Find Thy Neighbourhood: Privacy-Preserving Local Clustering

Pranav Shriram A
National Institute of Technology
Tiruchirappalli
Tamil Nadu, India
pranavshriram99@gmail.com

Nishat Koti
Indian Institute of Science
Bangalore, India
kotis@iisc.ac.in

Varsha Bhat Kukkala
Indian Institute of Science
Bangalore, India
varshak@iisc.ac.in

Arpita Patra
Indian Institute of Science
Bangalore, India
arpita@iisc.ac.in

Bhavish Raj Gopal
Indian Institute of Science
Bangalore, India
bhavishraj@iisc.ac.in

ABSTRACT

Identifying a cluster around a seed node in a graph, termed *local clustering*, finds use in several applications, including fraud detection, targeted advertising, community detection, etc. However, performing local clustering is challenging when the graph is distributed among multiple data owners, which is further aggravated by the privacy concerns that arise in disclosing their view of the graph. This necessitates designing solutions for privacy-preserving local clustering and is addressed for the first time in the literature. We propose using the technique of secure multiparty computation (MPC) to achieve the same. Our local clustering algorithm is based on the heat kernel PageRank (HKPR) metric, which produces the best-known cluster quality. En route to our final solution, we have two important steps: (i) designing data-oblivious equivalent of the state-of-the-art algorithms for computing local clustering and HKPR values, and (ii) compiling the data-oblivious algorithms into its secure realisation via an MPC framework that supports operations over fixed-point arithmetic representation such as multiplication and division. Keeping efficiency in mind for large graphs, we choose the best-known honest-majority 3-party framework of SWIFT (Koti et al., USENIX'21) and enhance it with some of the necessary yet missing primitives, before using it for our purpose. We benchmark the performance of our secure protocols, and the reported run time showcases the practicality of the same. Further, we perform extensive experiments to evaluate the accuracy loss of our protocols. Compared to their cleartext counterparts, we observe that the results are comparable and thus showcase the practicality of the designed protocols.

KEYWORDS

privacy-preserving local clustering, privacy-preserving heat kernel PageRank, secure multiparty computation

1 INTRODUCTION

Many real-world applications, such as communication networks, traffic networks, social networks, etc., generate an enormous amount of unstructured data. A natural and conventional approach to model such data is via graphs [57] owing to their highly expressive capabilities and ease of processing. Specifically, modelling a system as a graph involves representing each entity as a node and capturing their interactions as edges. Further, various techniques enable deriving meaningful information about the modelled system. One such useful approach is that of clustering, which allows analyzing the topology of a graph to identify entities that are related to each other [10, 21, 32, 46, 54]. At a high level, clustering is the process of grouping together similar nodes in a graph and finds use in several applications such as community detection in social networks [26, 30, 45, 60], behavioural analysis [15, 36], structural characterization of chemical networks [16, 25, 31, 51], etc.

Most clustering algorithms are designed to categorize every node into its specific cluster, termed *global* clustering. However, more often than not, one may be interested in identifying a *local* cluster around a specific node. For example, consider the COVID-19 contact network, where users are modelled as nodes, and their interactions are modelled as edges. Identifying the close-knit cluster around a user who has recently contracted the virus is important and enables the implementation of preventive measures. A global clustering algorithm may not correctly identify such a local cluster. Other examples, where identifying a local cluster is of importance, include targeted advertising [5], fraud detection [37], etc.

Graph-based local clustering algorithms in the literature [17, 34, 55, 56] assume that the topology of the graph is available in its entirety. However, this may not always be the case. For instance, as described earlier, contact networks may be held in a distributed fashion where each node is aware only of its neighbouring nodes. In general, the graph may be distributed across multiple data owners, such that each of them is aware of only a subset of the edges in the overall graph. Performing local clustering on distributed graphs thus requires data owners to disclose their view of the graph. However, data privacy concerns prevent them from doing so. Hence, the distributed nature of the global graph clubbed with privacy concerns makes it challenging to perform local clustering when accounting for the entire graph topology. We illustrate the above challenge with the help of a plausible use case of fraud detection.

Fraudulent account detection in banks. To closely monitor suspicious accounts that could possibly be a part of fraudulent transactions, consider the problem of computing a local cluster around a *given* fraudulent account. Identifying such a cluster requires the participation of multiple banks and cannot be performed locally by an

Proceedings on Privacy Enhancing Technologies YYYY(X), 1–17
© YYYY Copyright held by the owner/author(s).
https://doi.org/XXXXXXX.XXXXXXX

individual bank. For example, consider a fraudulent account Acc_X in bank Bank_A. Note that Acc_X could have its accomplice (fraudulent) accounts in other banks that could transact with Acc_X via intermediate accounts to camouflage fraud. Hence, Bank_A cannot locally determine the cluster around Acc_X, and may require investigating accounts that lie beyond the immediate neighbourhood of Acc_X. Thus, to identify such fraudulent accounts, banks require knowledge of the global transaction graph, where a node represents an account, and an edge captures the financial transaction between the corresponding accounts. This global transaction graph is distributed across multiple banks, with each bank being aware of transactions pertaining to only its own accounts. Thus, identifying the cluster requires cooperation from multiple banks. However, banks may not wish to reveal their view of the transaction graph to each other since it contains highly sensitive data. This motivates the need for designing a privacy-preserving protocol that allows computing a local cluster over the global transaction graph without requiring the banks to reveal their view of the transactions.

Consequent to the above discussion, our goal is to perform local clustering in a privacy-preserving manner. This will facilitate multiple data owners to perform local clustering on a graph that is held in a distributed fashion while ensuring that none of them is required to disclose their data in the clear. To achieve this, we rely on the technique of secure multiparty computation (MPC). Informally, MPC enables a set of parties to compute a function on their private inputs without leaking any information other than the output of the computation. In our setting, the private input is the graph (held distributedly) and a target seed node, while the output comprises a local cluster around the seed node. Since real-world graphs are massive, we focus on designing efficient solutions by relying on the optimal setting of 3-party computation (3PC) with honest-majority [27]. Further, we use the GraphSC paradigm [3, 44], to enable efficient, secure computations over graphs.

1.1 Our contributions

Our primary objective is to provide a privacy-preserving solution for local clustering, which is achieved for the first time. For this, we design a secure protocol for the state-of-the-art (cleartext) algorithm in [13]. This local clustering algorithm relies on heat kernel PageRank (HKPR) metric to quantify the similarity of nodes and thereby identify the cluster. Thus, a secure protocol for computing local clustering demands a secure protocol for computing HKPR values, which is also designed in our work. The state-of-the-art algorithm of [59] forms the basis for our secure HKPR protocol. We refer to §1.2 for justification of the choice of the cleartext algorithms.

To obtain as efficient a solution as possible, our protocols are designed in the GraphSC paradigm [3, 44] that provides a generic framework for evaluating message-passing graph algorithms securely while ensuring efficiency through parallel computations. At a high-level GraphSC paradigm works as follows. It takes as input a message-passing graph algorithm which involves updating data/state of the vertices in an iterative manner. Each round of the message-passing algorithm is translated into the GraphSC paradigm via the primitive operations of Scatter and Gather. Informally, Scatter involves *propagating* data computed at a vertex over its outgoing edges. On the other hand, Gather involves *aggregating* data present on the incoming edges into a vertex to update its state. These Scatter-Gather primitives, when performed on the appropriate (sorted) list representation of the graph, aid in obtaining a data-oblivious algorithm[1]. A secure realization of this algorithm via MPC eventually yields a secure realization of the graph algorithm. A discussion of the GraphSC paradigm is deferred to §2.3.

Given the above high-level idea, securely computing local clustering and HKPR metric via GraphSC entails the following steps:
① *cleartext → message-passing algorithm:* This involves identifying components of the relevant cleartext algorithm that can benefit from GraphSC and translating these to the message-passing paradigm. This is non-trivial to achieve since the graph algorithm under consideration may not lend itself as a message-passing one.
② *message-passing → data-oblivious algorithm:* This entails designing a data-oblivious variant of the message-passing algorithm by casting it in the GraphSC paradigm by defining the Scatter and Gather primitives. We emphasize that since GraphSC is a generic framework, it treats the operations within Scatter-Gather as a black-box (see §2.3) in the process of designing a data-oblivious solution. Hence, we define these specific to the considered graph algorithms. Note that our definitions of Scatter-Gather do not follow trivially and entail challenges as elaborated in §4.2, §5.2. Moreover, operations performed within Scatter-Gather should be defined carefully since these directly impact the efficiency of the resulting protocol.
③ *data-oblivious → secure protocol:* This involves designing secure protocols from the data-oblivious variant for which we rely on the state-of-the-art robust 3PC of SWIFT [27]. However, the framework of SWIFT lacks several essential primitives, such as division required for evaluating the local clustering algorithm in the GraphSC paradigm. We design these additional primitives, making SWIFT more comprehensive. Our detailed contributions appear below.

Secure local clustering. Naively translating the local clustering algorithm in [13] via the above steps results in a data-oblivious algorithm (and thereby a secure protocol) that has $O(|\mathsf{V}|(|\mathsf{V}| + |\mathsf{E}|))$ round complexity. This complexity results from the $|\mathsf{V}|$ iterations of the clustering algorithm, each of which requires computing a cluster-specific parameter (as described in §5.2) via Scatter-Gather where the latter has $O(|\mathsf{V}| + |\mathsf{E}|)$ complexity. Instead, we introduce a novel approach and design an algorithm (step ①) which can compute this cluster parameter incrementally, by reusing information from prior iterations without relying on Scatter-Gather. Elaborately, we augment the data associated with each vertex with a new component (to be computed via Scatter-Gather) that facilitates this incremental computation. Our reliance on a single call to Scatter-Gather to compute this new component, enables designing a significantly better data-oblivious algorithm having $O(|\mathsf{V}|+|\mathsf{E}|)$ round complexity (step ②). This complexity can further be improved to $O(\log(|\mathsf{V}| + |\mathsf{E}|))$ relying on the parallel variant of GraphSC described in [44] (recalled in §B.1). Our new message-passing algorithm (step ①) coupled with efficient definitions of Scatter-Gather for it (step ②), aid in obtaining a more efficient secure protocol for local clustering (step ③).

Secure HKPR. Although obtaining the message-passing algorithm (step ①) for computing HKPR is relatively simpler compared to the one for local clustering, we note that the resulting algorithm

[1]Data-oblivious algorithm is one whose control flow does not depend on input data.

requires to keep track of several parameters. Thus, keeping efficiency in mind, we define Scatter and Gather (step ②) such that we eliminate the overheads resulting from additional bookkeeping present in the message-passing variant. This results in obtaining an efficient realization of the secure protocol (step ③).

Note that our secure HKPR protocol can also be of independent interest since it finds use in other applications such as community detection [26], classification [40], etc. As in the case of the cleartext algorithm in [59], our secure protocol for computing HKPR is also versatile and allows computing several other graph propagation metrics such as L-hop transition probability, PageRank [47], personalized PageRank (PPR) [47], single-target PPR [33], Katz [22] measure, to name few. For further details, we refer to §4.4.

Enhancing the 3PC framework of SWIFT. The GraphSC framework [3, 44] treats the underlying MPC as a black-box, and hence does not require explicitly handling fixed-point arithmetic (FPA) operations. However, the algorithm of clustering requires operating over FPA and demands new primitives such as division, support for which is missing in SWIFT [27]. Hence, we design a secure protocol for division, which in turn requires designing secure protocols for conversion between arithmetic and Boolean representation and prefix OR. These primitives were originally missing in the framework of SWIFT. The addition of these makes SWIFT a more comprehensive framework. While SWIFT originally provided support for privacy-preserving machine learning (PPML) *inference* only (for neural networks), the inclusion of secure division protocol now also facilitates privacy-preserving *training* of neural networks. Further, since SWIFT was originally designed as an MPC framework for PPML, making it compliant with the GraphSC framework additionally requires a shuffle protocol. We rely on the shuffle protocol used in the GraphSC framework of [3] for the same. However, adapting this to work over SWIFT is non-trivial due to the difference in the sharing semantics of the two and the need for additional primitives.

We note that SWIFT and its enhanced version, as above, provide the strongest security of guaranteed output delivery (GOD). GOD ensures that the output of the computation is always delivered, irrespective of any adversarial misbehaviour. Thus, our designed secure protocols are also robust and inherit the security guarantee of SWIFT. Although this may be beneficial for several applications, it is worthwhile to note here that our protocols allow settling for the weaker security guarantee of fairness[2] or even security with abort[3], depending on the application scenario. Moreover, we note that the added security of GOD is achieved at no extra (amortized) cost over the weakest guarantee of abort security.

Benchmarks The designed secure protocols for clustering and HKPR operate on FPA, which has limited precision compared to its floating point counterpart. Further, probabilistic truncation and approximate division in the secure computation setting result in additional loss of accuracy. Hence, we perform extensive benchmarks to evaluate the accuracy loss of our secure clustering and HKPR protocols. Further, recall that the secure protocol for HKPR supports evaluating other graph propagation metrics. Hence, we also evaluate accuracy loss involved in securely realizing all these metrics. We observe that accuracy loss of our secure protocols is in the order of 10^{-5}, which is insignificant compared to their cleartext counterparts. We also benchmark the secure clustering and HKPR protocols over [27], and report the run time for the same. Our implementation also accounts for parallelization that can be achieved in the multiprocessor setting using the techniques described in [44]. The parallel variants witness improvements of up to 6.7× in the computation of HKPR as well as clustering when considering a graph of size 10^6. The reported numbers showcase the practicality of the designed protocols.

1.2 On the choice of cleartext algorithms

Keeping the quality of the output cluster in mind, we rely on the state-of-the-art works in the literature for computing the HKPR metric as well as for performing clustering using the same. However, one could question the performance of these algorithms when translated to their secure variants via MPC, i.e., do there exist other algorithms that can provide better performance in MPC by trading off the output quality? For this, we note that our solution comes at no extra cost. For example, consider randomized alternatives to state-of-the-art (in cleartext) that trade-off quality of the output to obtain a more efficient solution. The improved efficiency can be attributed to operating only on a subset of the nodes (sampled based on nodes that satisfy some condition) rather than considering the entire graph. However, when translating the same to a secure variant, it is required that the computations are input-independent. Thus, if the secure computation algorithm works on a subset of nodes that are sampled based on some condition, this will leak information about the number of nodes that satisfy the condition by simply observing which nodes are operated on. The number of such nodes may thereby leak information about the structure of the input graph, which should otherwise be kept private. Hence, to prevent such leakage, it is required that the secure algorithm operates on all nodes, albeit performing dummy operations on some nodes of the input graph (which ensures that computations are independent of the graph structure). Thus, the secure variants can no longer leverage the efficiency gains obtained by performing computations specific to a chosen subset of vertices. Hence, a secure variant of a graph algorithm is required to operate on the entire graph. This results in the asymptotic complexity of all alternatives being similar in the MPC domain. Hence, the designed secure protocols in no way trade-off output quality for efficiency.

With respect to the choice of [59] as the basis of our work, we note that the protocols in [59] outperform the prior protocols for HKPR-based clustering [13, 61] not only in terms of accuracy but also in terms of efficiency (see comparisons reported in [59]). Thus, [59] forms the state-of-the-art. We would further like to note that other solutions [13, 26, 61] are based on computing random walks, for which, to the best of our knowledge, there do not exist efficient MPC protocols. Naively performing the same would be highly expensive as it would require multiple scans of the edge-list (to ensure obliviousness) when identifying each node of the random walk. Further, random walks only constitute one component of the algorithm, and hence additional overhead will be incurred due to other algorithmic components. For real-world graphs with number of nodes and edges in the order of 10^6, this is highly inefficient. Hence, we conclude that the choice of our algorithm results in an efficient solution via MPC without compromising accuracy.

[2]This ensures that either all parties learn the output or none do.
[3]This may allow the corrupt party *alone* to learn the output.

