MultiCent: Secure and Scalable Computation of Centrality Measures on Multilayer Graphs

Andreas Brüggemann
Technical University of Darmstadt
Germany
brueggemann@encrypto.cs.tu-darmstadt.de

Nishat Koti
Aztec Labs
India
nishat@aztec-labs.com

Varsha Bhat Kukkala
IIT Tirupati
India
varshabhat@iittp.ac.in

Thomas Schneider
Technical University of Darmstadt
Germany
schneider@encrypto.cs.tu-darmstadt.de

Abstract

As real-world networks such as social networks and computer networks are often complex and distributed, modeling them as multilayer graphs is gaining popularity. For instance, when studying social interactions across platforms like LinkedIn, Facebook, TikTok, and Bluesky, users may be connected on several of these platforms. To identify important nodes/users, the platforms might wish to analyze user interactions using, e.g., centrality measures when accounting for connections across all platforms. This raises the challenge for platforms to perform such computation while simultaneously protecting their user data to shelter their own business as well as uphold data protection laws. Hence, it necessitates designing solutions that allow for performing secure computation on a multilayer graph which is distributed among mutually distrusting parties while keeping each party's data hidden.

The work of Asharov et al. (WWW'17) addresses this problem by designing secure solutions for centrality measures that involve computing the truncated Katz score and reach score on multilayer graphs. However, we identify several limitations in that work which render the solution inefficient or even unfeasible for realistic networks with significantly more than 10k nodes. We address these limitations by designing secure solutions that are significantly more efficient and scalable. In more detail, given that real-world graphs are known to be sparse, our solutions move away from an expensive matrix-based representation to a more efficient list-based representation. We design novel, secure, and efficient solutions for computing centrality measures and prove their correctness. Our solutions drastically reduce the asymptotic complexity from the prohibitive $O(|\mathsf{V}|^2)$ even for the fastest solution by Asharov et al. down to $O(|\mathsf{V}|\log|\mathsf{V}|)$, for $|\mathsf{V}|$ nodes. To design our solutions, we extend upon the secure graph computation framework of Koti et al. (CCS'24), providing a novel framework with improved capabilities in multiple directions. Finally, we provide an end-to-end implementation of our secure graph analysis framework and establish concrete efficiency improvements over prior work, observing several orders of magnitude improvement.

Proceedings on Privacy Enhancing Technologies 2025(4), 944–968

https://doi.org/10.56553/popets-2025-0166

Keywords

Secure multiparty computation, secure graph computation, multigraphs, centrality measures

1 Introduction

Many real-world scenarios require modeling different entities in the system while also capturing the relationships between them. A popular technique is to model the system as a graph, as done in the case of social networks, financial networks, computer networks, transportation networks, etc. However, a simple single-layered graph model often fails to capture the various kinds of complex relationships shared between the entities. To remedy this, many works rely on advanced modeling via multilayer graphs that consist of multiple layers, with each layer capturing a specific type of relation [15, 18, 42, 43, 59, 62, 66]. For instance, consider the scenario where the same set of users are connected to each other via multiple social networking platforms. In fact, the ease of logging into one platform and seamlessly switching to another using the same credentials greatly encourages cross-platform interactions. User interactions across various platforms can be collectively modeled as a *multilayer graph* where users are represented as a set of nodes V[1] while each platform i has its own set of edges E_i representing a layer of user interactions on the specific platform. Hence, multilayer graphs have significantly richer information than standalone graphs, making them a preferred choice for graph analysis.

