphs that mimic real world networks. Since the run time only depends on the number of nodes ($|V|$) and edges ($|E|$) in the graph, each input consists of a randomly picked *Erdos-Renyi* graph parameterized by $|V|$ and $|E|$. In order to maintain the sparsity of the graphs, we consider the number of edges to be a constant factor times the number of nodes (i.e. $|E| = 10|V|$). The factor is appropriately chosen as 10 by taking into account real world networks provided on SNAP [49]. We consider small sized datasets owing to memory constraints and prolonged runtime. In order to draw a comparison, we also implement the protocols designed in Kukkala et al. [46] for the $k$-Shell decomposition and the random surfer method of computing PageRank centrality. To have a fair comparison between the two set of protocols designed for the same network measure, we assume that the underlying network data is collectively known by 2 parties $P_1$ and $P_2$ in both the cases. This way, the highlight is on the trade-offs achieved between the adjacency matrix and the edgelist representation of the underlying graph. Furthermore, by choosing to evaluate the protocols under the same generic scheme of Yao's garbled circuits, we are able to achieve a comparison of the algorithmic complexity between the corresponding set of protocols. The party $P_1$ is assumed to be representative of a set of nodes $V_1 \subset V$ while $P_2$ is representative of $V_2 \subset V$, such that $V_1 \cup V_2 = V$ and $V_1 \cap V_2 = \phi$. In the case of the edgelist representation, we assume that $P_1$ distributes shares of all the outgoing edges of nodes in $V_1$, while $P_2$ shares those in $V_2$. Similarly, for the case of the adjacency matrix representation, party $P_1$ distributes shares of all the entries of the rows corresponding to the nodes in $V_1$, while $P_2$ shares the entries of the remaining rows. Since the total number of edges are known to be public and have only two data owners, we do not require to hide the number of edges being reported by each of the parties through padding of dummy edges. Having obtained shares of all the edges, party $P_1$ and $P_2$ perform oblivious sort on the list of shares to obtain the shares of edgelist as described in Section 3.1. We report the run time of the protocols under the above conditions. In the case where the network is known by multiple data holders, the cost of constructing the edgelist is briefly described in Section 8.2.1.

## 8.2 Results

In the current work, we compare the performance of the designed protocols under the linear scan and the Square-Root ORAM [44] schemes. The former refers to the trivial ORAM scheme which involves linearly scanning through the entire array each time an element at a private index is to be retrieved. These ORAM constructions are used in conjunction with the edgelist representation of the graph. The protocol to securely compute $k$-Shell decomposition using the adjacency matrix [46] has an overall complexity of $\mathcal{O}(|V|^3)$. Thus, considering only the highest order term contributing to the asymptotic complexity, we expect to observe an approximate eightfold increase in the run time of the protocol when we increase the number of nodes by twofold. We note that the reported run time respects the mentioned asymptotic complexity since we observe an average increase in the run time by a factor of 7.6. We note that, throughout the following discussion, we always refer to a twofold increase in the number of nodes. Alternatively, when the edgelist representation is considered, the resulting asymptotic complexity is $\mathcal{O}(|V|^2)$ assuming the linear scan ORAM scheme, while it is $\mathcal{O}(|V|^{1.5}\log^{1.5}(|V|))$ operations under the Square-Root ORAM scheme, assuming sparse input graphs. As described in the previous case, we expect to observe an approximate increase by a factor of 4 and 2.8 respectively in each case. The run time reported in Table 1 shows an average increase by a factor of 3.7 in the case of the linear scan and by an average of 2.7 in the case of the Square-Root ORAM.