1.3 Related work

Local clustering, as well as computing HKPR metric via MPC, has not been explored. Hence, we discuss the related works that consider cleartext computation of the same. Local clustering was initiated in the work of [55, 56], which identified clusters around a seed node by performing random walks over the graph. Since random walks starting from the seed node are more likely to visit nodes near the seed node, they help in identifying the local structure. Random walk-based method was improved in [1, 2] by relying on approximate PageRank vectors instead. When given a seed node, the PageRank vector provides a ranking of the nodes such that nodes that are more likely to be the endpoints of a random walk starting from the seed node get assigned a higher probability. Thus, the set of nodes with a higher rank constitutes a local cluster around the seed node. The state-of-the-art works on local clustering [13, 26] are based on heat kernel PageRank (HKPR) [12]. The advantage of HKPR over PageRank is that shorter random walks are more heavily weighted in HKPR, resulting in the walks being concentrated around the seed node. Hence, HKPR-based local clustering algorithms are known to provide a better quality cluster. Since [13] provides state-of-the-art solution in terms of cluster quality, we rely on the same while designing a secure protocol.

Computing the HKPR vector, as required for clustering, has been considered in a series of works [13, 26, 35, 59, 61] that aim to optimize the computation. Among these, [59] forms the state-of-the-art which outputs the most accurate HKPR vector compared to the prior works while also performing better in terms of efficiency. Since our goal is to identify the most accurate local cluster around a seed node, we rely on [59] for computing the HKPR vector.

Although we are the first to consider designing MPC protocols for local clustering as well as HKPR, several works in the literature use MPC, albeit for global clustering and graph propagation metrics other than HKPR. Despite these works addressing a different problem, we include a brief discussion on these in §A.

1.4 Organization of the paper

We start with preliminaries in §2 where we describe the system model, the MPC used, and the GraphSC paradigm. In §3, we describe the additional primitives designed over [27] as required for clustering. In §4 we describe the algorithm for securely computing the HKPR metric, followed by the secure clustering protocol in §5. These are followed by benchmarks for accuracy and performance in §6. Appendix §A discusses further related works, §B describes additional details of the GraphSC paradigm [3, 44] and §C provides a proof sketch for the security of the designed protocols,.

2 PRELIMINARIES

2.1 System model

MPC and threat model. We rely on secret-sharing based robust 3PC of [27]. We let $\mathcal{P} = \{P_1, P_2, P_3\}$ denote the three compute parties connected via pairwise private and authentic channels in a synchronous network. We assume a static, malicious adversary corrupting at most one party in $\mathcal{P}$.

Secure outsourced computation setting. We work in the secure outsourced computation setting where three (possibly hired) servers carry out the secure computation by emulating the three parties in 3PC. Any number of clients can send their inputs in a secret-shared manner to the servers. These servers hold secret-shared inputs of clients, on which they perform 3PC to obtain secret-shared output. Hence, our protocols assume that the input is already available in a shared (used interchangeably with secret-shared) format. The shared output is then reconstructed towards the intended client(s).

Preprocessing paradigm. Protocols are cast in the preprocessing model, where heavy, input-independent (yet function-dependent) computations are carried out in a preprocessing phase, followed by a fast input-dependent online phase. Here, the input comprises the graph represented as an adjacency list (as defined later). The preprocessing phase generates data for entries in the adjacency list, which is used to perform lightweight computations once the input graph structure is populated in this list (albeit in a secret-shared format) in the online phase.

Data representation. Computations are carried out over the algebraic ring structure, $\mathbb{Z}_{2^\ell}$ (arithmetic) or $\mathbb{Z}_2$ (Boolean). We use fixed-point arithmetic (FPA) to deal with decimal values. Here, a decimal value is represented as an ℓ-bit number in signed 2's complement notation, where the most significant bit (msb) is the sign bit, f least significant bits denote the fractional part, and the input is k bits long. We set $\ell = 64$, $\mathsf{k} = 32$ and $\mathsf{f} = 16$. Operations are performed on the ℓ-bit integer, treated as an element of $\mathbb{Z}_{2^\ell}$, modulo 2^ℓ. We use $(\mathsf{x})_\mathsf{f}$ to denote that x has f bit precision.

2.2 3PC of SWIFT

We first explain the 3PC sharing semantics of SWIFT [27] followed by its protocols that our work relies on.

Sharing semantics. SWIFT performs computation via masked evaluation and relies on a variant of replicated secret sharing (RSS) among 3 parties with threshold 1.

– A value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ is said to be RSS-shared (or $[\cdot]$-shared) among parties $\mathcal{P}$ if there exist $\mathsf{v}_1, \mathsf{v}_2, \mathsf{v}_3 \in \mathbb{Z}_{2^\ell}$ such that $\mathsf{v} = \mathsf{v}_1 + \mathsf{v}_2 + \mathsf{v}_3$, and P_1 holds $(\mathsf{v}_1, \mathsf{v}_2)$, P_2 holds $(\mathsf{v}_2, \mathsf{v}_3)$ and P_3 holds $(\mathsf{v}_3, \mathsf{v}_1)$.

– A value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ is said to be $[\![\cdot]\!]$-shared among parties in $\mathcal{P}$ if there exists $\alpha_\mathsf{v} \in \mathbb{Z}_{2^\ell}$ that is RSS-shared among $\mathcal{P}$, and $\beta_\mathsf{v} = \mathsf{v} + \alpha_\mathsf{v}$ is held by all parties in $\mathcal{P}$. That is, P_1 holds $(\alpha_{\mathsf{v}1}, \alpha_{\mathsf{v}2}, \beta_\mathsf{v})$, P_2 holds $(\alpha_{\mathsf{v}2}, \alpha_{\mathsf{v}3}, \beta_\mathsf{v})$ and P_3 holds $(\alpha_{\mathsf{v}3}, \alpha_{\mathsf{v}1}, \beta_\mathsf{v})$.

Sharing over $\mathbb{Z}_{2^\ell}$ is referred to as arithmetic sharing ($[\![\cdot]\!]$) while over $\mathbb{Z}_2$ is referred to as Boolean sharing ($[\![\cdot]\!]^\mathsf{B}$), where arithmetic operations (addition/subtraction) are replaced with Boolean XOR.

Protocols. The protocols we rely on from SWIFT, together with their description, are listed in Table 1.

2.3 GraphSC paradigm

Since real-world graphs are known to be sparse, naively using the adjacency matrix representation of the graph for computations would be expensive. Hence, designing an efficient solution to address the same involves- (i) designing an effective representation of the graph structure, (ii) ensuring the computation does not leak any private information, (iii) designing solutions that are highly parallelizable. The work by Nayak et al. [44] is the first to address the above problem and provides a framework for the same. The framework operates on a data augmented directed graph $\mathsf{G}(\mathsf{V}, \mathsf{E}, \mathsf{Data})$ which consists of a directed graph $\mathsf{G}(\mathsf{V}, \mathsf{E})$ where V is the set of vertices (or nodes, used interchangeably), E is the set of edges and

Building block	**Notation**	**Description**
Joint sharing	$[\![\mathsf{v}]\!] = \Pi_{\mathsf{Jsh}}(P_i, P_j, \mathsf{v})$	Enables $P_i, P_j \in \mathcal{P}$ to generate $[\![\mathsf{v}]\!]$ where $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ is held by P_i, P_j
Multiplication with truncation	$[\![\mathsf{z}]\!] = \Pi_{\mathsf{mult}}([\![\mathsf{x}]\!], [\![\mathsf{y}]\!], \mathsf{f})$	Multiplies x, y and outputs $\mathsf{z} = \mathsf{x} \cdot \mathsf{y}$ by truncated by f bits
Comparison	$[\![\mathsf{b}]\!]^{\mathbf{B}} = \Pi_{\mathsf{comp}}([\![\mathsf{x}]\!], [\![\mathsf{y}]\!])$	Outputs $\mathsf{b} = 1$ if $\mathsf{x} < \mathsf{y}$, else outputs $\mathsf{b} = 0$
Oblivious select	$[\![\mathsf{x_b}]\!] = \Pi_{\mathsf{sel}}([\![\mathsf{x}_0]\!], [\![\mathsf{x}_1]\!], [\![\mathsf{b}]\!]^{\mathbf{B}})$	Obliviously selects $\mathsf{x_b}$ among $\mathsf{x}_0, \mathsf{x}_1$
Bit2A	$[\![\mathsf{b}]\!] = \Pi_{\mathsf{bit2A}}([\![\mathsf{b}]\!]^{\mathbf{B}})$	Converts bit to its arithmetic equivalent
3-input multiplication	$[\![\mathsf{z}]\!]^{\mathbf{B}} = \Pi_{\mathsf{3\text{-}mult}}([\![\mathsf{a}]\!]^{\mathbf{B}}, [\![\mathsf{b}]\!]^{\mathbf{B}}, [\![\mathsf{c}]\!]^{\mathbf{B}})$	Multiplies (Boolean AND) 3 bits at once
4-input multiplication	$[\![\mathsf{z}]\!]^{\mathbf{B}} = \Pi_{\mathsf{4-mult}}([\![\mathsf{a}]\!]^{\mathbf{B}}, [\![\mathsf{b}]\!]^{\mathbf{B}}, [\![\mathsf{c}]\!]^{\mathbf{B}}, [\![\mathsf{d}]\!]^{\mathbf{B}})$	Multiplies (Boolean AND) 4 bits at once
Negation	$[\![\bar{\mathsf{b}}]\!]^{\mathbf{B}} = \Pi_{\mathsf{NOT}}([\![\mathsf{b}]\!]^{\mathbf{B}})$	Outputs $\bar{\mathsf{b}} = 1 \oplus \mathsf{b}$ where $\mathsf{b} \in \mathbb{Z}_2$

Table 1: Description of protocols from SWIFT [27].

Data is a set of user-defined data values associated with each vertex and edge of the graph. The data augmented graph is expressed as a list of vertices and edges where every vertex $\mathsf{v} \in \mathsf{V}$ is encoded as a tuple $(\mathsf{v}, \mathsf{v}, 1, \mathsf{data})$ and every edge $(\mathsf{u}, \mathsf{v}) \in \mathsf{E}$ is encoded as a tuple $(\mathsf{u}, \mathsf{v}, 0, \mathsf{data})$. The third entry in each tuple is a bit, isV, which equals 1 for a vertex and 0 otherwise, while data refers to the state information stored at each vertex and edge. These tuples constitute the data augmented graph list (DAG-list). The DAG-list representation of the graph is used to effectively represent the graph. Note that an undirected graph can be converted into a directed graph by accounting for each edge twice (incoming and outgoing edge). To perform secure computation over the graph while hiding its topology, each tuple in the data augmented graph is secret-shared entry-wise between the computing parties. This ensures that parties cannot distinguish between shares of a tuple corresponding to a vertex from that of an edge.

In principle, the framework of [44] enables securely evaluating message-passing graph algorithms. The latter are graph algorithms that operate in multiple rounds, where in each round, the nodes in the graph- (i) use their state information to send messages over their outgoing edges; (ii) receive messages along their incoming edges and aggregate these messages; (iii) use these messages to update their state. These three operations are abstracted into three primitives–Scatter, Gather, and Apply, respectively. Assuming that the graph algorithm can be expressed as a composition of the above primitives, [44] enables its secure evaluation via MPC [4]. Since designing an MPC protocol naively may leak information regarding the graph topology, the framework first designs a data-oblivious algorithm for each of these primitives, followed by a translation of the same using the generic 2-party protocol of [62]. In general, an algorithm is said to be data-oblivious if the instructions executed and the memory accesses made during the run of the algorithm are independent of the input and hence leak no information about the input. To obtain a data-oblivious algorithm for the GraphSC primitives, it is important to ensure that each entry in the DAG-list is visited when realizing these primitives to ensure no information about the association between the entries (such as an edge being incident on a node) is leaked. Observe that Apply can be computed obliviously by scanning through the DAG-list representation and applying the user-defined function if the element is a vertex and performing a dummy operation otherwise. To compute Scatter and Gather obliviously, [44] relies on two different sorted orders of the DAG-list representation. The *source order* requires the DAG-list to be sorted such that every node in the graph is placed before all edges that originate from it. The *destination order* requires the DAG-list to be sorted such that all edges that end at a particular node are placed before that node. Let $\Pi_{\mathsf{Sort}}([\![\vec{\mathsf{x}}]\!], key)$ denote an oblivious sort protocol that outputs a sorted list of elements in vector $\vec{\mathsf{x}}$ based on the key *key*. Given a data-oblivious sort protocol such as Bitonic sort [49], switching between the source order and destination order each time a Scatter or Gather is applied, ensures obliviousness as follows. Scatter can be accomplished obliviously by linearly scanning through the DAG-list sorted in the source order. For this, if the current tuple in the list is a node, the data value at the node is picked up, and if the current tuple is an edge, then the value picked up at the most recent tuple is used to update the edges. Gather can also be done obliviously by a linear scan through the DAG-list sorted in the destination order. During the scan, if the current tuple is an edge, its value is stored in an aggregate variable by applying an aggregation operation, and if the current tuple is a node, then the aggregate variable is stored along with the node. This approach of performing Scatter and Gather by performing a linear scan over a sorted order is oblivious as every node and edge of the graph is operated on, without revealing the relationship between the nodes and edges. Given that the primitives can be performed obliviously, as described above, the graph computation can be performed securely using MPC protocols. To summarize, message-passing graph algorithms can be computed obliviously by using the following steps in every message-passing round– (i) sort based on source order, (ii) Scatter, (iii) sort based on destination order, (iv) Gather and (v) Apply. An illustration of the operations involved in GraphSC paradigm of [44] is given in Fig. 9. Further, the framework in [44] provides a parallel algorithm for each of the individual primitives– Scatter, Gather, and Apply. In a multiprocessor setting, the parallel variants allow the computations to be performed in sub-linear complexity rather than the linear complexity of $O(|\mathsf{V}| + |\mathsf{E}|)$ in the size of the graph. We remark that this technique of obtaining a sub-linear solution in the multiprocessor setting also extends to our protocols. An overview of the sub-linear solution of [44] is provided in §B.1.

The work of [3] improves on the work of [44]. First, it combines Gather and Apply primitives such that both operations can be achieved in a single pass through the DAG-list. Moreover, [3] observes that the approach of [44] has the drawback of requiring an oblivious sort each time a Scatter or Gather primitive is applied.

[4]The operations within Scatter and Gather will vary across different graph algorithms. Hence, [44] provides a generic GraphSC framework, where the operations to be performed within Scatter and Gather are assumed as a black-box.

This amounts to two calls to an oblivious sort in every message-passing round. Instead, [3] observes that a secret shuffle followed by an insecure sort (which reveals the result of the comparisons) can be used to realize an oblivious sort. Let $\Pi_{\mathsf{Shuffle}}(\llbracket \vec{\mathsf{x}} \rrbracket)$ denote an oblivious shuffle protocol that outputs the elements of $\vec{\mathsf{x}}$ in a random shuffled order (see §B.2 for details of shuffle used in [3]). Observe that since the list is first shuffled, revealing the result of comparisons during sort does not break the obliviousness property. On the other hand, the insecure sort is required to be performed only once in the beginning, subsequent to which, the public permutation (obtained as output from the insecure sort) can be applied to sort the DAG-list, non-interactively. In summary, in the first message-passing round, a secret shuffle followed by an insecure sort (e.g., a comparison sort algorithm) is applied to get the required source or destination order. The permutations which map from the shuffled orders to the source and destination order are made public, as they do not reveal any information about the DAG-list. In the subsequent rounds, the secret shuffle, followed by the public permutation, is applied to get the required sorted order. Since shuffle can be performed much more efficiently than a sort, this change brings in significant efficiency improvements. An illustration of the operations involved in GraphSC paradigm of [3] is given in Fig. 9. Finally, to perform secure computation, [3] considers a 3PC setting to further enhance efficiency.