A key tool for analyzing multilayer graphs is computing centrality measures to rank nodes [25, 26, 38, 65]. While centrality measures have been extensively studied for standalone graphs, their extension to multilayer graphs is relatively recent and has been applied in areas such as influence detection across multiple social media platforms [20], identification of key hubs for efficient routing across different transportation layers (e.g., road, rail, air) [32], or identifying individuals in multilayer human contact networks to model disease spread [61]. However, most of these works assume that the multilayer graph is available centrally. The challenge arises when data for each layer is held by different data owners. For example, in a multilayer graph formed by considering different social networking platforms, each platform considers its data private due

[1] Platforms may use several means of identifying shared nodes, e.g., using names, email addresses linked to accounts, connected accounts, or logins over third parties.

to commercial interests and privacy regulations.[2] However, the platforms may wish to identify influential spreaders across layers as required for information cascades. Hence, privacy-preserving solutions to analyze such multilayer graphs are needed.

A privacy-preserving approach for computing centrality measures on multilayer graphs entails computing the measures across all layers without data owners having to reveal information about their layers. For this, we use secure multiparty computation (MPC). MPC enables a set of parties to compute a function on their private inputs without leaking anything beyond the output. In our setting, the private input is each layer of the graph, while the output is a ranking of nodes based on a centrality score computed on the entire multilayer graph. While several works design MPC-based solutions to identify the influence of a given node [1, 36, 37], they focus on single-layer or simple graphs. In this work, we extend this to multilayer graphs.

To the best of our knowledge, [6] is the only prior work that designs MPC-based solutions for computing centrality measures on multilayer graphs. It identifies three key centrality measures that rank the nodes based on (i) Reach score π_{R}^{D}: number of nodes reachable from a given node within a radius of D, (ii) Truncated Katz score π_{K}^{D} [24]: number of paths of length at most D that originate at a given node, (iii) Multilayer truncated Katz score π_{M}^{D} [6]: similar to π_{K}^{D} but accounts for multiple edges between the same nodes. In general, the Katz centrality measure and its extensions have found a wide range of applicability, including problems such as link prediction [27, 40], anomalous link detection [53], and identifying influential nodes [41, 50, 54, 67]. While the metrics π_{R}^{D}, π_{K}^{D} are generic enough to be applicable for simple graphs, π_{M}^{D} is most relevant for multilayer graphs and hence the primary focus of our work. We identify and discuss several limitations of [6] next.

Graphs modeling real-world applications are often large. Despite acknowledging the importance of *scalability*, [6] relies on a matrix-based representation to store and process the underlying multigraph. This yields an inevitable overhead of $O(|\mathsf{V}|^2)$ when designing secure solutions via MPC. Furthermore, real-world networks are typically sparse [21], yet the solution in [6] does not take advantage of this property to improve efficiency. For this reason, several works, including those in the cleartext domain [8, 48], prefer list-based representation with complexity $O(|\mathsf{V}| + |\mathcal{E}|)$, where $\mathcal{E}$ is the set of all edges across layers. [6] propose to make π_{M}^{D} more scalable, but it reveals intermediate values, sacrificing privacy for better efficiency which we assess to be inadequate here. Our work addresses these limitations of [6] and provides the following contributions.

1.1 Our contributions

Conscious Design Choices. The limitations of [6] stem from its design choices that result in unnecessary communication and computation overhead. To improve efficiency and scalability, as required for real-world applications, we make the following design choices: (1) To leverage the sparsity of real-world networks, we operate on a list-based representation of multigraphs that improves communication, computation, and memory efficiency. (2) To make our solutions suitable for multiple lightweight data providers, we work in the secure outsourced computation (SOC) setting. Here, data and computations are securely outsourced to non-colluding servers that run the MPC protocol.[3] (3) To further improve efficiency, we rely on state-of-the-art MPC techniques and operate in the simplest setting of having two non-colluding semi-honest servers and a helper assisting with preprocessing.