In the case of the PageRank algorithm, the protocol in [46] computes the centrality value based on the random surfer model. The run time reported for the same considers random walks of length $10|V|$ and hence results in an overall asymptotic complexity of $\mathcal{O}(|V|^3)$ operations. Thus, as expected, the run time for the protocol increases by eightfold. However, the protocol for computing PageRank centrality in the current work uses the iterative approach. The run time for the protocol considers $\log(|V|)$ iterations of the algorithm, resulting in an overall complexity of $\mathcal{O}(|V|^2\log(|V|))$ using linear scan and $\mathcal{O}(|V|^{1.5}\log^{2.5}(|V|))$ using the Square-Root ORAM scheme, for the considered sparse graphs. We observe that the run time for the linear scan variant approximately quadruples, while in the case of the Square-Root ORAM the run time approximately triples ($\approx 2^{1.5}$). Furthermore, we note that that the PageRank centrality value of each node reaches convergence within the $\log(|V|)$ iterations of the iterative approach, unlike in the random surfer model even with $10|V|$ iterations. The loss due to precision in our fixed point representation results in marginal error in the absolute value of the PageRank centrality. However, the relative ranking of the nodes is preserved allowing the identification of the top-$k$ influential nodes.

We are the first to implement the VoteRank protocol under both the graph representations. The protocols are designed to identify the top 10% of the nodes based on their VoteRank. Thus, the overall complexity of the protocol using the adjacency matrix representation is of $\mathcal{O}(|V|^3)$ operations. This tallies with the average eightfold increase in the run time of the protocol under this representation. In the case of the edgelist representation, the overall complexity is of $\mathcal{O}(|V|^3)$ operations using linear scan and $\mathcal{O}(|V|^{2.5}\log^{1.5}|V|)$ operations under the Square-Root ORAM scheme. The run time for these variants correspondingly increases by a factor of 8 and 5.6, on an average. Unlike the previous centrality measures, the break-even points for the variants of the VoteRank protocols is not captured in Table 1. The VoteRank protocol has a higher asymptotic complexity compared to the other two network measure protocols under the Square-Root ORAM scheme. Furthermore, since the difference in the asymptotic complexity of the variants of the VoteRank protocol is small, the scale of the experiments performed do not suffice to bring out the break-even points clearly. This is the reason that the VoteRank protocol with the adjacency matrix representation appears to outperform the edgelist variants.

Since the run time reported in Table 1 tallies with the asymptotic complexity, we conclude that the protocols for computing $k$-Shell decomposition and PageRank centrality designed in the current work outperforms the protocols proposed by Kukkala et al. [46].

### 8.2.1 Edgelist Construction

When the network is collectively known by multiple data holders, say $D_1, D_2, \ldots D_n$, we assume that each data holder $D_i$ is representative of a set of nodes $V_i \subset V$ such that $\bigcup_i^n V_i = V$ and $\bigcap_i^n V_i = \phi$. The outgoing edges of the nodes in $V_i$ is considered to be the private data of $D_i$. This is similar to the assumption made in the 2-party scenario described previously. For each node $u \in V_i$, data holder $D_i$ distributes $|V-1|$ shares among the computing parties $P_1$ and $P_2$. Each share corresponds to either a valid edges $(u, v)$ or a dummy edge denoted by $(\infty, \infty)$. Since each node can have a maximum of $|V|-1$ outgoing links to every other node in the network, we consider distributing $|V|-1$ shares of

| No. of Nodes | $k$-Shell | | | PageRank | | | VoteRank | | |
|---|---|---|---|---|---|---|---|---|---|
| | Edgelist | | Adjacency Matrix[46] | Edgelist | | Adjacency Matrix[46] | Edgelist | | Adjacency Matrix |
| | Linear | SQRT | | Linear | SQRT | | Linear | SQRT | |
| 32 | 6s | 13s | 3s | 22s | 28s | 3m 59s | 16s | 21s | 1s |
| 64 | 23s | 31s | 27s | 1m 28s | 1m 19s | 31m 54s | 2m 10s | 1m 37s | 7s |
| 128 | 1m 30s | 1m 25s | 3m 8s | 5m 4s | 3m 23s | 4h 17m | 16m 38s | 8m 50s | 53s |
| 256 | 5m 43s | 4m 9s | 25m 21s | 22m 49s | 11m 34s | 34h 12m 52s | 2h 21m 47s | 56m 53s | 7m 7s |
| 512 | 22m 27s | 12m 53s | 3h 19m 44s | 1h 41m 58s | 40m 17s | 11d 9h 42m 57s* | 19h 2m 28s | 6h 16m 49s | 58m 13s |