3 PRIMITIVES FOR CLUSTERING

Most of the primitives required for clustering can be obtained from SWIFT [27] (described in Table 1). However, clustering additionally requires other primitives, which we detail in this section. These primitives can be seen as making the SWIFT framework more comprehensive to support applications beyond PPML. The addition of these primitives also enables neural network training, which was not originally supported in SWIFT. Since these primitives are known from the literature [8, 28, 42, 48], we provide a high-level intuition of realizing them in SWIFT next.

Arithmetic to Boolean conversion. Given arithmetic shares of $\mathsf{x} \in \mathbb{Z}_{2^\ell}$, denoted by $\llbracket \mathsf{x} \rrbracket$, protocol Π_{A2B} enables generating its Boolean shares, $\llbracket \mathsf{x} \rrbracket^{\mathbf{B}}$, where each bit in $\mathsf{x} \in \mathbb{Z}_{2^\ell}$ is Boolean shared. Recall that any $\llbracket \cdot \rrbracket$-shared value x in SWIFT can be written as a sum of three values $\mathsf{x} = (\beta_{\mathsf{x}} - \alpha_{\mathsf{x}2}) - \alpha_{\mathsf{x}1} - \alpha_{\mathsf{x}3}$, where each summand is held by a pair of parties, i.e., P_1, P_3 hold $\mathsf{x}_1 = \alpha_{\mathsf{x}1}$, P_1, P_2 hold $\mathsf{x}_2 = \beta_{\mathsf{x}} - \alpha_{\mathsf{x}2}$, and P_2, P_3 hold $\mathsf{x}_3 = \alpha_{\mathsf{x}3}$. Each pair jointly generates Boolean shares of the summand that it holds, via the joint sharing protocol of SWIFT, resulting in generating $\llbracket \mathsf{x}_1 \rrbracket^{\mathbf{B}}, \llbracket \mathsf{x}_2 \rrbracket^{\mathbf{B}}, \llbracket \mathsf{x}_3 \rrbracket^{\mathbf{B}}$. The correct generation of these Boolean shares is guaranteed because each x_i for $i \in \{1, 2, 3\}$ is held by two parties, at most one of which can be corrupt. The presence of an honest party in each pair enforces correct behaviour. Following the approach of [42], parties then evaluate a full adder circuit (FA) with $\llbracket \mathsf{x}_1 \rrbracket^{\mathbf{B}}, \llbracket \mathsf{x}_2 \rrbracket^{\mathbf{B}}, \llbracket \mathsf{x}_3 \rrbracket^{\mathbf{B}}$ as input to obtain $\llbracket \cdot \rrbracket^{\mathbf{B}}$-shares of carry c and sum s as output, where x can be written as $\mathsf{x} = \mathsf{x}_1 + \mathsf{x}_2 + \mathsf{x}_3 = 2c + s$. Elaborately, c is the vector of carry bits generated when performing the bit-wise addition of $\mathsf{x}_1, \mathsf{x}_2, \mathsf{x}_3$ using the FA, and s is the vector of bits denoting its sum. Having generated $\llbracket \cdot \rrbracket^{\mathbf{B}}$-shares of c, s, parties evaluate the optimized parallel prefix adder circuit of [48] on $\llbracket \cdot \rrbracket^{\mathbf{B}}$-shares of $2c, s$ to obtain $\llbracket \cdot \rrbracket^{\mathbf{B}}$-shares of the sum $\mathsf{x} = 2c + s$. This generates $\llbracket \cdot \rrbracket^{\mathbf{B}}$-shares of x.

Boolean to arithmetic conversion. The protocol Π_{B2A} takes as input Boolean shares $\llbracket \mathsf{x} \rrbracket^{\mathbf{B}}$ of $\mathsf{x} \in \mathbb{Z}_{2^\ell}$, and generates its corresponding arithmetic shares $\llbracket \mathsf{x} \rrbracket$. Note that x can be written as $\mathsf{x} = \sum_{i=0}^{\ell-1} 2^i \mathsf{x}_i$, where x_i for $i \in \{0, \ldots, \ell - 1\}$ is the i^{th} bit of x. Thus, to obtain arithmetic shares of x, we invoke bit to arithmetic conversion protocol of SWIFT on each bit of x to generate its arithmetic share, and combine these shares of x_i to obtain $\llbracket \mathsf{x} \rrbracket$ using the above equation.

Secure Prefix OR. Given a sequence of Boolean shared bits, $\mathsf{x}_{\ell-1}, \ldots, \mathsf{x}_0$, protocol Π_{PreOr} outputs Boolean shares of a sequence of bits $\mathsf{y}_{\ell-1}, \ldots, \mathsf{y}_0$ such that $\mathsf{y}_i = \vee_{j=i}^{\ell-1} \mathsf{x}_j$, where $\vee$ denotes the OR operator and $i \in \{0, \ldots, \ell - 1\}$. An efficient realization of this primitive appears in the work of [8]. However, it exploits the property that every element has an inverse in the field algebraic structure. Since SWIFT works over the ring algebraic structure where all elements do not have inverses, we resort to a different approach. Observe that OR can be written as a combination of NOT and AND gates, where NOT can be performed non-interactively. Hence, we rely on a series of Boolean multiplications (AND), specifically multi-input multiplications [58], to compute the prefix OR. For example, the prefix OR of four $\llbracket \cdot \rrbracket^{\mathbf{B}}$-shared bits $\mathsf{x}_3, \mathsf{x}_2, \mathsf{x}_1, \mathsf{x}_0$ can be computed in a single round by computing the following in parallel.

– Compute $\llbracket \bar{\mathsf{x}_i} \rrbracket^{\mathbf{B}} = \Pi_{\mathsf{NOT}}(\llbracket \mathsf{x}_i \rrbracket^{\mathbf{B}})$ for $i \in \{0, \ldots, \ell - 1\}$
– Set $\llbracket \mathsf{y}_3 \rrbracket^{\mathbf{B}} = \llbracket \mathsf{x}_3 \rrbracket^{\mathbf{B}}$
– Set $\llbracket \mathsf{y}_2 \rrbracket^{\mathbf{B}} = \Pi_{\mathsf{NOT}}\left(\Pi_{\mathsf{mult}}(\llbracket \bar{\mathsf{x}_3} \rrbracket^{\mathbf{B}}, \llbracket \bar{\mathsf{x}_2} \rrbracket^{\mathbf{B}})\right)$
– Set $\llbracket \mathsf{y}_1 \rrbracket^{\mathbf{B}} = \Pi_{\mathsf{NOT}}\left(\Pi_{\mathsf{3\text{-}mult}}(\llbracket \bar{\mathsf{x}_3} \rrbracket^{\mathbf{B}}, \llbracket \bar{\mathsf{x}_2} \rrbracket^{\mathbf{B}}, \llbracket \bar{\mathsf{x}_1} \rrbracket^{\mathbf{B}})\right)$
– Set $\llbracket \mathsf{y}_0 \rrbracket^{\mathbf{B}} = \Pi_{\mathsf{NOT}}\left(\Pi_{\mathsf{4\text{-}mult}}(\llbracket \bar{\mathsf{x}_3} \rrbracket^{\mathbf{B}}, \llbracket \bar{\mathsf{x}_2} \rrbracket^{\mathbf{B}}, \llbracket \bar{\mathsf{x}_1} \rrbracket^{\mathbf{B}}, \llbracket \bar{\mathsf{x}_0} \rrbracket^{\mathbf{B}})\right)$

Proceeding along similar lines, prefix OR of more than four bits can be performed by computing the 2,3,4-input multiplication with respect to every consecutive group of four bits in each level and repeating this process in a tree-based approach to compute the result. Similarly, prefix OR of up to 16 bits can be computed in two rounds and up to 64 bits in three rounds.

```
Protocol $\Pi_{\mathsf{AppRec}}(\mathcal{P}, \llbracket \mathsf{b} \rrbracket)$

– $\alpha = (2.9142)_{\mathsf{k}-1}$
– $\llbracket \mathsf{b} \rrbracket^{\mathbf{B}} = \Pi_{\mathsf{A2B}}(\llbracket \mathsf{b} \rrbracket)$
– for $i = 1$ to $\mathsf{k} - 2$ do
    ○ $\llbracket \mathsf{b}'_i \rrbracket^{\mathbf{B}} = \llbracket \mathsf{b}_i \rrbracket^{\mathbf{B}} \oplus \llbracket \mathsf{b}_{\mathsf{k}-1} \rrbracket^{\mathbf{B}}$
– $\{\llbracket \mathsf{y}_i \rrbracket\}_{i=0}^{\mathsf{k}-2} = \Pi_{\mathsf{PreOr}}(\{\llbracket \mathsf{b}'_i \rrbracket^{\mathbf{B}}\}_{i=0}^{\mathsf{k}-2})$ and $\llbracket \mathsf{y}_{\mathsf{k}-1} \rrbracket^{\mathbf{B}} = \llbracket 0 \rrbracket^{\mathbf{B}}$
– for $i = \mathsf{k} - 2$ to 1 do: $\llbracket \mathsf{y}_i \rrbracket^{\mathbf{B}} = \llbracket \mathsf{y}_i \rrbracket^{\mathbf{B}} \oplus \llbracket \mathsf{y}_{i+1} \rrbracket^{\mathbf{B}}$
– $\llbracket \mathsf{v} \rrbracket = \Pi_{\mathsf{B2A}}\left(\{\llbracket \mathsf{y}_i \rrbracket^{\mathbf{B}}\}_{i=\mathsf{k}-1}^{0}\right)$ (in reverse order)
– $\llbracket \mathsf{z} \rrbracket^{\mathbf{B}} = \Pi_{\mathsf{NOT}}(\Pi_{\mathsf{mult}}(\llbracket \mathsf{b}_{\mathsf{k}-1} \rrbracket^{\mathbf{B}}, \llbracket \mathsf{y}_0 \rrbracket^{\mathbf{B}}))$
– $\llbracket \mathsf{w}' \rrbracket = \alpha(1 - \Pi_{\mathsf{bit2A}}(\llbracket \mathsf{b}_{\mathsf{k}-1} \rrbracket^{\mathbf{B}}) - 2\Pi_{\mathsf{mult}}(\llbracket \mathsf{b} \rrbracket, \llbracket \mathsf{v} \rrbracket, 0)$
– $\llbracket \mathsf{w} \rrbracket = \Pi_{\mathsf{mult}}(\llbracket \mathsf{v} \rrbracket, \llbracket \mathsf{w}' \rrbracket, 2(\mathsf{k} - \mathsf{f} - 1))$
– return $\llbracket \mathsf{w} \rrbracket, \llbracket \mathsf{z} \rrbracket^{\mathbf{B}}$
```

Figure 1: Protocol for computing initial approximation

Secure division. We rely on the Goldschmidt algorithm [38] for division and extend the techniques of [8] to SWIFT. To compute a/b, the approach is to normalize b to a value c in $[0.5, 1)$, and obtain an initial approximate reciprocal of b as $w_0 \approx 1/b = 2.9142 - 2c$. This term has relative error $\epsilon_0 = 1 - bw_0 < 1$ [9]. Letting $e_i = \epsilon_0^{2^i}$, the recurrence equation for approximating a/b is given by $d_1 = aw_0$, $d_{i+1} = d_i(1 + e_{i-1})$, $e_i = e^2_{i-1}$. After i iterations, $d_{i+1} = (a/b)(1 - \epsilon_0^{2^i}) \approx a/b$ with relative error $\epsilon_0^{2^i}$. Since division is non-trivial to extend to SWIFT, we detail the protocol for computing the initial approximation for $1/b$ in Fig. 1 and for computing a/b in Fig. 2. With $\mathsf{f} = 16$ bit precision and $k = 32$ bit inputs, we note that 4 iterations suffice for obtaining good accuracy in our applications while working over $\mathbb{Z}_{2^{64}}$. Correctness of division follows from [8].

Protocol $\Pi_{\mathsf{Div}}(\mathcal{P}, [\![\mathsf{a}]\!], [\![\mathsf{b}]\!])$

- $[\![\mathsf{w}]\!], [\![\mathsf{z}]\!]^{\mathsf{B}} = \Pi_{\mathsf{AppRec}}(\mathcal{P}, [\![\mathsf{b}]\!])$
- $\alpha = (1)_{2\mathsf{f}}$ and $[\![\mathsf{e}]\!] = \alpha - \Pi_{\mathsf{mult}}([\![\mathsf{b}]\!], [\![\mathsf{w}]\!], 0)$
- $[\![\mathsf{d}]\!] = \Pi_{\mathsf{mult}}([\![\mathsf{a}]\!], [\![\mathsf{w}]\!], 0)$
- for $i = 1$ to θ do
  - $[\![\mathsf{d}]\!] = \Pi_{\mathsf{mult}}([\![\mathsf{d}]\!], \alpha + [\![\mathsf{e}]\!], 2\mathsf{f})$
  - $[\![\mathsf{e}]\!] = \Pi_{\mathsf{mult}}([\![\mathsf{e}]\!], [\![\mathsf{e}]\!], 2\mathsf{f})$
- $[\![\mathsf{d}]\!] = \Pi_{\mathsf{mult}}([\![\mathsf{d}]\!], \alpha + [\![\mathsf{e}]\!], 2\mathsf{f})$
- return $[\![\mathsf{d}]\!]$

Figure 2: Protocol for division

Security of the designed protocols. Observe that the designed protocols rely on invoking protocols given in SWIFT [27] whose security was established therein in the standard real-world/ideal-world simulation paradigm. Hence, security of the designed protocols follows directly from security of underlying protocols of SWIFT.

4 PRIVACY-PRESERVING HKPR

To measure the similarity of nodes prior to performing local clustering, we rely on the graph propagation metric of heat kernel PageRank (HKPR). The HKPR metric is favourable since it is known to converge fast and produce good quality cluster [12, 13, 61]. Hence, we provide a secure local clustering algorithm that uses HKPR as the graph propagation metric (or the similarity measure), and can be accomplished in two steps. Given a seed node as input, we first compute the HKPR values of all vertices in the graph, assuming the seed node to be the source. The computed HKPR values are then used as input to a clustering algorithm to find a suitable cluster around the seed vertex. In this section, we describe how HKPR values can be generated securely, followed by describing the secure clustering algorithm in §5. Although the protocols are described as linear round solutions, they can easily be translated to a sub-linear round solution following the techniques of [44] (see §B.1). We begin by describing the cleartext algorithm for HKPR, followed by a secure protocol for its data-oblivious variant. Note that translation of cleartext to message-passing algorithm is accounted for implicitly by identifying (in place) the cleartext algorithm components that can benefit from computation via GraphSC.

4.1 The cleartext algorithm

For a seed node $s \in \mathsf{V}$, the HKPR value $\vec{\rho}[\mathsf{v}]$ of a node v, captures the probability with which a heat kernel random walk from s terminates at v. Thus, the graph propagation equation to compute HKPR of all nodes with respect to a specific seed node can be given as a $|\mathsf{V}|$-dimensional vector as :

$$\vec{\rho} = \sum_{i=0}^{\infty} \mathsf{w}_i \cdot (\mathsf{AD}^{-1})^i \cdot \vec{\mathsf{x}}, \tag{1}$$

where $\mathsf{w}_i = \frac{e^{-t}t^i}{i!}$ is the weight with t being the Poisson distribution parameter, A is the adjacency matrix of the input graph, D is degree matrix of the input graph with i^{th} diagonal entry storing degree of i^{th} vertex in the graph ($\deg_i$), and $\vec{\mathsf{x}}$ is the signal vector (of dimension $|\mathsf{V}|$), which is a one-hot encoding of s, that is, entry at position s is set to 1 (i.e., $\vec{\mathsf{x}}[s] = 1$) and the rest are set to 0.