Secure Centrality Measures. While [6] relies on a matrix-based representation, we use a list-based representation to improve scalability. While data-obliviousness (i.e., the algorithm's control flow is independent of the graph topology to not leak information) comes for free when operating on matrices, our solution is carefully designed to be data-oblivious despite operating on a list-based representation. To this end, we express our computation in a novel vertex-centric manner where computations are performed by nodes exchanging messages. These message-passing algorithms are then realized by building on top of and extending the state-of-the-art MPC-based graph analysis framework Graphiti [35]. This yields the first truly scalable computation of the considered centrality measures on multilayer graphs. For sparse graphs, moving to the list-based representation not only improves the overall computation complexity, but also reduces the communication complexity of data owners from $O(|\mathsf{V}|^2)$ in [6] to $O(|\mathsf{V}|)$. The improvements in computation of $\pi_{\mathsf{R}}^{D}, \pi_{\mathsf{K}}^{D}, \pi_{\mathsf{M}}^{D}$ on sparse graphs are as follows:

- $\pi_{\mathsf{K}}^{D}/\pi_{\mathsf{M}}^{D}$: For these measures, the score for all nodes can be computed at once with a complexity of only $O(|\mathsf{V}| \log |\mathsf{V}| + |\mathsf{V}| \cdot D)$ for radius D, compared to $O(|\mathsf{V}|^2 \cdot D)$ in [6].
- π_{R}^{D}: Although π_{R}^{D} is the simplest of all the measures, as in [6] it remains the most expensive to compute. Our protocol brings down the overall complexity from $O(|\mathsf{V}|^3)$ in [6] to $O(|\mathsf{V}|^2)$, which is a substantial improvement, but still has limited scalability compared to $\pi_{\mathsf{K}}^{D}/\pi_{\mathsf{M}}^{D}$.

Framework for Secure Message-Passing. We build a framework for secure message-passing that we use for computing centrality measures on multilayer graphs but which can also be used for other applications. Our framework is based on Graphiti [35], improving and extending upon Graphiti's capabilities. First, we describe how the data owners can perform *secure input sharing* involving efficiently secret sharing a list-based representation of a (multilayer) graph in complexity $O(|\mathsf{V}| + |\mathcal{E}|)$ as opposed to $O(|\mathsf{V}|^2)$ in [35], eliminating the last quadratic factor and hence boosting scalability. This also immediately generalizes to multilayer graphs or even more general multigraphs, whereas Graphiti only supports standard graphs. Furthermore, Graphiti focuses on optimizing certain operations in the message-passing iterations while requiring an expensive initialization phase in which we discover multiple problems restricting the protocol's applicability. We resolve these issues by carefully engineering a new initialization phase and accordingly reworking parts of the message-passing iterations, even improving the asymptotic complexity of Graphiti's original initialization.

Implementation and Benchmarks. We provide a full implementation of our new framework and make its code publicly available[4]. This is also of independent interest, especially given that

[2] https://techcrunch.com/2013/01/24/my-precious-social-graph, https://mashable.com/archive/twitter-instagram-find-friends

[3] The servers do not learn the private inputs of the data owners or any intermediate values during the computation since the data is secret shared among the servers.

[4] Full code available at https://encrypto.de/code/MultiCent.

Graphiti [35] only provides implementations of single building blocks of the message-passing iterations but no code for the initialization, and its predecessor [5] does not have any publicly available code. Based on our framework implementation, we also implement all our protocols and benchmark them against those in [6], showcasing substantial performance and scalability improvements. On a small graph with only 1000 nodes, our asymptotic improvement already makes us an order of magnitude faster than [6]. A direct comparison quickly becomes unfeasible as the severe memory footprint of [6] exceeds 100 GB already before reaching a graph size of 10k nodes. In contrast, our protocols can evaluate π_{M}^{D} for up to half a million nodes and 5 million edges within a few minutes, which would require terabytes of communication with [6].

1.2 Organization

In §2, we give the required notation, definitions for the considered centrality measures, and details regarding the setting, MPC, and the secure framework of Graphiti [35] as the foundation of our work. §3 proposes our vertex-centric computation of centrality measures utilizing message-passing. Our secure framework for message-passing is introduced in §4, and we discuss its improvements over the prior state of the art [35] in §5. After evaluating the performance of our solution in §6, we discuss related works in §7, and finally conclude our work in §8. In the appendix, we provide additional details on all used building blocks in §A, protocols in §B, and evaluation results in §C.