**Table 1.** We report the run time of the various protocols for different graph representations. The column labels Linear and SQRT represent the usage of the linear scan and the Square Root ORAM schemes respectively. The starred value is an estimate.

data. Having obtained shares of all the edges from the data holders, party $P_1$ and $P_2$ perform oblivious sort on the list of shares to obtain the shares of edgelist as described in Section 3.1. Each of the data holders could simultaneously distribute shares of their data to parties $P_1$ and $P_2$. Thus, the cost of constructing the edgelist is only affected by the length of the list that needs to be obliviously sorted rather on the number of data holders. The run time for constructing the edgelist from an unsorted list of reported edges, having length $\mathcal{O}(V^2)$ as described above, is reported in Table 2. We consider the input graphs from the same dataset and obliviously sort the edges using Batcher's algorithm.

| No. of Nodes | Run Time |
|---|---|
| 32 | 1s |
| 64 | 6s |
| 128 | 30s |
| 256 | 2m 34s |
| 512 | 12m 43s |

**Table 2.** Edgelist construction with multiple data holders.

# 9 Conclusions

A steady increase in the perceived threat to privacy has been the major reason for the changing trends in performing social network analysis. Thus, developing a privacy preserving approach to mining social network data is the need of the hour. In this work, we describe the use of secure multiparty computation as a solution to analyzing social network data that is distributedly present. More precisely, we focus on identifying the influential spreaders in the network by securely performing $k$-Shell decomposition, PageRank and VoteRank algorithms. By implementing our protocols, we conclude that these can be deployed for practical application scenarios such as supply chain networks [17], as an alternative to performing analysis using a trusted third party. The presented protocols are a first step towards achieving privacy, and can be improved by considering more challenging adversarial models. Improvements in the concrete run time can be tested using hybrid approaches over garbled circuits.

# 10 Acknowledgements

This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

# References

[1] HuffPost-Staff. Whatsapp launches research grants up to $50,000 to fight fake news. [Online]. Available: https://www.huffingtonpost.in/2018/07/09/whatsapp-launches-research-grants-to-fight-misinformation-offering-up-to-50-000-per-proposal_a_23477400/

[2] K. Bikoff. Sice receives $1.2 million as part of a darpa grant to study information spread. [Online]. Available: https://www.sice.indiana.edu/news/story.html?story=SICE-faculty-part-of-1.2-million-DARPA-grant-to-study-information-spread

[3] J.-X. Zhang, D.-B. Chen, Q. Dong, and Z.-D. Zhao, "Identifying a set of influential spreaders in complex networks," *Scientific reports*, vol. 6, p. 27823, 2016.

[4] R. Pastor-Satorras, C. Castellano, P. Van Mieghem, and A. Vespignani, "Epidemic processes in complex networks," *Reviews of modern physics*, vol. 87, no. 3, p. 925, 2015.

[5] M. Yahya, "Polio vaccines- "no thank you!" barriers to polio eradication in northern nigeria," *African Affairs*, vol. 106, no. 423, pp. 185–204, 2007.

[6] D. Kempe, J. Kleinberg, and É. Tardos, "Maximizing the spread of influence through a social network," in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2003, pp. 137–146.

[7] S. Aral and D. Walker, "Identifying influential and susceptible members of social networks," *Science*, p. 1215842, 2012.

[8] J. Ugander, L. Backstrom, C. Marlow, and J. Kleinberg, "Structural diversity in social contagion," *Proceedings of the National Academy of Sciences*, p. 201116502, 2012.

[9] E. Bakshy, I. Rosenn, C. Marlow, and L. Adamic, "The role of social networks in information diffusion," in *Proceedings of the 21st international conference on World Wide Web*. ACM, 2012, pp. 519–528.

[10] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.

[11] M. Kitsak, L. K. Gallos, S. Havlin, F. Liljeros, L. Muchnik, H. E. Stanley, and H. A. Makse, "Identification of influential spreaders in complex networks," *Nature physics*, vol. 6, no. 11, p. 888, 2010.

[12] J. Weng, E.-P. Lim, J. Jiang, and Q. He, "Twitterrank: finding topic-sensitive influential twitterers," in *Proceedings of the third ACM international conference on Web search and data mining*. ACM, 2010, pp. 261–270.