We rely on the basic propagation algorithm in [59] that approximates $\vec{\rho}$ for each vertex by iteratively computing the sum in Eq. (1) only up to L terms. In fact, the algorithm in [59] is capable of computing various other graph propagation metrics such as PageRank, Personalized PageRank, L-hop transition probability, etc. Each of these metrics can be computed using the generalized graph propagation equation (Eq. (2)) and by appropriately parameterizing it. Further details regarding this are provided in §4.4. Steps for computing propagation vector $\vec{\rho}$ in Eq. (2) via graph propagation algorithm are described in Algorithm 1. The intuition is described next.

$$\vec{\rho} = \sum_{i=0}^{\infty} \mathsf{w}_i \cdot (\mathsf{D}^{-a}\mathsf{AD}^{-b})^i \cdot \vec{\mathsf{x}} \tag{2}$$

Algorithm 1: Graph propagation

```
Input: Undirected graph $G = (V, E)$, signal vector $\vec{x}$ of dimen-
-sion $|V|$, number of iterations L, parameters of graph propag-
-ation equation $a$, $b$, weights $\{w_j\}_{j=0}^{L}$, partial weights $\{Y_j\}_{j=0}^{L}$
Output: Estimated propagation vector $\vec{\rho}$ of dimension $|V|$
1  $\vec{r}^{(0)} = \vec{x}$, $\vec{\rho} = \vec{0}$ (the all zero vector);
2  for $i = 0$ to $L - 1$ do
3      for each $u \in V$ with non-zero $\vec{r}^{(i)}[u]$ do
4          for for each $v \in N_u$ do
5              $\vec{r}^{(i+1)}[v] = \vec{r}^{(i+1)}[v] + \left(\frac{Y_{i+1}}{Y_i}\right)\frac{\vec{r}^{(i)}[u]}{(\deg_v)^a(\deg_u)^b}$
6          end
7          $\vec{q}^{(i)}[u] = \vec{q}^{(i)}[u] + \left(\frac{w_i}{Y_i}\right)\vec{r}^{(i)}[u]$
8      end
9      $\vec{\rho} = \vec{\rho} + \vec{q}^{(i)}$
10 end
```

The algorithm requires that the weights add up to 1, and hence the weight entries in the generalized graph propagation equation (Eq. (2)) are normalised in such a way that $\mathsf{w}_i = \frac{\mathsf{w}_i}{\Sigma_{i=0}^{\infty}\mathsf{w}_i}$. The i^{th} iteration of the algorithm computes the i^{th} term in the infinite sum of $\vec{\rho}$. To compute this efficiently, the authors in [59] make the following observation that any two consecutive terms i and $i+1$ in the infinite sum in Eq. (2) have a lot of computation in common. To identify a recursive structure and avoid unnecessary re-computation, two vectors known as residue vector $\vec{\mathsf{r}}$ and reserve vector $\vec{\mathsf{q}}$ are defined as follows:

$$\vec{\mathsf{r}}^{\,(i)} = \mathsf{Y}_i \cdot (\mathsf{D}^{-a}\mathsf{AD}^{-b})^i \cdot \vec{\mathsf{x}}, \qquad \vec{\mathsf{q}}^{\,(i)} = \mathsf{w}_i \cdot (\mathsf{D}^{-a}\mathsf{AD}^{-b})^i \cdot \vec{\mathsf{x}}$$

Here, the partial weight sum, Y_i, is defined as $\mathsf{Y}_i = \Sigma_{k=i}^{\infty}\mathsf{w}_k$. Thus, it is clear that the $(i+1)^{\text{th}}$ residue vector can be derived from the i^{th} residue vector as given in Eq. (3). Similarly, the i^{th} reserve vector is nothing but the i^{th} term in the infinite sum of the graph propagation equation, and it can be expressed in terms of the i^{th} term of the residue vector as given in Eq. (3).

$$\vec{\mathsf{r}}^{\,(i+1)} = \frac{\mathsf{Y}_{i+1}}{\mathsf{Y}_i} \cdot \mathsf{D}^{-a}\mathsf{AD}^{-b} \cdot \vec{\mathsf{r}}^{\,(i)}, \qquad \vec{\mathsf{q}}^{\,(i)} = \frac{\mathsf{w}_i}{\mathsf{Y}_i} \cdot \vec{\mathsf{r}}^{\,(i)} \tag{3}$$

By using $\vec{r}^{(0)}$ and the relation between $\vec{r}^{\,(i)}$, $\vec{r}^{\,(i+1)}$, all residue vectors $\vec{r}^{(i)}$ for $i \in \{0, \dots, L\}$ can be computed. Each of the reserve vectors can also be obtained from the corresponding residue vectors, which can then be used to compute the graph propagation vector as $\vec{\rho} = \Sigma_{i=0}^{\infty}\vec{q}^{\,(i)}$. Given the above background, the steps of Algorithm 1 can be summarised as follows. Initially, residue vector $\vec{r}^{(0)} = Y_0 \cdot (D^{-a}AD^{-b})^0 \cdot \vec{x}$ is set as the signal vector $\vec{x}$ with the partial sum $Y_0 = \Sigma_{k=0}^{\infty} w_k = 1$. The algorithm repeats for L iterations where in each iteration i, the i^{th} term of the graph propagation sum is computed. Lines 4-6 in the algorithm compute the residue vector $\vec{r}^{\,(i+1)}$. Note that the term $(D^{-a}AD^{-b})$ in $\vec{r}^{\,(i+1)}$ is a square matrix with the (u, v) entry having the corresponding value of A weighted by a factor of $\frac{1}{(\deg_u)^a(\deg_v)^b}$. Hence, entries in A that are 0 will not contribute in the computation of $\vec{r}^{\,(i+1)}$. Thus, the algorithm makes use of the fact that the index v of the residue vector, $\vec{r}^{\,(i+1)}[v]$ is only updated by the residue entries of all it's neighbours u such that $v \in N_u$. Here N_u denotes the neighbours of node u. In this way, the authors in [59] design a vertex-centric algorithm that operates on vectors rather than matrices. In line 8, the reserve vector is derived from the residue vector, and in line 10 the reserve vector is used to update the graph propagation result vector $\vec{\rho}$. Note that, for the case of HKPR, the above computation is parameterized with $a = 0, b = 1$.

4.2 The data-oblivious variant

We begin by discussing the challenges that arise in naively using MPC to translate Algorithm 1 into a secure variant, followed by discussing the resolutions for the same. Consider the steps in Algorithm 1. For each node u, step 3 executes for only those nodes in the input graph that have a non-zero value in the corresponding component of the residue vector $\vec{r}[u]$. Hence, step 3 reveals whether a given node u has a non-zero value $\vec{r}[u]$. Further, steps 4-6 selectively update the residue vector components $\vec{r}[v]$ corresponding to each neighbour v of the current node u. Hence, the number of times the residue vector $\vec{r}$ is updated reveals the degree of the node u. Thus, the algorithm clearly does not qualify to be data-oblivious. Translating Algorithm 1 via MPC to obtain its secure variant by using secure protocols for operations used in Algorithm 1, will thus result in leaking the above-mentioned information. Hence, designing a secure variant of Algorithm 1 first requires designing a data-oblivious equivalent of the same. Since Algorithm 1 adheres to the message-passing paradigm, we obtain its data-oblivious equivalent by relying on the GraphSC paradigm. This requires defining the Scatter, Gather primitives specific to the considered algorithm, which we describe next.

To adhere to the GraphSC paradigm, the input graph is expressed using a DAG-list representation G (see §2.3). The i^{th} tuple in the list is denoted by $G[i]$. The data values associated with each tuple $G[i]$ in the list representation includes–(i) $G[i]$.dt to store messages that are sent across the edges, (ii) $G[i]$.deg to store the degree of a node, (iii) $G[i]$.r to store the residue vector component of a node, and (iv) $G[i].\rho$ to store the propagation vector component of each node (i.e., HKPR values). Recall that $G[i]$.isV denotes whether a tuple corresponds to a vertex or not. Since the graph propagation Algorithm 1 proceeds in an iterative manner, assuming the index of the current iteration to be j, the GraphSC primitives of Scatter and Gather are defined in Fig. 3. Running one iteration of Scatter followed by Gather accomplishes the same computation as in one iteration of Algorithm 1, albeit in a data-oblivious manner, as follows. Computing the graph propagation value $\vec{\rho}[v]$ for a node v requires the residue vector component $\vec{r}[u]$ corresponding to each of its neighbour u as well as $(\deg_u)^b$. Hence, the required information is made available on the edges via Scatter. Since the weights w_j and partial sums Y_j are public parameters, each node propagates $\frac{Y_{j+1}G[i].r}{Y_j(G[i].\deg)^b}$ across its outgoing edges in Scatter. During Gather, these values over incoming edges are aggregated in agg by summing them up. The residue vector component at vertex v is updated as $\frac{agg}{Y_j(G[v].\deg)^a}$. However, note that this generates the residue vector $\vec{r}^{\,(j+1)}$, required for the next iteration. Hence, prior to performing this update, the value stored in the residue vector $\vec{r}^{(j)}$ is used to update the graph propagation vector at vertex v as $G[v].\rho = G[v].\rho + \frac{w_j G[v].r^{(j)}}{Y_{j+1}}$. Our approach of designing Gather in this way does not require explicitly storing the reserve vector $\vec{q}$ as well as both, $\vec{r}^{\,(j)}$ and $\vec{r}^{\,(j+1)}$.

```
Scatter(G)
for $i = 1$ to $|V| + |E|$ do
  if $G[i]$.isV then
    val = $\left(\frac{Y_{j+1}}{Y_j}\right) \frac{G[i].r}{(G[i].\deg)^b}$
  else
    $G[i]$.dt = val

Gather(G)
for $i = 1$ to $|V| + |E|$ do
  if $G[i]$.isV then
    $G[i].\rho = G[i].\rho + \frac{w_j}{Y_j} G[i].r$
    $G[i].r = \frac{agg}{(G[i].\deg)^a}$
    agg = 0
  else
    agg = agg + $G[i]$.dt
```

Figure 3: Scatter and Gather for j^{th} iteration of Algorithm 1

4.3 The secure variant

We now describe the MPC protocol Π_{SGP} designed to securely evaluate the graph propagation algorithm Fig. 4. The protocol begins by assuming the required inputs G, $\vec{x}$, are already held in secret-shared fashion among the parties. It also takes as input the public parameters of Y_j, w_j for $j \in \{0, \dots, L-1\}$. Since the components of the residue vector $\vec{r}$ are initialized using the entries in the signal vector $\vec{x}$, unlike in the cleartext variant, $\vec{x}$ is required to be of the same size as the DAG-list G. That is, $\vec{x}$ must be secret shared as a vector of dimension $|V| + |E|$, with value 1 at the location of the source vertex and 0 at all other entries. As in the definition of Scatter in Fig. 3, val should be computed only at the tuples corresponding to nodes. Hence, Π_{SGP} computes val′ at each tuple. However, val is obliviously updated to store the correct value using Π_{sel} (see Table 1). Similarly, for Gather, aggregation is obliviously performed only at the tuples corresponding to edges using Π_{sel}. Further, note that multiplying values that are secret shared requires invoking the Π_{mult} protocol. Finally, note that an explicit call to division for computing $[\![\frac{1}{G[i].\deg}]\!]$ is avoided by ensuring this value is made available in shares when obtaining shares of G (and it is set to 0 when $G[i]$ represents an edge). Since public parameters $a, b \in \{0, 1\}$, no additional multiplications are required.

4.4 Other graph propagation metrics

The graph propagation Algorithm 1 is used to compute the graph propagation result vector $\vec{\rho}$ using the graph propagation equation $\vec{\rho}$

Graph propagation metric	Parameters			Graph propagation equation
	a	b	w_i	
L-hop transition probability	0	1	$w_i = 0\ (i \neq L), w_L = 1$	$\vec{\rho} = (AD^{-1})^L \cdot \vec{x}$
PageRank	0	1	$\alpha(1-\alpha)^i$	$\vec{\rho} = \Sigma_{i=0}^{\infty}\alpha(1-\alpha)^i \cdot (AD^{-1})^i \cdot \vec{x}$
Personalised PageRank	0	1	$\alpha(1-\alpha)^i$	$\vec{\rho} = \Sigma_{i=0}^{\infty}\alpha(1-\alpha)^i \cdot (AD^{-1})^i \cdot \vec{x}$
Single Target PageRank	1	0	$\alpha(1-\alpha)^i$	$\vec{\rho} = \Sigma_{i=0}^{\infty}\alpha(1-\alpha)^i \cdot (D^{-1}A)^i \cdot \vec{x}$
Heat Kernel PageRank	0	1	$\frac{e^{-t}t^i}{i!}$	$\vec{\rho} = \Sigma_{i=0}^{\infty}\frac{e^{-t}t^i}{i!} \cdot (AD^{-1})^i \cdot \vec{x}$
Katz	0	0	β^i	$\vec{\rho} = \Sigma_{i=0}^{\infty}\beta^i \cdot (A)^i \cdot \vec{x}$

Note that $\vec{x}$ denotes the signal vector, which is the one-hot encoding of the seed node. However, in the case of PageRank, $\vec{x}$ is set as the uniform probability distribution vector with each entry being $\frac{1}{|V|}$. Similarly, in the case of Personalized PageRank, it is set as the teleportation probability distribution vector corresponding to the seed node.

Table 2: Graph propagation metrics

$= \Sigma_{i=0}^{\infty} w_i \cdot (D^{-a}AD^{-b})^i \cdot \vec{x}$. Recall that this graph propagation equation can also be used to compute many other graph propagation metrics such as PageRank, personalised PageRank, Katz centrality score etc. These propagation metrics, as described in the work of [59], are listed in Table 2. Given the parameters of the graph propagation equation, such as a, b, weights w and the initial seed node captured in the signal vector $\vec{x}$, the graph propagation algorithm can be used to find the corresponding information regarding the nodes of the graph. Hence, using the secure protocol given in Fig. 4, several graph propagation metrics can be computed without leaking any private information.

```
Protocol $\Pi_{SGP}\left(\mathcal{P}, [\![G]\!], [\![x]\!], \{Y_j, w_j\}_{j=0}^{L}, a, b\right)$
– for $i = 1$ to $|V| + |E|$ do
    ∘ $[\![G[i].r]\!] = [\![x[i]]\!]$ , $[\![G[i].\rho]\!] = [\![0]\!]$
– for $j = 0$ to $L - 1$ do
    ∘ $[\![G]\!] = \Pi_{Shuffle}([\![G]\!])$ and apply public permutation to source sort edge list
    ∘ $[\![val]\!] = [\![0]\!]$
    Scatter(G)
    ∘ for $i = 1$ to $|V| + |E|$ do
        - $[\![val']\!] = \left(\frac{Y_{j+1}}{Y_j}\right) \cdot \Pi_{mult}\left([\![G[i].r]\!], [\![\frac{1}{(G[i].deg)^b}]\!], f\right)$
        - $[\![val]\!] = \Pi_{sel}([\![val]\!], [\![val']\!], [\![G[i].isV]\!]^B)$
        - $[\![G[i].dt]\!] = [\![val]\!]$
    ∘ $[\![G]\!] = \Pi_{Shuffle}([\![G]\!])$ and apply public permutation to destination sort edge list
    Gather(G)
    ∘ for $i = 1$ to $|V| + |E|$ do
        - $[\![G[i].\rho]\!] = [\![G[i].\rho]\!] + \left(\frac{w_j}{Y_j}\right)[\![G[i].r]\!]$
        - $[\![G[i].r]\!] = \Pi_{mult}\left([\![agg]\!], [\![\frac{1}{(G[i].deg)^a}]\!], f\right)$
        - $[\![agg]\!] = \Pi_{sel}([\![agg]\!] + [\![G[i].dt]\!], [\![0]\!], [\![G.isV]\!]^B)$
```

Figure 4: Protocol for secure HKPR computation

5 PRIVACY-PRESERVING CLUSTERING

Here, we describe the protocol for realizing local clustering in a privacy-preserving manner using the computed HKPR values.