2 Preliminaries

We denote the ring of integers modulo 2^k by $\mathbb{Z}_{2^k}$. Vectors are written as $\vec{\mathbf{x}}$, where x_i corresponds to the i'th element of the vector, counting from 1. By $\vec{\mathbf{0}}$ and $\vec{\mathbf{1}}$ we denote vectors with all 0 respectively 1 entries. Furthermore, we write $\mathcal{S}_n$ for the symmetric group on $\{1, \dots, n\}$.

2.1 Graph Theory

We denote a directed graph by $\mathsf{G} = (\mathsf{V}, \mathsf{E})$ with nodes V and directed edges $\mathsf{E} \subseteq \{(v, w) \in \mathsf{V} \times \mathsf{V} \mid v \neq w\}$. We let $\mathsf{V} = \{0, \dots, |\mathsf{V}| - 1\}$. A multilayer graph, as defined in [6], consists of ℓ graphs $\mathsf{G}_1 = (\mathsf{V}, \mathsf{E}_1), \dots, \mathsf{G}_\ell = (\mathsf{V}, \mathsf{E}_\ell)$ sharing the same set of nodes. We simplify multilayer graphs to multigraphs where a multigraph is $\mathcal{G} = (\mathsf{V}, \mathcal{E})$ with $\mathcal{E}$ being a multiset of edges $(v, w) \in \mathsf{V} \times \mathsf{V}, v \neq w$, i.e., each edge is contained as many times as its multiplicity, and $\mathcal{E}$ is the (multiset) union over all E_i. For instance, if an edge (v, w) appears in E_1 and E_2, $\mathcal{E}$ contains this edge twice. Thus, $|\mathcal{E}| = \sum_{i=1}^{\ell} |\mathsf{E}_i|$. Finally, a multigraph $\mathcal{G} = (\mathsf{V}, \mathcal{E})$ can be transformed to a unified graph $\mathsf{G} = (\mathsf{V}, \mathsf{E})$ by discarding the multiplicity of all edges in $\mathcal{E}$, transforming the multiset to a standard set.

2.2 Centrality Measures

We begin by defining paths and path scenarios on multigraphs. A *path* from u to v is a sequence of nodes $(w_0 = u, w_1, \dots, w_{s-1}, w_s = v)$ such that $(w_{i-1}, w_i) \in \mathcal{E}$ for all $1 \leq i \leq s$, i.e., each edge exists in any of the layers; we call s the length of the path. For example, $(0, 1, 3, 1, 2)$ is a path of length 4 in Fig. 1. A *path scenario* [6] is a path as defined above together with a sequence of indices $1 \leq t_1, \dots, t_s \leq \ell$ such that $(w_{i-1}, w_i) \in \mathsf{E}_{t_i}$ for all $1 \leq i \leq s$, indicating for each edge a specific layer where it exists. In our prior example of path $(0, 1, 3, 1, 2)$, there are path scenarios using layers $(1, 3, 1, 2)$ as well as $(1, 3, 3, 2)$ due to edge $(3, 1)$ existing in layers E_1 and E_3.

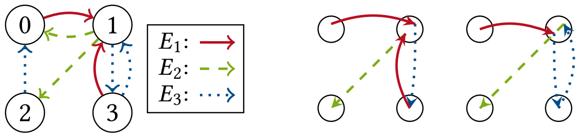


Figure 1: A multilayer graph with $\ell = 3$ layers (left) and two path scenarios on the graph, corresponding to the same path.