[13] W. Chen, Y. Wang, and S. Yang, "Efficient influence maximization in social networks," in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2009, pp. 199–208.

[14] S. Ji, L. Lu, C. H. Yeung, and Y. Hu, "Effective spreading from multiple leaders identified by percolation in social networks," *arXiv preprint arXiv:1508.04294*, 2015.

[15] F. Kerschbaum, A. Schroepfer, A. Zilli, R. Pibernik, O. Catrina, S. de Hoogh, B. Schoenmakers, S. Cimato, and E. Damiani, "Secure collaborative supply-chain management," *Computer*, no. 9, pp. 38–43, 2011.

[16] G. Fridgen and T. Z. Garizy, "Supply chain network risk analysis-a privacy preserving approach." in *ECIS*, 2015.

[17] Y. Kim, T. Y. Choi, T. Yan, and K. Dooley, "Structural investigation of supply networks: A social network analysis approach," *Journal of Operations Management*, vol. 29, no. 3, pp. 194–211, 2011.

[18] M. Fire and R. Puzis, "Organization mining using online social networks," *Networks and Spatial Economics*, vol. 16, no. 2, pp. 545–578, 2016.

[19] P. Glasserman and H. P. Young, "How likely is contagion in financial networks?" *Journal of Banking & Finance*, vol. 50, pp. 383–399, 2015.

[20] F. Kerschbaum and A. Schaad, "Privacy-preserving social network analysis for criminal investigations," in *Proceedings of the 7th ACM workshop on Privacy in the electronic society*. ACM, 2008, pp. 9–14.

[21] V. B. Kukkala and S. Iyengar, "Computing betweenness centrality: An efficient privacy-preserving approach," in *International Conference on Cryptology and Network Security*. Springer, 2018, pp. 23–42.

[22] S. Zahur and D. Evans, "Obliv-c: A language for extensible data-oblivious computation." *IACR Cryptology ePrint Archive*, vol. 2015, p. 1153, 2015.

[23] A. C. Yao, "Protocols for secure computations," in *Foundations of Computer Science, 1982. SFCS'08. 23rd Annual Symposium on*. IEEE, 1982, pp. 160–164.

[24] J. Alwen, R. Ostrovsky, H.-S. Zhou, and V. Zikas, "Incoercible multi-party computation and universally composable receipt-free voting," in *Annual Cryptology Conference*. Springer, 2015, pp. 763–780.

[25] A. Aly and M. Van Vyve, "Practically efficient secure single-commodity multi-market auctions," in *International Conference on Financial Cryptography and Data Security*. Springer, 2016, pp. 110–129.

[26] S. Jha, L. Kruger, and V. Shmatikov, "Towards practical privacy for genomic computation," in *Security and Privacy, 2008. SP 2008. IEEE Symposium on*. IEEE, 2008, pp. 216–230.

[27] G. Asharov and Y. Lindell, "A full proof of the bgw protocol for perfectly secure multiparty computation," *Journal of Cryptology*, vol. 30, no. 1, pp. 58–151, 2017.

[28] M. Ben-Or, S. Goldwasser, and A. Wigderson, "Completeness theorems for non-cryptographic fault-tolerant distributed computation," in *Proceedings of the twentieth annual ACM symposium on Theory of computing*. ACM, 1988, pp. 1–10.

[29] O. Goldreich, S. Micali, and A. Wigderson, "How to play any mental game," in *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. ACM, 1987, pp. 218–229.

[30] A. C.-C. Yao, "How to generate and exchange secrets," in *Foundations of Computer Science, 1986., 27th Annual Symposium on*. IEEE, 1986, pp. 162–167.

[31] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *Journal of the ACM (JACM)*, vol. 43, no. 3, pp. 431–473, 1996.

[32] B. Pinkas and T. Reinman, "Oblivious ram revisited," in *Annual Cryptology Conference*. Springer, 2010, pp. 502–519.