5.1 The cleartext algorithm

We rely on the HKPR-based clustering algorithm described in [13] as it provides state-of-the-art results for local clustering. The steps for the same are provided in Algorithm 2, which describes an approximate algorithm to find a local cluster around a given seed node. To find a suitable local cluster, the algorithm uses heat kernel PageRank values of all vertices in the graph, computed with respect to the seed node. To assess the quality of the cluster, the algorithm uses *Cheeger ratio* as the metric. Cheeger ratio Φ_s with respect to a set of nodes S is defined as in Eq. (4) where, ∂ is the number of edges that cross from the set S to the set of remaining vertices in the graph $V\backslash S$, and volume vol_S of a set S is defined as the sum of degrees of all vertices present in the set S.

$$\Phi_s = \frac{\partial}{min\{vol_S, vol_{V\backslash S}\}} \tag{4}$$

Thus, a smaller Cheeger ratio signifies a better cluster, since it minimizes the number of edges that cross the cluster and maximizes the degree of nodes within the cluster. The clustering algorithm takes as input the target Cheeger ratio ϕ and the target cluster volume ς, with the constraint that ς must be less than or equal to $vol_G/4$. The algorithm identifies S_i as a suitable cluster if it satisfies the following condition:

$$\varsigma/2 \leq vol_{S_i} \leq 2\varsigma \text{ and } \Phi_{s_i} \leq \sqrt{8\phi} \tag{5}$$

The cluster S_i with the lowest Cheeger ratio among all suitable clusters, if found, is output by the algorithm. As clusters are identified based on HKPR, the algorithm also requires the HKPR values $\vec{\rho}$ supplied as input.

The algorithm begins by sorting the vertices in the graph in the descending order of $\frac{\vec{\rho}[v]}{deg_v}$, where deg_v is the degree of vertex v. This ensures that a higher priority is given to those vertices having high HKPR values and low degrees when forming the cluster. In lines 3-12, the algorithm performs a linear scan over the sorted vertices such that in each iteration i, the cluster S_i under consideration is the set of vertices $\bigcup_{j\leq i} v_j$. If the volume of this cluster is more than 2ς, then the protocol can break and consider no further clusters. Since each subsequent iteration includes more vertices in the considered cluster, the corresponding volume will continue to increase. Thus, the volume constraint will continue to fail, and hence the larger clusters can be discounted. We refer an interested reader to [13] for further details on Algorithm 2 and its correctness.

5.2 The data-oblivious variant

Algorithm 2 does not qualify as a data-oblivious algorithm since many of the steps are input-dependent. For example, step 1 requires sorting the vertices in the graph with respect to the ratio $\frac{\vec{\rho}[v]}{deg_v}$. The sorted order clearly is dependent on the structure of the graph G.

Algorithm 2: HKPR-based graph clustering

```
Input: Undirected graph G = (V, E),
  $\vec{\rho}$ : Graph propagation vector of dimension $|V|$,
  $\varsigma$ : Target cluster volume,
  $\phi$ : Target Cheeger ratio, u = Seed vertex
Output: S = Set of nodes that satisfies the constraints in
  equation 5 and have minimum Cheeger ratio
1  sort vertices of G with respect to $\vec{\rho}[v]/\mathrm{deg}_v$;
2  $\phi_{\min} = \infty$, S = Empty set;
3  for $i = 1$ to $|V|$ do
4      $S_i = \bigcup_{j \le i} v_j$ ;
5      $\Phi_{s_i}$ = Cheeger ratio of $S_i$ as per Eq. (4)
6      if $\mathrm{vol}_{S_i} > 2\varsigma$ then
7          Break;
8      else if $\varsigma/2 \le \mathrm{vol}_{S_i}$ and $\Phi_{s_i} \le \sqrt{8\phi}$ then
9          if $\Phi_{s_i} < \phi_{\min}$ then
10             $\phi_{\min} = \Phi_{s_i}$;  $S = S_i$;
11         end
12     end
13 end
14 if S is not an Empty set then Output S ;
15 else Output "No Cluster found";
```

Further, computing the Cheeger ratio in step 5 depends on the current cluster and the overall topology of G, as evident from equation 4. Thus, similar to the graph propagation algorithm, we first design a data-oblivious algorithm, followed by designing its secure variant. Given that the vertices are sorted with respect to the ratio $\frac{\vec{\rho}[v]}{\mathrm{deg}_v}$, note that each iteration of the clustering algorithm computes the Cheeger ratio specific to the considered set S_i. This requires computation of ∂, vol_{S_i}, $\mathrm{vol}_{V\setminus S_i}$, which can benefit from translation to the GraphSC paradigm. Taking Algorithm 2 as the input algorithm to GraphSC, computing vol_{S_i}, $\mathrm{vol}_{V\setminus S_i}$ can be performed in one linear scan given the degree of the nodes. With respect to computation of ∂, a naive approach via Scatter and Gather would require computing ∂ for each S_i, and thereby require a linear scan each time. Since performing Scatter and Gather has $O(|V| + |E|)$ complexity, computing ∂ for all vertices has a complexity of $O(|V|(|V| + |E|))$. This is highly inefficient. We overcome the inefficiency and avoid the need for multiple linear scans by carefully modifying the algorithm such that only a single scan over the DAG-list G suffices for computing ∂ for all vertices. The complexity of our resulting data-oblivious algorithm is thus drastically reduced to $O(|V| + |E|)$ from $O(|V|(|V| + |E|))$ of the naive approach. Recall that this linear complexity in $|V| + |E|$ can further be reduced to sub-linear following the technique of [44]. To achieve this, the data values associated with $G[i]$ (see §4.2), are augmented with a new component $G[i]$.greaterCount. This component is used to store the number of neighbors v of a given node that have a greater $\frac{\vec{\rho}[v]}{\mathrm{deg}_v}$ value than the current node. As will be described next, greaterCount is used to determine the number of edges that cross a cluster of vertices, and thereby compute ∂ in a single scan of the DAG-list. Thus, Scatter and Gather are defined to populate the greaterCount component.

We let FindCluster (Algorithm 3) denote the modified version of Algorithm 2 that relies on greaterCount to compute local clusters. We now discuss how FindCluster uses greaterCount to compute ∂, and then define the Scatter and Gather primitives to compute greaterCount. Assuming that greaterCount is computed, FindCluster proceeds to sort the DAG-list in descending order of $\frac{\vec{\rho}[v]}{\mathrm{deg}_v}$. This is followed by performing a scan of the DAG-list to identify the suitable cluster with the minimum Cheeger ratio. Each time a vertex v_i is encountered, it is added to the existing cluster, say S_{i-1}, to form a new cluster, say S_i. To compute the Cheeger ratio of the new cluster, we require the number of edges ∂ that cross the new cluster S_i. We observe that the computation of the same can be simplified by accounting for (i) ∂ of the previous cluster S_{i-1}, (ii) new cross edges (i.e., cross edges between v_i and $V \setminus S_{i-1}$), and (iii) previous cross edges that now lie within the new cluster (i.e., cross edges between S_{i-1} and v_i). Since greaterCount was defined to count the number of neighbors u of the vertex v_i that had a higher value of $\frac{\vec{\rho}[u]}{\mathrm{deg}_u}$, it implicitly tracks the number of neighbours of v_i that were part of the previous cluster S_{i-1}.

Algorithm 3: FindCluster(G)

```
Input: Undirected graph G = (V, E)
Output: S = Set of nodes that belong to the local cluster
1  Sort G in source order;
2  Perform Scatter as given in Fig. 5 ;
3  Sort G in destination order ;
4  Perform Gather as given in Fig. 5 ;
5  Sort G by $\vec{\rho}[v]/\mathrm{deg}_v$ ;
6  $\partial = 0$, $\mathrm{vol}_S = 0$, $\mathrm{vol}_{V\setminus S} = \sum_{v\in V} G[v].\mathrm{deg}$ $\phi_{\min} = \infty$, flag = 0 ;
7  for $i = 1$ to $|V| + |E|$ do
8      if $G[i]$.isV then
9          $\partial = \partial + G[i].\mathrm{deg} - 2 \cdot G[i].\mathrm{greaterCount}$ ;
10         $\mathrm{vol}_S = \mathrm{vol}_S + G[i].\mathrm{deg}$ ;
11         $\mathrm{vol}_{V\setminus S} = \mathrm{vol}_{V\setminus S} - G[i].\mathrm{deg}$ ;
12         $\Phi_s = \frac{\partial}{min(\mathrm{vol}_S, \mathrm{vol}_{V\setminus S})}$ ;
13         if $(\varsigma/2 \le \mathrm{vol}_S \le 2\varsigma)$ && $(\Phi_s \le \sqrt{8\phi})$ && $(\Phi_s < \phi_{\min})$ then
14             $\phi_{\min} = \Phi_s$;   flag = $i$ ;
15         end
16     end
17 end
18 if flag > 0 then Output the first flag elements of G ;
19 else Output "No Cluster found";
```

Thus, greaterCount keeps track of the number of type (iii) edges. Given that all the edges of v_i are either of type (ii) or type (iii), the number of edges of type (ii) can be computed as $G[v].\mathrm{deg} - G[v].\mathrm{greaterCount}$. Further, vol_{S_i} and $\mathrm{vol}_{V\setminus S_i}$ can be updated based on the degree $G[v_i].\mathrm{deg}$ of the vertex v_i being added to the cluster. Hence, the Cheeger ratio of the new cluster S_i can be calculated using the updated ∂ and vol_{S_i}, $\mathrm{vol}_{V\setminus S_i}$. Further, note that since $G[i].\rho$ component for an edge will be 0, when G is sorted in the descending order according to $\frac{\rho}{\mathrm{deg}}$, all the vertices will be placed before the edges. Hence, the flag variable is used to keep track of the number of vertices in the sorted list that constitute the largest cluster that satisfies the condition in Eq. (5).

We now define the Scatter and Gather primitives to compute greaterCount. During Scatter, the vertices scatter $\frac{\vec{\rho}[\mathrm{v}]}{\mathrm{deg_v}}$ across their outgoing edges. To compute greaterCount, Gather is defined to first perform a reverse scan of the DAG-list, followed by a forward scan. During the reverse scan, the data value at each edge (u, v) is updated to store the difference $\frac{\vec{\rho}[\mathrm{u}]}{\mathrm{deg_u}} - \frac{\vec{\rho}[\mathrm{v}]}{\mathrm{deg_v}}$ corresponding to its end points. In the forward scan, each vertex counts the number of incoming edges having data value (i.e., the above difference) greater than zero and stores the same in greaterCount. In this way, performing a Scatter followed by Gather allows every node v to count the number of neighboring vertices u with a greater $\frac{\vec{\rho}[\mathrm{u}]}{\mathrm{deg_u}}$ than $\frac{\vec{\rho}[\mathrm{v}]}{\mathrm{deg_v}}$. The Scatter and Gather primitives are defined in Fig. 5.

```
Scatter(G)
for $i$ = 1 to |V| + |E| do
    if G[$i$].isV then
        val = $\frac{G[i].\rho}{G[i].deg}$
    else
        G[i].dt = val
```

```
Gather(G)
agg = 0
for $i$ = |V| + |E| to 1 do
    if G[i].isV then
        val = $\frac{G[i].\rho}{G[i].deg}$
    else
        G[$i$].dt = G[$i$].dt − val

for $i$ = 1 to |V| + |E| do
    if G[i].isV then
        G[i].greaterCount = agg
        agg = 0
    else
        if G[i].dt > 0
            agg = agg + 1
```

Figure 5: Scatter and Gather to compute greaterCount

5.3 The secure variant

The secure protocol for computing local cluster is given in Fig. 6. The protocol takes as input secret shares of G, ρ, and the public target Cheeger ratio ϕ. In addition to the primitives required for securely computing HKPR metric, the current protocol mainly requires Π_{Comp} for securely evaluating the conditional statements that require comparison and Π_{Div} for securely computing the Cheeger ratio. The secure protocol for computing local cluster internally relies on the secure protocol to compute greaterCount via the Scatter, and Gather. The secure protocol for the latter appears in Fig. 7.

5.4 Security of the protocols

Observe that our secure protocols for HKPR computation and clustering invoke the underlying 3PC protocols of SWIFT and the newly designed protocols provided in §3. Since the designed 3PC protocols invoke the protocols of SWIFT (as described in §3), whose security has been established therein, the security of Π_{SGP} and $\Pi_{\mathsf{SClustering}}$ follows directly. While correctness and obliviousness of the designed protocols were given in place, an overview is provided in §C for completeness.

6 EXPERIMENTAL EVALUATION

In this section, we demonstrate the applicability of our protocols by benchmarking their performance. We benchmark the secure HKPR-based clustering protocol on synthetic graphs, and YouTube social network [41]. We additionally compare the accuracy of our secure protocols to their cleartext counterparts.

```
Protocol $\Pi_{\mathsf{SClustering}}(\mathcal{P}, \llbracket G \rrbracket, \llbracket \vec{\rho} \rrbracket, \phi)$
– $\Pi_{\mathsf{greaterCount}}(\mathcal{P}, \llbracket G \rrbracket, \llbracket \vec{\rho} \rrbracket)$
– $\llbracket G \rrbracket = \Pi_{\mathsf{Sort}}(G, \llbracket G.\vec{\rho} \rrbracket)$
– $\partial = \llbracket 0 \rrbracket, \llbracket vol_S \rrbracket = \llbracket 0 \rrbracket, \llbracket vol_{V \setminus S} \rrbracket = \sum_{v \in V} \llbracket G[v].deg \rrbracket$
– $\llbracket \phi_{min} \rrbracket = \llbracket 2^{32} \rrbracket$ and $\llbracket flag \rrbracket = \llbracket 0 \rrbracket$
– for $i$ = 1 to |V| + |E| do
    ∘ $\llbracket \partial \rrbracket = \llbracket \partial \rrbracket + \llbracket G[i].deg \rrbracket - 2 \cdot \llbracket G[i].greaterCount \rrbracket$
    ∘ $\llbracket vol_S \rrbracket = \llbracket vol_S \rrbracket + \llbracket G[i].deg \rrbracket$, $\llbracket vol_{G \setminus S} \rrbracket = \llbracket vol_{G \setminus S} \rrbracket - \llbracket G[i].deg \rrbracket$
    ∘ $\llbracket min \rrbracket^{B} = \Pi_{\mathsf{Comp}}(\llbracket vol_{G \setminus S} \rrbracket, \llbracket vol_S \rrbracket)$
    ∘ $\llbracket vol_{min} \rrbracket = \Pi_{\mathsf{sel}}(\llbracket vol_S \rrbracket, \llbracket vol_{G \setminus S} \rrbracket, \llbracket min \rrbracket^{B})$
    ∘ $\llbracket \Phi_s \rrbracket = \Pi_{\mathsf{Div}}(\llbracket \partial \rrbracket, \llbracket vol_{min} \rrbracket)$
    ∘ $\llbracket \theta_1 \rrbracket^{B} = 1 \oplus \Pi_{\mathsf{Comp}}(2\llbracket \varsigma \rrbracket, \llbracket vol_S \rrbracket)$, $\llbracket \theta_2 \rrbracket^{B} = 1 \oplus \Pi_{\mathsf{Comp}}(\llbracket vol_S \rrbracket, \llbracket \varsigma \rrbracket/2)$,
      $\llbracket \theta_3 \rrbracket^{B} = 1 \oplus \Pi_{\mathsf{Comp}}\left(\llbracket \sqrt{8\phi} \rrbracket, \llbracket \Phi_s \rrbracket\right)$
    ∘ $\llbracket min \rrbracket^{B} = \Pi_{\mathsf{Comp}}(\llbracket \Phi_s \rrbracket, \llbracket \phi_{min} \rrbracket)$
    ∘ $\llbracket \theta \rrbracket^{B} = \Pi_{\mathsf{4\text{-}mult}}(\llbracket \theta_1 \rrbracket^{B}, \llbracket \theta_2 \rrbracket^{B}, \llbracket \theta_3 \rrbracket^{B}, \llbracket min \rrbracket^{B})$
    ∘ $\llbracket \phi_{min} \rrbracket = \Pi_{\mathsf{sel}}(\llbracket \phi_{min} \rrbracket, \llbracket \Phi_s \rrbracket, \llbracket \theta \rrbracket^{B})$
      $\llbracket flag \rrbracket = \Pi_{\mathsf{sel}}(\llbracket flag \rrbracket, \llbracket i \rrbracket, \llbracket \theta \rrbracket^{B})$
– Reconstruct flag. If flag > 0, reconstruct and output the first flag elements of G
```