[33] M. T. Goodrich and M. Mitzenmacher, "Privacy-preserving access of outsourced data via oblivious ram simulation," in *International Colloquium on Automata, Languages, and Programming*. Springer, 2011, pp. 576–587.

[34] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li, "Oblivious ram with o ((logn) 3) worst-case cost," in *International Conference on The Theory and Application of Cryptology and Information Security*. Springer, 2011, pp. 197–214.

[35] E. Stefanov, E. Shi, and D. Song, "Towards practical oblivious ram," *arXiv preprint arXiv:1106.3652*, 2011.

[36] C. W. Fletcher, M. v. Dijk, and S. Devadas, "A secure processor architecture for encrypted computation on untrusted programs," in *Proceedings of the seventh ACM workshop on Scalable trusted computing*. ACM, 2012, pp. 3–8.

[37] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song, "Phantom: Practical oblivious computation in a secure processor," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 311–324.

[38] D. Apon, J. Katz, E. Shi, and A. Thiruvengadam, "Verifiable oblivious storage," in *International Workshop on Public Key Cryptography*. Springer, 2014, pp. 131–148.

[39] M. Keller and P. Scholl, "Efficient, oblivious data structures for mpc," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2014, pp. 506–525.

[40] S. D. Gordon, J. Katz, V. Kolesnikov, F. Krell, T. Malkin, M. Raykova, and Y. Vahlis, "Secure two-party computation in sublinear (amortized) time," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 513–524.

[41] C. Gentry, K. A. Goldman, S. Halevi, C. Julta, M. Raykova, and D. Wichs, "Optimizing oram and using it efficiently for secure computation," in *International Symposium on Privacy Enhancing Technologies Symposium*. Springer, 2013, pp. 1–18.

[42] X. S. Wang, Y. Huang, T. H. Chan, A. Shelat, and E. Shi, "Scoram: oblivious ram for secure computation," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 191–202.

[43] X. Wang, H. Chan, and E. Shi, "Circuit oram: On tightness of the goldreich-ostrovsky lower bound," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 850–861.

[44] S. Zahur, X. Wang, M. Raykova, A. Gascón, J. Doerner, D. Evans, and J. Katz, "Revisiting square-root oram: efficient random access in multi-party computation," in *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016, pp. 218–234.

[45] Keyhubs - uncover the hidden organization. [Online]. Available: http://www.keyhubs.com/

[46] V. B. Kukkala, J. S. Saini, and S. Iyengar, "Privacy preserving network analysis of distributed social networks," in *International Conference on Information Systems Security*. Springer, 2016, pp. 336–355.

[47] A. Aly and M. Van Vyve, "Securely solving classical network flow problems," in *International Conference on Information Security and Cryptology*. Springer, 2014, pp. 205–221.

[48] A. Aly, E. Cuvelier, S. Mawet, O. Pereira, and M. Van Vyve, "Securely solving simple combinatorial graph problems," in *International Conference on Financial Cryptography and Data Security*. Springer, 2013, pp. 239–257.

[49] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[50] M. Ajtai, J. Komlós, and E. Szemerédi, "An 0 (n log n) sorting network," in *Proceedings of the fifteenth annual ACM symposium on Theory of computing*. ACM, 1983, pp. 1–9.

[51] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, spring joint computer conference*. ACM, 1968, pp. 307–314.

[52] M. T. Goodrich, "Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in o (n log n) time," in *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*. ACM, 2014, pp. 684–693.

[53] ——, "Randomized shellsort: A simple oblivious sorting algorithm," in *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, 2010, pp. 1262–1277.

[54] ——, "Spin-the-bottle sort and annealing sort: Oblivious sorting via round-robin random comparisons," in *2011 Proceedings of the Eighth Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*. SIAM, 2011, pp. 93–106.

[55] S. B. Seidman, "Network structure and minimum degree," *Social networks*, vol. 5, no. 3, pp. 269–287, 1983.

[56] S. Pei and H. A. Makse, "Spreading dynamics in complex networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2013, no. 12, p. P12002, 2013.

[57] V. Batagelj and M. Zaversnik, "An o (m) algorithm for cores decomposition of networks," *arXiv preprint cs/0310049*, 2003.