Figure 6: Protocol for secure clustering

```
Protocol $\Pi_{\mathsf{greaterCount}}(\mathcal{P}, \llbracket G \rrbracket, \llbracket \vec{\rho} \rrbracket)$
– Compute $\llbracket G \rrbracket = \Pi_{\mathsf{Shuffle}}(\llbracket G \rrbracket)$ and apply public permutation to source order DAG-list
Scatter(G)
– $\llbracket val \rrbracket = \llbracket 0 \rrbracket$
– for $i$ = 1 to |V| + |E| do
    ∘ $\llbracket G[i].\rho \rrbracket = \Pi_{\mathsf{mult}}(\llbracket G[i].\rho \rrbracket, \llbracket \frac{1}{G[i].deg} \rrbracket, f)$
    ∘ $\llbracket val \rrbracket = \Pi_{\mathsf{sel}}(\llbracket val \rrbracket, \llbracket G[i].\rho \rrbracket, \llbracket G[i].isV \rrbracket^{B})$
    ∘ $\llbracket G[i].dt \rrbracket = \llbracket val \rrbracket$
– Compute $\llbracket G \rrbracket = \Pi_{\mathsf{Shuffle}}(\llbracket G \rrbracket)$ and apply public permutation to destination order DAG-list
Gather(G)
– $\llbracket agg \rrbracket = \llbracket 0 \rrbracket$
– for $i$ = |V| + |E| to 1 do
    ∘ $\llbracket val \rrbracket = \Pi_{\mathsf{sel}}(\llbracket val \rrbracket, \llbracket G[i].\rho \rrbracket, \llbracket G.isV \rrbracket^{B})$
    ∘ $\llbracket G[i].dt \rrbracket = \llbracket G[i].dt \rrbracket - \llbracket val \rrbracket$
– for $i$ = 1 to |V| + |E| do
    ∘ $\llbracket G[i].greaterCount \rrbracket = \llbracket agg \rrbracket$
    ∘ $\llbracket agg \rrbracket = \Pi_{\mathsf{sel}}\left(\llbracket agg \rrbracket, 0, \llbracket G[i].isV \rrbracket^{B}\right)$
    ∘ $\llbracket c \rrbracket^{B} = \Pi_{\mathsf{Comp}}(0, \llbracket G[i].dt \rrbracket)$
    ∘ $\llbracket agg \rrbracket = \Pi_{\mathsf{sel}}(\llbracket agg \rrbracket, \llbracket agg \rrbracket + 1, \llbracket c \rrbracket^{B})$
```

Figure 7: Protocol for computing greaterCount

6.1 Accuracy results

Computing the HKPR metric, as well as the Cheeger ratio used in clustering demands operating on decimal values. Hence, the secure computation of these proceeds via fixed-point arithmetic (FPA).

Since numbers in FPA representation have limited precision, operating over FPA introduces errors in comparison to floating-point arithmetic since the latter allows for higher precision. This holds true even when performing the computations on cleartext. Further, the method of probabilistic truncation and approximation of division used for secure computation introduces additional errors over the cleartext FPA algorithm. To showcase that secure computation via FPA does not result in significant accuracy loss in comparison to computations performed on cleartext floating-point equivalent, we compare and report the accuracy of both. Further, to showcase the error introduced when moving from floating-point to fixed-point representation, we report the difference in the accuracy of the two when considering the cleartext algorithm. All the implementations are in python over a 64-bit ring and use NumPy[19] and NetworkX[18] libraries. Henceforth, the terms *float* and *fixed* denote the floating-point and fixed-point computation of the cleartext algorithm, respectively. Similarly, the term *secure* refers to the secure fixed-point computation. We report accuracy of graph propagation algorithm first, followed by that of clustering.
Datasets: We benchmark accuracy of algorithms on three different types of synthetic graphs. To showcase the accuracy loss when computing over large real-world networks, we also consider YouTube social network [41]. Table 3 summarises details of these graphs.

Model	|V|	d	p
Small world	100	5	0.1
	500	5	0.1
	800	5	0.1
	1000	5	0.1
Powerlaw cluster	100	5	0.1
	500	5	0.1
	800	5	0.1
Preferential Attachment	100	5	-
	500	5	-
	800	5	-
Youtube	1134890	-	-

|V| denotes number of vertices, d denotes number of neighbours each vertex is assigned, and p denotes probability of switching an edge for the case of a small world graph, or probability of forming a triangle for the case of powerlaw cluster.

Table 3: Graph datasets used for accuracy testing

6.1.1 Graph propagation metrics. Recall that our secure graph propagation protocol Π_{SGP} allows to compute various graph propagation metrics. Since accuracy varies with the propagation metric under consideration, we account for the following metrics– L-hop transition probability, PageRank, single target PageRank and HKPR. We use MaxError and L1 error to capture accuracy loss. Given two graph propagation vectors $\vec{\rho}_1$ and $\vec{\rho}_2$, the MaxError is computed as $\max_{v \in V}\left(\left|\frac{\vec{\rho}_1[v]}{\deg_v} - \frac{\vec{\rho}_2[v]}{\deg_v}\right|\right)$. The MaxError measures the maximum absolute error in $\vec{\rho}_2$ with respect to $\vec{\rho}_1$. Similarly, the L1 error is given by $L1 = \frac{1}{|V|}\sum_{v \in V}\left(\left|\frac{\vec{\rho}_1[v]}{\deg_v} - \frac{\vec{\rho}_2[v]}{\deg_v}\right|\right)$. The L1 error measures the average absolute error in $\vec{\rho}_2$ with respect to $\vec{\rho}_1$ over all vertices of the graph. We compute the MaxError and L1 error in the fixed-point cleartext as well as *secure* algorithms, each with respect to the floating-point cleartext algorithm. The errors are reported with respect to the YouTube graph in Table 4. As evident from results of Table 4, both the MaxError and L1 error is in the order of $\times 10^{-5}$ or smaller with respect to floating-point algorithm. This implies that the difference in accuracy is visible only after the 5th decimal digit, and thus, the loss is very small. Benchmarks on synthetic graphs yield errors in similar orders, and hence are not reported.

Similarity measures	MaxError		L1 Error	
	Fixed	Secure	Fixed	Secure
L-Hop transition	1.3674×10^{-5}	6.0872×10^{-5}	1.6719×10^{-6}	1.6719×10^{-6}
PageRank(PR)*	1.8223×10^{-08}	1.8622×10^{-8}	5.0661×10^{-7}	3.7420×10^{-7}
Single target PR	3.2904×10^{-5}	5.2424×10^{-5}	1.7061×10^{-7}	3.0636×10^{-7}
HKPR	1.3865×10^{-5}	2.2393×10^{-5}	0.7911×10^{-8}	1.0205×10^{-8}

* Precision is set to 28 bits when operating on fixed-point to accommodate small values of signal vector

Table 4: MaxError and L1 error comparison of cleartext FPA and secure FPA algorithms with cleartext floating-point algorithm for various graph propagation metrics

6.1.2 Clustering. We perform local clustering on the various graphs described in Table 3 and report the accuracy results. While reporting the accuracy of the secure local clustering algorithm, we also account for the accuracy loss incurred in securely computing the HKPR propagation vector ($\vec{\rho}$), which is fed as input to our secure clustering algorithm. Although Personalized PageRank has also been used to perform local clustering, it is known from the literature [13, 26, 61] that HKPR-based clustering outputs better quality clusters. Hence, we report accuracy results for clustering based on HKPR only. We take the Cheeger ratio (Φ_S) and intersection difference (dist) as parameters to analyze the quality of the clusters output by the clustering algorithm. The Cheeger ratio for a cluster S, as explained in section §5.1, is given by $\Phi_S = \frac{\partial}{min(\text{vol}_S, \text{vol}_{G\setminus S})}$. Note that a lower Cheeger ratio implies a cluster of higher quality. We report the Cheeger ratio of the cluster output by the cleartext algorithm (which operates with floating-point as well as fixed-point representation) and our secure algorithm (which operates over fixed-point representation) in Table 5. The intersection difference, dist, measures the similarity between two sets. Given two clusters S and S' sorted with respect to $\vec{\rho}[v]/\deg_v$, the intersection difference of S' with respect to S is given by $\text{dist}(S, S') = \frac{1}{n}\sum_{i=1}^{n}\frac{|(S_i \oplus S'_i)|}{2i}$. Here, S_i, S'_i are sets containing first i elements of S and S', respectively, and $S_i \oplus S'_i = (S_i \setminus S'_i) \bigcup (S'_i \setminus S_i)$. The values of set intersection difference lies between $[0, 1]$ where it is 0 for absolutely identical sets, and 1 for disjoint sets. We compute the dist of the cluster generated by our secure protocol (operating over fixed-point representation) with respect to the cluster generated by the floating-point cleartext algorithm. We also compute the dist of cluster generated by the fixed-point cleartext algorithm with respect to that generated by the floating-point cleartext algorithm to showcase the effect of moving from floating-point computation to fixed-point computation. These values are reported in Table 5. As evident from Table 5, the Cheeger ratio of the output clusters in the three variants of the clustering algorithm are small and almost similar. This indicates that the quality of the cluster output by the secure protocol is similar to that output by the cleartext algorithm. Further, the set intersection distance is also very close to 0. This too, implies the cluster output by the secure protocol is nearly identical to the cluster generated by the cleartext algorithm.

Model	$\|V\|$	ς	Minimum Cheeger ratio			Intersection Difference	
			Float	Fixed	Secure	Fixed	Secure
Small world	100	100	0.24223	0.24221	0.24221	0.01149	0.01942
	500	500	0.1737	0.17507	0.17466	0.07723	0.03248
	800	500	0.14965	0.14964	0.14964	0.02118	0.01312
	1000	500	0.17176	0.17224	0.17218	0.01866	0.07169
Powerlaw cluster	100	20	0.63963	0.63963	0.63963	0.03439	0.01587
	500	100	0.45454	0.45452	0.45452	0.02120	0.06819
	800	100	0.4923	0.49227	0.49227	0.01830	0.04332
Preferential Attachment	100	100	0.45915	0.45913	0.45913	0.01252	0.01234
	500	500	0.45213	0.45211	0.45211	0.01761	0.02174
	800	500	0.54759	0.54747	0.54757	0.01809	0.02302
YouTube	1134890	500	0.61556	0.61433	0.61433	0.01020	0.01281

Table 5: Cluster quality with respect to Cheeger ratio and intersection difference. $|V|$ denotes the number of vertices and ς denotes the target cluster volume. We set the target Cheeger ratio, ϕ, to 0.1 for all the graphs. The choice of parameters ς and ϕ follow from [13]

6.2 Secure computation

To show the practicality of our secure protocols, we benchmark and report their performance. We describe the benchmark environment and parameters first, followed by our observations. We report results in LAN (1 Gbps bandwidth) with 2.3 GHz Quad-Core Intel Core i7 machines having 16GB RAM. The average round trip time (rtt) for communicating 1KB data between a pair of machines is 0.29 milliseconds (ms). The protocols build on the ENCRYPTO library [14] in C++17 over a 64-bit ring. We use multi-threading, wherever possible, to facilitate efficient computation and communication among the parties. We report the time taken (in seconds) for the protocols to complete as the benchmark parameter.

Since the complexity of graph propagation and clustering algorithms depend on the size of the edge list (G), we analyze these algorithms by varying $|V|+|E|$ between 100 and 10^6. We benchmark the performance in a multiprocessor setting with four processors and report the maximum time taken by a processor. GraphSC provides a way to perform Scatter, Gather operations in parallel, which we use in the multiprocessor setting to ensure that our algorithms do not require linear complexity in $|V| + |E|$. To perform the sort operation in parallel, we use bitonic sort, which can be performed in parallel at the circuit level [49]. We use the secure shuffle protocol of [3]. However, it is not trivial to perform the shuffle in a parallel setting. Hence, the reported times for multiprocessor setting account for a non-optimized version of the secure shuffle.

For secure graph propagation, we estimate the run time for one iteration of the algorithm. Note that since all the graph propagation metrics can be computed via Π_{SGP}, the cost of evaluation of all these is the same. As expected and is evident from Table 6, the run time of the algorithm increases linearly with increasing $|V| + |E|$ in both single and multiprocessor settings. The parallelized algorithm gives 4× saving in run time in comparison to the single processor setting. The gain is even more eminent (up to 6.7×) for a higher number of processors. Fig. 8 gives a visual representation of variation in run time of graph propagation and clustering algorithm when the number of processors is varied.

We perform similar benchmarks for the secure clustering algorithm. Unlike the algorithm in [13], our algorithm takes the graph propagation vector ($\vec{\rho}$) as input, and hence, the run times reported do not account for the HKPR computation time. We report the results in Table 7 and observe the same trend as seen secure in HKPR. A visual representation of the variation in run time with increasing processors appears in Fig. 8.

$\|V\| + \|E\|$	Single Processor		Multi Processor	
	Preprocessing (s)	Online(s)	Preprocessing (s)	Online(s)
10^2	0.084	0.321	0.084	0.258
10^3	0.259	1.767	0.201	0.785
10^4	1.32	12.685	0.614	4.332
10^5	12.361	114.838	4.101	31.523
10^6	139.614	1161.475	41.149	295.073

Table 6: HKPR: Sequential computation and parallel computation (4 processors) for varying $|V| + |E|$

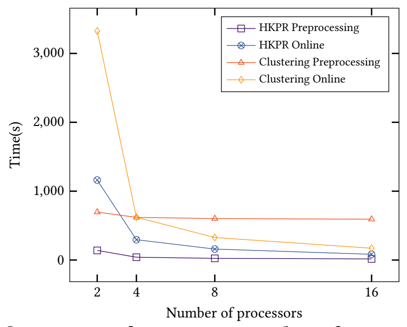


Figure 8: Run time for varying number of processors for $|V| + |E| = 10^6$

$\|V\| + \|E\|$	Single Processor		Multi Processor	
	Preprocessing (s)	Online(s)	Preprocessing (s)	Online(s)
10^2	0.109	0.446	0.14	0.3
10^3	0.761	3.634	0.777	1.131
10^4	6.928	33.805	6.4	7.65
10^5	68.716	331.812	61.925	64.101
10^6	695.913	3325.92	619.749	622.032

Table 7: Clustering: Sequential computation and parallel computation (4 processors) for varying $|V| + |E|$

7 CONCLUSION

We design secure protocols for local clustering and for computing HKPR graph propagation metric. To ensure that the designed protocols scale for large-sized graphs, we rely on the framework of GraphSC [3, 44] and the 3PC of SWIFT [27]. Through extensive experiments, we showcase that the accuracy loss of the secure protocols, in comparison to the cleartext counterparts, is very low. Further, the run time reported for the secure protocols showcases their practicality for real-world problems such as identifying a local cluster around an infected COVID-19 patient. In this way, relying on the state-of-the-art techniques for local clustering [13], HKPR metric [59], secure parallel graph computation framework [3, 44], and 3-party computation [27], facilitates achieving an efficient solution for privacy-preserving local clustering. Finally, our secure algorithm for computing HKPR also enables evaluating graph neural networks (GNN) [59]. Given that we also provide secure protocols for division, we believe that extending our techniques to provide support for GNN training and inference is an interesting question.

ACKNOWLEDGMENTS

Arpita Patra, Varsha Bhat Kukkala, Nishat Koti and Bhavish Raj Gopal would like to acknowledge financial support from National Security Council, India. Nishat Koti would also like to acknowledge support from Centre for Networked Intelligence (a Cisco CSR initiative) at the Indian Institute of Science, Bengaluru. The authors also acknowledge the support from Google Cloud for benchmarking.