[58] R. S. Wahby, S. T. Setty, Z. Ren, A. J. Blumberg, and M. Walfish, "Efficient ram and control flow in verifiable outsourced computation." in *NDSS*, 2015.

[59] Q. Liu, B. Xiang, N. J. Yuan, E. Chen, H. Xiong, Y. Zheng, and Y. Yang, "An influence propagation view of pagerank," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 11, no. 3, p. 30, 2017.

[60] N. Ma, J. Guan, and Y. Zhao, "Bringing pagerank to the citation analysis," *Information Processing & Management*, vol. 44, no. 2, pp. 800–810, 2008.

[61] T. Tassa and F. Bonchi, "Privacy preserving estimation of social influence." in *EDBT*, 2014, pp. 559–570.

[62] V. Kolesnikov and T. Schneider, "Improved Garbled Circuit: Free XOR Gates and Applications," in *International Colloquium on Automata, Languages, and Programming*. Springer, 2008, pp. 486–498.

[63] G. Asharov, Y. Lindell, T. Schneider, and M. Zohner, "More Efficient Oblivious Transfer and Extensions for Faster Secure Computation," in *SIGSAC Conference on Computer & Communications Security*. ACM, 2013, pp. 535–548.

[64] S. Zahur, M. Rosulek, and D. Evans, "Two Halves Make a Whole," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2015, pp. 220–250.

[65] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway, "Efficient Garbling from a Fixed-key Blockcipher," in *Security and Privacy (SP)*. IEEE, 2013, pp. 478–492.

[66] J. Doerner, "The Absentminded Crypto Kit," "https://bitbucket.org/jackdoerner/absentminded-crypto-kit.git".

[67] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *science*, vol. 286, no. 5439, pp. 509–512, 1999.

[68] M. Girvan and M. E. Newman, "Community structure in social and biological networks," *Proceedings of the national academy of sciences*, vol. 99, no. 12, pp. 7821–7826, 2002.

[69] S. Fortunato, "Community detection in graphs," *Physics reports*, vol. 486, no. 3-5, pp. 75–174, 2010.

[70] S. P. Borgatti and M. G. Everett, "Models of core/periphery structures," *Social networks*, vol. 21, no. 4, pp. 375–395, 2000.

[71] J. Travers and S. Milgram, "The small world problem," *Phychology Today*, vol. 1, no. 1, pp. 61–67, 1967.

[72] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world'networks," *nature*, vol. 393, no. 6684, p. 440, 1998.

[73] T. Y. Choi and Y. Hong, "Unveiling the structure of supply networks: case studies in honda, acura, and daimlerchrysler," *Journal of Operations Management*, vol. 20, no. 5, pp. 469–493, 2002.

# A Social Network Analysis

A social network captures the intricate web of interactions among social entities, where each entity could be representative of an individual or even a group of indi-

viduals (e.g. an organization). The type of interaction captured in a social network can also largely vary, some of which include friendships, email communications, scientific collaboration, relationship of trust among employees, buyer-seller interactions among organizations, etc. Although these interactions seemingly appear different, it has been observed that there are several commonalities that characterize the underlying network. The existence of scale-free degree distribution [67], presence of a community structure [68, 69], core-periphery structure [70] and the property of *small-world* [71, 72] are a few examples of the characteristics commonly observed across different social networks. Thus, social network analysis deals with modeling social interactions as a graph consisting of nodes and edges, followed by applying the common set of tools and techniques developed to study the structural properties of the underlying network. Characterizing nodes and edges through various centrality measures, detecting communities, identifying influential nodes, determining the spreading pattern, etc. are some of the crucial problems addressed in the field.

# B Iterative PageRank using Adjacency Matrix

The detailed description of computing PageRank centrality assuming the adjacency matrix graph representation is provided through the pseudocode in Figure 7. This iterative method to compute PageRank is oblivious since the execution of the instructions are independent of the input graph structure. This approach to computing PageRank centrality would require $\mathcal{O}(l|V|^2)$ operations. This is majorly attributed to the $l$ iterations of the PageRank algorithm and accessing the entries of the update matrix $N$ of size $|V|^2$ within each iteration. Since the above algorithm is inherently data-oblivious, implementing the same using generic MPC constructs would continue to have the same asymptotic complexity of $\mathcal{O}(l|V|^2)$.