REFERENCES

[1] Reid Andersen, Fan Chung, and Kevin Lang. 2006. Local graph partitioning using pagerank vectors. In *FOCS*.
[2] Reid Andersen and Yuval Peres. 2009. Finding sparse cuts locally using evolving sets. In *ACM STOC*.
[3] Toshinori Araki, Jun Furukawa, Kazuma Ohara, Benny Pinkas, Hanan Rosemarin, and Hikaru Tsuchida. 2021. Secure Graph Analysis at Scale. In *ACM SIGSAC*.
[4] Gilad Asharov, Francesco Bonchi, David García-Soriano, and Tamir Tassa. 2017. Secure centrality computation over multiple networks. In *WWW*.
[5] Yikun Ban and Jingrui He. 2021. Local clustering in contextual multi-armed bandits. In *The Web Conference*.
[6] Beyza Bozdemir, Sébastien Canard, Orhan Ermis, Helen Möllering, Melek Önen, and Thomas Schneider. 2021. Privacy-preserving density-based clustering. In *ACM CCS*.
[7] Paul Bunn and Rafail Ostrovsky. 2007. Secure two-party k-means clustering. In *ACM CCS*.
[8] Octavian Catrina. 2018. Round-Efficient Protocols for Secure Multiparty Fixed-Point Arithmetic. In *COMM*.
[9] Octavian Catrina and Amitabh Saxena. 2010. Secure computation with fixed-point numbers. In *FC*.
[10] Pak K Chan, Martine DF Schlag, and Jason Y Zien. 1994. Spectral k-way ratio-cut partitioning and clustering. *IEEE TCAD* (1994).
[11] Jung Hee Cheon, Duhyeong Kim, and Jai Hyun Park. 2019. Towards a practical cluster analysis over encrypted data. In *SAC*.
[12] Fan Chung. 2007. The heat kernel as the pagerank of a graph. *PNAS* (2007).
[13] Fan Chung and Olivia Simpson. 2018. Computing heat kernel pagerank and a local clustering algorithm. *European Journal of Combinatorics* (2018).
[14] Cryptography and Privacy Engineering Group at TU Darmstadt. 2018. ENCRYPTO Utils. https://github.com/encryptogroup/ENCRYPTO_utils.
[15] Priyanka Das and Asit Kumar Das. 2017. Behavioural analysis of crime against women using a graph based clustering approach. In *ICCCI*.
[16] Jennifer A Dunne, Richard J Williams, and Neo D Martinez. 2002. Food-web structure and network theory: the role of connectance and size. *PNAS* (2002).
[17] Shayan Oveis Gharan and Luca Trevisan. 2012. Approximating the Expansion Profile and Almost Optimal Local Graph Clustering. In *FOCS*.
[18] Aric Hagberg, Pieter Swart, and Daniel S Chult. 2008. *Exploring network structure, dynamics, and function using NetworkX.* Technical Report. LANL.
[19] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* (2020).
[20] Angela Jäschke and Frederik Armknecht. 2018. Unsupervised machine learning on encrypted data. In *SAC*.
[21] Ravi Kannan, Santosh Vempala, and Adrian Vetta. 2004. On clusterings: Good, bad and spectral. *JACM* (2004).
[22] Leo Katz. 1953. A new status index derived from sociometric analysis. *Psychometrika* (1953).
[23] Hannah Keller, Helen Möllering, Thomas Schneider, and Hossein Yalame. 2021. Balancing Quality and Efficiency in Private Clustering with Affinity Propagation. In *SECRYPT*.
[24] Hyeong-Jin Kim and Jae-Woo Chang. 2018. A privacy-preserving k-means clustering algorithm using secure comparison protocol and density-based center point selection. In *IEEE CLOUD*.
[25] Pan-Jun Kim, Dong-Yup Lee, and Hawoong Jeong. 2009. Centralized modularity of N-linked glycosylation pathways in mammalian cells. *PloS one* (2009).
[26] Kyle Kloster and David F Gleich. 2014. Heat kernel based community detection. In *ACM SIGKDD*.
[27] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. 2021. SWIFT: Super-fast and Robust Privacy-Preserving Machine Learning. In *USENIX Security*.
[28] Nishat Koti, Arpita Patra, Rahul Rachuri, and Ajith Suresh. 2022. Tetrad: Actively Secure 4PC for Secure Training and Inference. In *NDSS*.
[29] Varsha Bhat Kukkala and SRS Iyengar. 2020. Identifying influential spreaders in a social network (While preserving Privacy). *PETS* (2020).
[30] Yanhua Li, Zhi-Li Zhang, and Jie Bao. 2012. Mutual or unrequited love: Identifying stable clusters in social networks with uni-and bi-directional links. In *WAW*.
[31] Chung-Shou Liao, Kanghao Lu, Michael Baym, Rohit Singh, and Bonnie Berger. 2009. IsoRankN: spectral methods for global alignment of multiple protein networks. *BMC Bioinformatics* (2009).
[32] Frank Lin and William W Cohen. 2010. Power iteration clustering. In *ICML*.
[33] Peter Lofgren and Ashish Goel. 2013. Personalized pagerank to a target node. *preprint arXiv:1304.4658* (2013).
[34] László Lovász and Miklós Simonovits. 1990. The mixing rate of Markov chains, an isoperimetric inequality, and computing the volume. In *FOCS*.
[35] Zhenqi Lu, Johan Wahlström, and Arye Nehorai. 2021. Local clustering via approximate heat kernel PageRank with subgraph sampling. *Scientific Reports* (2021).
[36] David Lusseau, Karsten Schneider, Oliver J Boisseau, Patti Haase, Elisabeth Slooten, and Steve M Dawson. 2003. The bottlenose dolphin community of Doubtful Sound features a large proportion of long-lasting associations. *Behavioral Ecology and Sociobiology* (2003).
[37] Jun Ma, Danqing Zhang, Yun Wang, Yan Zhang, and Alexey Pozdnoukhov. 2018. GraphRAD: a graph-based risky account detection system. In *ACM SIGKDD*.
[38] Peter Markstein. 2004. Software division and square root using Goldschmidt's algorithms. In *RNC*.
[39] X Meng, D Papadopoulos, A Oprea, and N Triandopoulos. 2019. Private two-party cluster analysis made formal & scalable. *preprint arXiv:1904.04475 v2* (2019).
[40] Ekaterina Merkurjev, Andrea L Bertozzi, and Fan Chung. 2018. A semi-supervised heat kernel pagerank MBO algorithm for data classification. *Communications in Mathematical Sciences* (2018).
[41] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. 2007. Measurement and Analysis of Online Social Networks. In *IMC*.
[42] Payman Mohassel and Peter Rindal. 2018. ABY^3: A Mixed Protocol Framework for Machine Learning. In *ACM CCS*.
[43] Payman Mohassel, Mike Rosulek, and Ni Trieu. 2020. Practical privacy-preserving k-means clustering. *PETS* (2020).
[44] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. 2015. Graphsc: Parallel secure computation made easy. In *IEEE S&P*.
[45] Mark EJ Newman and Elizabeth A Leicht. 2007. Mixture models and exploratory analysis in networks. *PNAS* (2007).
[46] Andrew Ng, Michael Jordan, and Yair Weiss. 2001. On spectral clustering: Analysis and an algorithm. In *NeurIPS*.
[47] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report. Stanford InfoLab.
[48] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. 2021. ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation. In *USENIX Security*.
[49] Hagen Peters, Ole Schulz-Hildebrandt, and Norbert Luttenberger. 2011. Fast in-place, comparison-based sorting with CUDA: A study with bitonic sort. *Concurrency and Computation: Practice and Experience* (2011).
[50] Fang-Yu Rao, Bharath K Samanthula, Elisa Bertino, Xun Yi, and Dongxi Liu. 2015. Privacy-preserving and outsourced multi-user k-means clustering. In *IEEE CIC*.
[51] Corban G Rivera, Rachit Vakil, and Joel S Bader. 2010. NeMo: network module identification in Cytoscape. *BMC bioinformatics* (2010).
[52] Alex Sangers, Maran van Heesch, Thomas Attema, Thijs Veugen, Mark Wiggerman, Jan Veldsink, Oscar Bloemen, and Daniël Worm. 2019. Secure multiparty PageRank algorithm for collaborative fraud detection. In *FC*. Springer.
[53] Stephen B Seidman. 1983. Network structure and minimum degree. *Social networks* (1983).
[54] Jianbo Shi and Jitendra Malik. 2000. Normalized cuts and image segmentation. *IEEE PAMI* (2000).
[55] Daniel A Spielman and Shang-Hua Teng. 2004. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *ACM STOC*.
[56] Daniel A Spielman and Shang-Hua Teng. 2013. A local clustering algorithm for massive graphs and its application to nearly linear time graph partitioning. *SIAM Journal on computing* (2013).
[57] Steven H Strogatz. 2001. Exploring complex networks. *Nature* (2001).
[58] Ajith Suresh. 2021. MPCLeague: Robust MPC Platform for Privacy-Preserving Machine Learning. *CoRR* (2021). https://arxiv.org/abs/2112.13338
[59] Hanzhi Wang, Mingguo He, Zhewei Wei, Sibo Wang, Ye Yuan, Xiaoyong Du, and Ji-Rong Wen. 2021. Approximate Graph Propagation. In *ACM SIGKDD*.
[60] Yuchung J Wang and George Y Wong. 1987. Stochastic blockmodels for directed graphs. *J. Amer. Statist. Assoc.* (1987).
[61] Renchi Yang, Xiaokui Xiao, Zhewei Wei, Sourav S Bhowmick, Jun Zhao, and Rong-Hua Li. 2019. Efficient estimation of heat kernel pagerank for local clustering. In *ACM SIGMOD*.
[62] Andrew Chi-Chih Yao. 1982. Protocols for Secure Computations (Extended Abstract). In *FOCS*.

[63] Samee Zahur and David Evans. 2013. Circuit structures for improving efficiency of security and privacy tools. In *IEEE Symp. Secur. Priv.* IEEE.

[64] Jian-Xiong Zhang, Duan-Bing Chen, Qiang Dong, and Zhi-Dan Zhao. 2016. Identifying a set of influential spreaders in complex networks. *Scientific reports* (2016).

A RELATED WORK

Since many graph propagation metrics qualify as centrality measures, here we discuss works that securely compute other such measures such as PageRank, K-shell decomposition, followed by privacy-preserving global clustering algorithms.

Among the various measures, computing PageRank via MPC has been looked at [3, 4, 29, 44, 52], with several works in the 2PC setting. [44] securely computes PageRank using garbled circuits, while, [52] uses homomorphic encryption (HE) for the same. The work of [29] continues to be in the 2PC setting and focuses on ORAM-based implementation to securely compute PageRank. Additionally, [29] considers other centrality measures such as K-shell decomposition [53] and VoteRank [64]. [3] improves on the works of [44] and provides an efficient implementation of PageRank in the 3-party malicious setting. The work of [4] gives an n-party protocol for computing PageRank and Katz centrality in the semi-honest setting. Despite the popularity of the PageRank, due to it being a global network parameter, it is a misfit for use as a similarity measure in local clustering algorithms. Hence, we are the first to address the problem of securely computing HKPR, which is known to be the apt fit as a similarity measure for local clustering.

With respect to clustering, privacy-preserving variants of global clustering algorithms has been actively looked [6, 7, 11, 20, 23, 24, 39, 43, 50, 63]. There are several works [7, 20, 24, 43, 50] that realize privacy-preserving K-means clustering using HE in the two-party setting. However, these are not parallelizable and are highly inefficient. The work of [11] provides an efficient implementation of mean-shift clustering using HE that achieves linear complexity. [23] looks at the secure computation of affinity propagation-based clustering algorithms using arithmetic secret sharing. [63] performs DBSCAN clustering using garbled circuits in a 2 party setting. The work of [6] improves on [11] and gives an efficient implementation of DBSCAN for clustering using garbled circuits and arithmetic secret sharing in the two-party setting. The work of [39] describes hierarchical clustering using homomorphic encryption and garbled circuits in the two-party setting.

B GRAPHSC PARADIGM OF [3, 44]

The transformations involved in a GraphSC application and the improvements of [3] are depicted with the help of an example graph in Fig. 9. Although the framework of [44] requires an explicit "Apply" operation, we note that it can be performed along with "Gather" operation and hence, is not explicitly depicted in Fig. 9. Table 8 provides a list of most frequently used notations.

B.1 Parallel variant of GraphSC paradigm [44]

We provide an overview of the steps involved in translating the linear round sequential protocols for computing Scatter and Gather to attain a logarithmic round parallel protocol, in the presence of multiple processors, as described in the work of [44]. Assuming that there are $M = |\mathsf{V}| + |\mathsf{E}|$ processors, we begin by describing how Gather can be performed in parallel. The parallel variant of Scatter can also be achieved similarly. We conclude by describing the approach to perform Scatter-Gather in parallel when there are only $P < M$ processors. We refer an interested reader to [44] for further details.

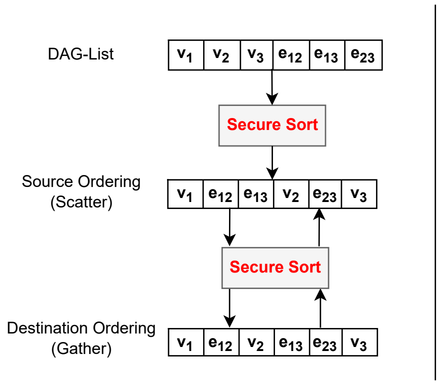
Notation	Description
V	Set of vertices
E	Set of edges
Data	Set of Data values
$\mathsf{G}(\mathsf{V}, \mathsf{E}, \mathsf{Data})$	Data augmented graph
DAG-list	List representation of data augmented graph
$\mathsf{G}[i]$	i^{th} entry/tuple of the DAG-list
$\mathsf{G}[i].\mathsf{isV}$	Denotes if tuple i corresponds to a vertex
$\mathsf{G}[i].\mathsf{dt}$	State information stored at tuple i
A	Adjacency Matrix
$\vec{\mathsf{z}}[\mathsf{v}]$	v^{th} component of a vector $\vec{\mathsf{z}}$
$(\mathsf{x})_{\mathsf{f}}$	Denotes that $\mathsf{x} \in \mathbb{Z}_{2^{\ell}}$ has f bits of precision

Table 8: Table of notations

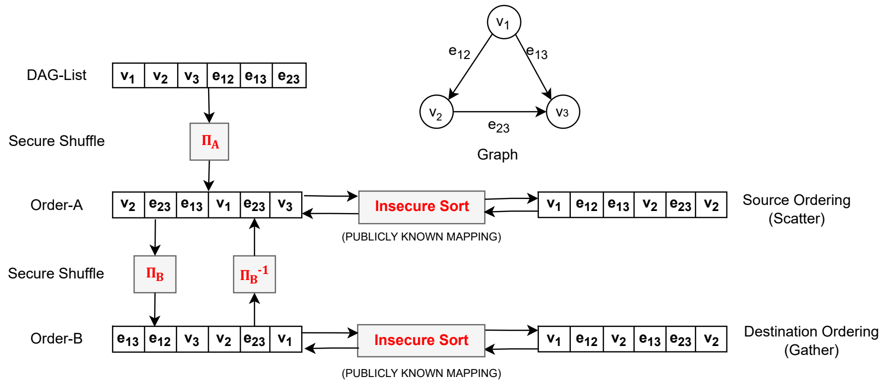
Performing Gather *in parallel.* Let $\ominus$ represent the aggregate operation to be performed within Gather. Recall that when considering the destination sorted list, the aggregate operation updates the data associated with every vertex with the data present in the longest consecutive sequence of edges preceding it. Let $\mathsf{LPS}[i, j)$ for $1 \leq i < j \leq |\mathsf{V}| + |\mathsf{E}|$ denote the "sum" (aggregation with respect to $\ominus$ operator) of the data present in the longest consecutive sequence of edges before j beginning from (and including) i. Observe that computing $\mathsf{LPS}[1, j)$ for $1 \leq j \leq |\mathsf{V}| + |\mathsf{E}|$ and updating a vertex j with $\mathsf{LPS}[1, j)$, is equivalent to updating the data at the vertices as done in Gather. Thus, to obtain the parallel variant of Gather given $M = |\mathsf{V}| + |\mathsf{E}|$ processors, we assign the task of computing $\mathsf{LPS}[1, j)$ to the j^{th} processor.