# C Real world networks

In this section, we consider the supply chain networks reported by Choi and Hong [73] and investigated by Kim et al. [17]. The networks consist of three product lines including Honda Accord, Acura CL/TL, and Daimler-Chrysler (DCX) Grand Cherokee. In specific, we consider the contractual relationships that exist between the different organizations that are a part of the supply chain network, under each of the three product lines. Using these networks as input to the designed protocols, we report the run time for each of the computed network measures. As in the previous case, we assume that the network is distributedly known between two parties who run the designed protocols. The results are presented in Table 3.

**Input:** Graph $G(V, E)$, $l$ (number of iterations), $s$ (redistribution parameter)
**Output:** The PageRank values of all the nodes
1: Determine the entries of the update matrix $N = [N_{ij}]$
2: Initialize $r_i^{(0)} \leftarrow 1/n$ for $1 \leq i \leq n$ (all other $r^{(k)}$ vectors are initialized as zero vectors for $k \geq 1$)
3: **for** $k = 1$ to $l$ **do**
4:  **for** $i = 1$ to $n$ **do**
5:  **for** $j = 1$ to $n$ **do**
6:  $r_i^{(k)} \leftarrow r_i^{(k)} + N_{ji} r_j^{(k-1)}$
7:  **end for**
8:  **end for**
9: **end for**

**Fig. 7.** Pseudocode for the iterative method to PageRank Algorithm using adjacency matrix

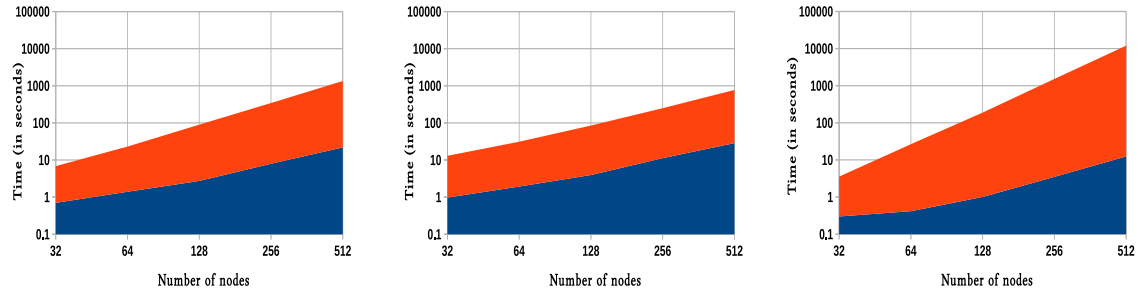
# D Performance Analysis

The run time for evaluating all the designed protocols, reported in Table 1, accounts for the overall time taken in executing them. To present a more thorough analysis, we include plots in Figures 8-10 that account for majorly two phases of the overall run time. Each protocol execution begins with the two parties, who collectively hold the network data, distributing shares of their private data to one another. Using these shares, the two parties collectively generate shares for the edge-list or the adjacency matrix representation, based on the graph representation under consideration. They also set up the required private variables, arrays as well as the ORAM structure. This constitutes the *initialization* phase of the protocol. This is followed by the *iterative* phase of the protocol. Each of the designed protocols includes a central looping construct that essentially scans through the input graph and performs the major part of the required computation of securely computing the centrality values. It is this loop that forms the crux of the protocol and the execution of the same is referred to as the iterative phase. The results are presented using a log-log plot in Figures 8-10. We note that the curve

denoting the overall run time is close to a straight line. Since the number of edges considered in the graph is a function of the number of nodes, the overall run time is directly expressed as a function of the number of nodes, as described in Section 8.2. We note that the slope of the straight line observed in each of the plots matches with the exponent of the independent variable, in our case it is the number of nodes $|V|$. This confirms that the run time of the protocols tallies with the previously mentioned asymptotic complexities. As an alternative to measuring the run time of the protocols, we found that the number of gates required to evaluate them is a better metric to gauge the performance of the protocols. The rate at which the gate count increases as we increase the size of the input graphs, corroborates with the asymptotic complexity better than as seen in the case of run time of the protocols. Due to space constraints, we do not report the same.

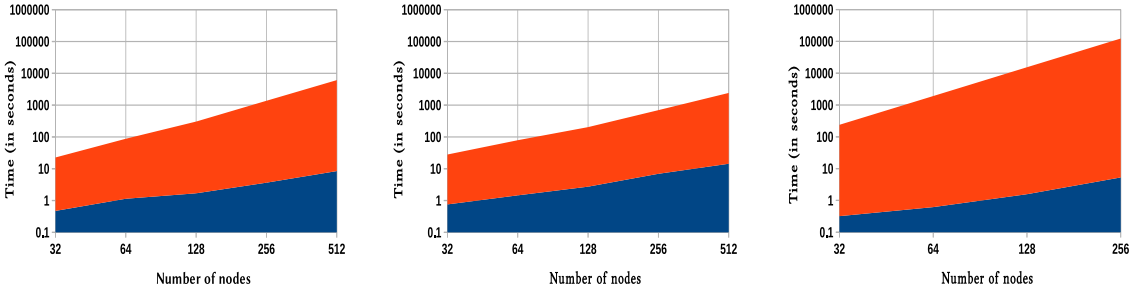
| Network | $k$-Shell | | | PageRank | | | VoteRank | | |
|---|---|---|---|---|---|---|---|---|---|
| | Edgelist | | Adjacency | Edgelist | | Adjacency | Edgelist | | Adjacency |
| | Linear | SQRT | Matrix[46] | Linear | SQRT | Matrix[46] | Linear | SQRT | Matrix |
| DCX | 1s | 3s | 3s | 4s | 6s | 2m 21s | 1s | 3s | 1s |
| Accord | 1s | 3s | 3s | 5s | 7s | 2m 33s | 1s | 3s | 1s |
| Acura | 2s | 4s | 4s | 5s | 9s | 4m 23s | 3s | 5s | 2s |

**Table 3.** We report the run time of the various protocols for three supply chain networks. The network size of DCX, Accord and Acura are 27, 28 and 34 nodes respectively. The corresponding edge count for each of the networks is 52, 56 and 74 edges, by accounting for both directions of the contractual relationship.



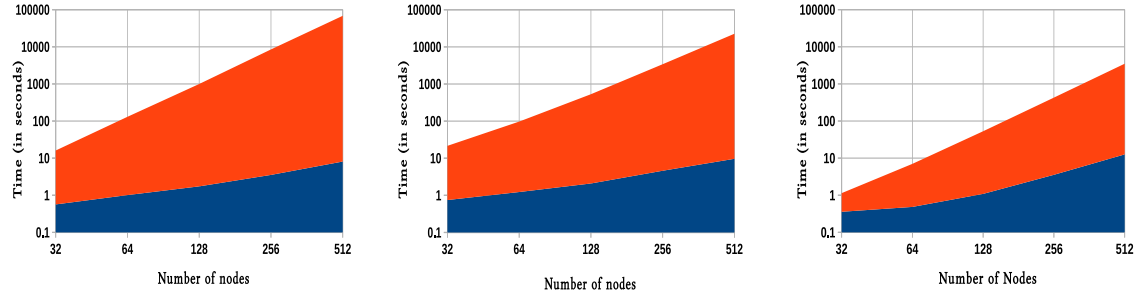

**(a)** Linear Scan **(b)** Square-Root ORAM **(c)** Adjacency Matrix

**Fig. 8.** Secure variants of the $k$-Shell decomposition protocol




**(a)** Linear Scan **(b)** Square-Root ORAM **(c)** Adjacency Matrix

**Fig. 9.** Secure variants of the PageRank protocol




**(a)** Linear Scan **(b)** Square Root ORAM **(c)** Adjacency Matrix

**Fig. 10.** Secure variants of the VoteRank protocol

The run time reported in each of the sub-figures accounts for two phases in the execution of the protocol. The blue region denotes the *initialization phase*. The region in orange denotes the *iterative phase*.