To attain the logarithmic round complexity, in each time step τ, the parallel algorithm computes LPS values for all segments of length 2^{τ}. The LPS values of the consecutive 2^{τ}-length segments computed in time step τ are then used to compute LPS values of $2^{\tau+1}$-length segments in the next time step, $\tau + 1$. In this way, since the input is a segment of length $|\mathsf{V}|+|\mathsf{E}|$, the computation of the last LPS entry, $\mathsf{LPS}[1, |\mathsf{V}|+|\mathsf{E}|)$ can thus be accomplished in $\log(|\mathsf{V}|+|\mathsf{E}|)$ time steps. Concretely, in time step τ, a processor $j \in \{1, 2, \ldots, |\mathsf{V}| + |\mathsf{E}|\}$ computes $\mathsf{LPS}[j - 2^{\tau}, j)$ (a τ-length segment). Thus, in the next time step, it can compute $\mathsf{LPS}[j - 2^{\tau+1}, j)$ by combining $\mathsf{LPS}[j - 2^{\tau+1}, j - 2^{\tau})$ and $\mathsf{LPS}[j - 2^{\tau}, j)$. A subtle thing to note here is that a segment can be aggregated with the immediately preceding segment of equal size *only if* a vertex has not been encountered in between. At the end of $\log(|\mathsf{V}| + \mathsf{E})$ time steps, vertex j's data for $j \in \{1, 2, \ldots, |\mathsf{V}| + |\mathsf{E}|\}$ is updated with $\mathsf{LPS}[1, j)$. The formal protocol for the same appears in Algorithm 4.

Performing Scatter *in parallel.* Recall that in the propagate operation performed as part of Scatter, each edge updates its data with the data of the nearest vertex preceding it. This propagate operation can also be performed in parallel, similar to as done in the aggregate operation. In fact, propagate can be represented as a special case of aggregate as follows. Initially, each edge either stores the value of the vertex if a vertex immediately precedes it, else it stores $-\infty$. Next,



(a) GraphSC paradigm of [44]



(b) GraphSC paradigm of [3]

Figure 9: Overview of operations involved in GraphSC

an aggregate operation can be performed where $\ominus$ is the max operator, i.e., $\mathsf{LPS}[j-2^{\tau+1}, j) = \max\{\mathsf{LPS}[j-2^{\tau+1}, j-2^{\tau}), \mathsf{LPS}[j-2^{\tau}, j)\}$. At the end of $\log(|\mathsf{V}| + \mathsf{E})$ time steps, the j^{th} processor computes $\mathsf{LPS}[1, j)$ which is the value of the nearest vertex preceding j. Thus, if j is an edge, its data can be updated with $\mathsf{LPS}[1, j)$.

Performing Gather *in parallel with $P < M$ processors.* In such a scenario, instead of holding a processor responsible for computing a single cell of LPS, it is assigned a consecutive range of cells to be computed. Without loss of generality, assume that M is a multiple of P. Let processor j be assigned the range $[s_j, t_j]$ where $s_j = (j-1)\cdot\frac{M}{P} + 1$ and $t_j = j\cdot\frac{M}{P}$. Each processor first computes $\mathsf{LPS}[s_j, t_j + 1)$ sequentially in $O(\frac{M}{P})$ time steps. Then, assuming that each $\mathsf{LPS}[s_j, t_j + 1)$ is a single value, the processors run the parallel algorithm for Gather on this segment of length P. Thus, the number of time steps required for this algorithm is $O\left(\frac{M}{P} + \log P\right)$.

Algorithm 4: Parallel Gather(G)

```
  Initialization: Every processor $j$ computes ;
1 $\mathsf{LPS}[j-1, j) = \begin{cases} G[j-1].\mathsf{dt}, & \text{if } G[j-1].\mathsf{isV} = 0 \\ 0, & \text{otherwise} \end{cases}$
2 $\mathsf{isV}[j-1, j) = \begin{cases} 1, & \text{if } G[j-1].\mathsf{isV} = 1 \\ 0, & \text{otherwise} \end{cases}$
  Main Algorithm: ;
3 For $\tau = 0$ to $\log(|\mathsf{V}| + |\mathsf{E}|) - 2$, each processor $j$ computes:
4     $\mathsf{LPS}[j-2^{\tau+1}, j) = \begin{cases} \mathsf{LPS}[j-2^{\tau+1}, j-2^{\tau}) \\ \quad \ominus \mathsf{LPS}[j-2^{\tau}, j), & \text{if } \mathsf{isV}[j-2^{\tau}, j) = 0 \\ \mathsf{LPS}[j-2^{\tau}, j), & \text{otherwise} \end{cases}$
5     $\mathsf{isV}[j-2^{\tau+1}, j) = \mathsf{isV}(j-2^{\tau+1}, j-2^{\tau}) \vee \mathsf{isV}[j-2^{\tau}, j)$
  end
```
*$\vee$ denotes the OR operation

B.2 Details of shuffle protocol of [3]

The shuffle protocol from the work of [3] uses two subroutines, a semi-honest shuffle pair $\Pi_{\mathsf{Shuffle-Pair}}$ and a robust maliciously secure set equality $\Pi_{\mathsf{Set-Equality}}$. It makes three invocations to $\Pi_{\mathsf{Shuffle-Pair}}$ where in each invocation it takes as inputs the RSS shares of a vector $\vec{v}$ to be shuffled, $[\vec{v}]$, and a set of two distinct parties who hold a common permutation, π (generated using a shared secret key). The output consists of RSS shares of $\pi(\vec{v})$. After every invocation of shuffle pair, a protocol $\Pi_{\mathsf{Set-Equality}}$ is called to verify the correctness of the shuffle pair, i.e., it checks whether the shares of the vector output by $\Pi_{\mathsf{Shuffle-Pair}}$ is a permutation of the shares of the vector that was input to this $\Pi_{\mathsf{Shuffle-Pair}}$. Thus, $\Pi_{\mathsf{Set-Equality}}$ takes in as input the shares of the vector that constitute the input and output to the $\Pi_{\mathsf{Shuffle-Pair}}$, and outputs a bit, indicating the correctness of $\Pi_{\mathsf{Shuffle-Pair}}$.

The described shuffle protocol, however, cannot be directly adapted to the framework of SWIFT, and there are two challenges that need to be addressed– (i) the shuffle of [3] uses RSS secret sharing semantics ($[\cdot]$-shares), and hence in order to adapt the protocol to SWIFT where values are $[\![\cdot]\!]$-shared, the vector which is shuffled needs to be converted from SWIFT sharing semantics ($[\![\cdot]\!]$-shares) to RSS sharing ($[\cdot]$-shares) and vice-versa, (ii) the shuffle of [3] operates entirely in Boolean worlds whereas our clustering protocol requires operating in the arithmetic world. To address the first challenge, we use the fact that conversion from $[\![\cdot]\!]$-shares to $[\cdot]$-shares can be performed non-interactively, relying on local operations. On the other hand, conversion from $[\cdot]$-shares to $[\![\cdot]\!]$-shares requires invoking the Π_{Jsh} primitive of SWIFT. To address the second challenge, we note that $\Pi_{\mathsf{Shuffle-Pair}}$ that operates in the Boolean world can be easily translated to work in the arithmetic world by replacing the XOR operations in the former with the corresponding addition/subtraction operations in the latter. A similar translation does not hold true for $\Pi_{\mathsf{Set-Equality}}$. Hence, to enable operating in the Boolean world while performing $\Pi_{\mathsf{Set-Equality}}$, we perform an arithmetic to Boolean conversion on the inputs of $\Pi_{\mathsf{Set-Equality}}$. Formal details of the shuffle protocol appear in Fig. 10.

Protocol $\Pi_{\mathsf{Shuffle}}(\llbracket \vec{\mathsf{v}} \rrbracket)$

1. Let $\vec{\alpha}_{\mathsf{v}1}, \vec{\alpha}_{\mathsf{v}2}, \vec{\alpha}_{\mathsf{v}3}, \vec{\beta}_{\mathsf{v}}$ denote the $\llbracket \cdot \rrbracket$-shares of $\vec{\mathsf{v}}$. Compute $[\vec{\mathsf{v}}]$ from $\llbracket \vec{\mathsf{v}} \rrbracket$ by setting $\vec{\mathsf{v}}_1 = \vec{\alpha}_{\mathsf{v}1}, \vec{\mathsf{v}}_2 = \vec{\alpha}_{\mathsf{v}2}$ and $\vec{\mathsf{v}}_3 = \vec{\alpha}_{\mathsf{v}3} - \vec{\beta}_{\mathsf{v}}$.
2. Compute Boolean shares of $\llbracket \vec{\mathsf{v}} \rrbracket$ as $\llbracket \vec{\mathsf{v}} \rrbracket^{\mathsf{B}} = \Pi_{\mathsf{A2B}} \llbracket \vec{\mathsf{v}} \rrbracket$. Generate $[\vec{\mathsf{v}}]^{\mathsf{B}}$ from $\llbracket \vec{\mathsf{v}} \rrbracket^{\mathsf{B}}$ as described above.
3. Compute $[\vec{\mathsf{v}}\,'] = \Pi_{\mathsf{Shuffle-Pair}}([\vec{\mathsf{v}}], P_0, P_1, \pi_{01})$ where $\vec{\mathsf{v}}\,' = \pi_{01}(\vec{\mathsf{v}})$ and P_0, P_1 hold π_{01} on clear.
4. Let $\vec{\mathsf{v}}_1', \vec{\mathsf{v}}_2', \vec{\mathsf{v}}_3'$ denote the three shares of $[\vec{\mathsf{v}}\,']$. Invoke Π_{Jsh} on each $\vec{\mathsf{v}}_i'$ to generate $\llbracket \vec{\mathsf{v}}_i' \rrbracket$. Compute $\llbracket \vec{\mathsf{v}}\,' \rrbracket = \llbracket \vec{\mathsf{v}}_1' \rrbracket + \llbracket \vec{\mathsf{v}}_2' \rrbracket + \llbracket \vec{\mathsf{v}}_3' \rrbracket$.
5. Compute Boolean shares of $\llbracket \vec{\mathsf{v}}\,' \rrbracket$ as $\llbracket \vec{\mathsf{v}}\,' \rrbracket^{\mathsf{B}} = \Pi_{\mathsf{A2B}} \llbracket \vec{\mathsf{v}}\,' \rrbracket$. Generate $[\vec{\mathsf{v}}\,']^{\mathsf{B}}$ from $\llbracket \vec{\mathsf{v}}\,' \rrbracket^{\mathsf{B}}$ as described above.
6. Check correctness of $\Pi_{\mathsf{Shuffle-Pair}}$, i.e., $\vec{\mathsf{v}}\,' = \pi_{01}(\vec{\mathsf{v}})$, by asserting $\Pi_{\mathsf{Set-Equality}}([\vec{\mathsf{v}}]^{\mathsf{B}}, [\vec{\mathsf{v}}\,']^{\mathsf{B}})$ is equal to 0.
7. Parties compute $[\vec{\mathsf{v}}\,''] = \Pi_{\mathsf{Shuffle-Pair}}([\vec{\mathsf{v}}\,'], P_1, P_2, \pi_{12})$ where $\vec{\mathsf{v}}\,'' = \pi_{12}(\vec{\mathsf{v}}\,')$ and P_1, P_2 hold π_{12} on clear.
8. Perform similar steps as described in steps 4, 5 to generate $[\vec{\mathsf{v}}\,'']^{\mathsf{B}}$.
9. Check correctness of $\Pi_{\mathsf{Shuffle-Pair}}$, i.e., $\vec{\mathsf{v}}\,'' = \pi_{12}(\vec{\mathsf{v}}\,')$, by asserting $\Pi_{\mathsf{Set-Equality}}([\vec{\mathsf{v}}\,']^{\mathsf{B}}, [\vec{\mathsf{v}}\,'']^{\mathsf{B}})$ is equal to 0.
10. Repeat steps 7, 8, 9 to robustly compute $[\vec{\mathsf{v}}\,'''] = [\pi_{02}(\vec{\mathsf{v}}\,'')]$ where P_0, P_2 hold π_{02} on clear.
11. Parties compute $\llbracket \vec{\mathsf{v}}\,''' \rrbracket$ from $[\vec{\mathsf{v}}\,''']$ via Π_{Jsh}, as described before.

Figure 10: Shuffle of [3] adapted over the framework of SWIFT

C SECURITY OF THE PROTOCOLS

In this section, we give an overview of the security of our protocols for secure graph propagation and secure local clustering that were described in §4 and §5 respectively. These protocols rely on the GraphSC paradigm for setting forth the algorithms in an oblivious fashion. These oblivious algorithms are then evaluated in a secure manner using the MPC protocols of SWIFT [27], together with the protocols described in §3. Since the protocols in §3 rely on invoking secure protocols of SWIFT, their security follows directly from the security of SWIFT. Thus, the security of our clustering and graph propagation protocols too follows from SWIFT.

Now, it remains to be shown that the algorithms described in sections §4.2 (oblivious graph propagation) and §5.2 (oblivious clustering) are data-oblivious. In §4.2, the graph propagation algorithm is given using the Scatter and Gather primitives of GraphSC. Observe that in each of these primitives, every entry in the entire DAG-list representation (including both nodes and edges) is accessed in each iteration. Further, the steps within these primitives can be made data-oblivious. Combined with the fact that Scatter and Gather are data-oblivious due to source and destination sort as provided by GraphSC, this ensures that the graph propagation algorithm in 4.2 is data-oblivious.

Regarding the algorithm in §5.2, the Scatter and Gather primitives are oblivious using the same arguments given as above. The only other function in §4.2 is FindCluster. FindCluster makes use of the number of neighbours of a vertex v, $\mathsf{u} \in \mathsf{N}_{\mathsf{v}}$, that have a greater value of $\frac{\vec{\rho}[\mathsf{u}]}{\mathsf{deg}_{\mathsf{u}}}$ than that of vertex v's $\frac{\vec{\rho}[\mathsf{v}]}{\mathsf{deg}_{\mathsf{v}}}$. This value is called G[v].greaterCount, and is computed during the Scatter-Gather step. Then, the function makes a single sweep over the list representation of the graph. If a vertex satisfying the required constraints is found, it is added to a set S. The correctness of FindCluster relies on correctly computing the Cheeger ratio Φ_s of set S. For this, the volumes of sets S, V \ S, denoted as $\mathsf{vol}_{\mathsf{S}}$ and $\mathsf{vol}_{\mathsf{V} \setminus \mathsf{S}}$, respectively, can be computed easily by adding or subtracting the degree of the vertex v being added to the set S. Next, ∂, which is the number of edges crossing the cluster S needs to be computed. A common global variable is used for computing ∂ by accumulating edges in it during a sweep over the DAG-list that is sorted according to $\frac{\vec{\rho}[i]}{\mathsf{deg}_i}$ for all G$[i]$ belonging to the DAG-list.

The edges incident on a vertex v that is yet to be added to the set S can be categorized as either (i) having one endpoint within S and the other endpoint outside S, or (ii) having both endpoints outside S. The number of edges at v of type (i) are G[v].greaterCount, while those of type (ii) are G[v].deg - G[v].greaterCount. Thus, the number of edges that cross the new cluster S ∪ v is updated by subtracting the number of edges of type (i) and adding the number of edges of (ii). Updating ∂ in this way ensures that the Cheeger ratio is computed correctly. This ensures the correctness of FindCluster. Observe that during the computation of FindCluster, all the edges and vertices of the graph are swept through once. Further, since every update happens depending on only whether the current entry in the DAG-list is a vertex or an edge, and not based on the structure of the graph, the sweep is oblivious. Finally, since the Scatter, Gather, and FindCluster is oblivious, this ensures that the entire algorithm is oblivious.