We next define the three centrality measures that are also considered in the prior work of [6][5]:

1. *Reach Score* $\pi_{\mathsf{R}}^{D}(v)$: This is the number of nodes reachable from v by a path of length $\leq D$. Formally stated:[6]
$$\pi_{\mathsf{R}}^{D}(v) = |\{w \in \mathsf{V} \mid \exists \text{path of length} \leq D \text{ from } v \text{ to } w\}| \qquad (1)$$
In Fig. 1, $\pi_{\mathsf{R}}^{2}(2) = 3$ as all nodes except for node 3 can be reached in ≤ 2 hops, starting from node 2.
2. *Truncated Katz Score* $\pi_{\mathsf{K}}^{D}(v)$ [24, 44, 64]: This is the number of paths of length $\leq D$, starting at node v. It is parametrized by $\beta_1 \geq \dots \geq \beta_D > 0$ where paths of length i are weighted by β_i.
$$\pi_{\mathsf{K}}^{D}(v) = \sum_{i=1}^{D} \beta_i \cdot (\#\text{paths from } v \text{ of length } i) \qquad (2)$$
In Fig. 1, $\pi_{\mathsf{K}}^{2}(1) = 33$ for $\beta_1 = 10, \beta_2 = 1$, as there are 3 outgoing edges and 3 paths of length 2 starting at node 1.
3. *Multilayer Truncated Katz Score* π_{M}^{D}: While $\pi_{\mathsf{R}}^{D}(v), \pi_{\mathsf{K}}^{D}(v)$ are of interest for standard and multilayer graphs, $\pi_{\mathsf{M}}^{D}(v)$ counts the number of path scenarios of length $\leq D$ on a multilayer graph, starting at v, applying weights as in $\pi_{\mathsf{K}}^{D}(v)$. By considering path scenarios, it gives higher weight to connections that exist in more layers simultaneously, making it a more sensitive choice for the multilayer case [6]. Formally:
$$\pi_{\mathsf{M}}^{D}(v) = \sum_{i=1}^{D} \beta_i \cdot (\#\text{path scenarios from } v \text{ of length } i) \qquad (3)$$
In Fig. 1, $\pi_{\mathsf{M}}^{2}(1) = 34$ for $\beta_1 = 10, \beta_2 = 1$, as in contrast to counting path $(1, 3, 1)$ once as in π_{K}^{2}, it now corresponds to two different path scenarios.

2.3 System and Threat Model

Among ℓ clients $\mathsf{C}_1, \dots, \mathsf{C}_\ell$, each C_i holds a private directed graph $\mathsf{G}_i = (\mathsf{V}, \mathsf{E}_i)$ on the same[7] set of nodes V, all graphs jointly defining the multilayer graph. Our computation is carried out by two non-colluding, semi-honest[8] servers (or parties) denoted as $\mathcal{P} = \{P_0, P_1\}$. The servers are connected via pairwise secure channels, for instance using TLS. Given the outsourced setting, the computations begin with the clients secret-sharing (introduced below) their private

[5] Note that [6] defines the measures immediately on matrices fitting the format of their data representation. Here, we keep it more general as we do not rely on matrices.

[6] We note that [6] has a redundant way of defining the metric by including weights for paths of different lengths, which is anyway canceled out by a clipping operation.

[7] In §4.1 we will discuss the case where clients do not hold the same set of nodes.

[8] They do not deviate from the intended computation but still try to derive information from all data they receive.

input graphs to the servers. The servers then run the secure computation on the resulting multilayer graph. While our secure protocols are generic, for efficiency reasons, they are cast in the preprocessing paradigm, where compute-intensive input-independent computation is pushed to a preprocessing phase to pave the way for a fast input-dependent online phase, ensuring low latency once the actual inputs are given. To facilitate fast preprocessing, as done in several works [13, 14, 55, 56, 63], we rely on a non-colluding semi-honest helper server $\mathcal{H}$ that aids in generating preprocessing data.

We assume that the total size of the graph (nodes + edges) and the number of nodes are known to all servers, identical to Graphiti [35]. Furthermore, clients need to disclose their individual input sizes to the servers.[9] Beyond that, clients and servers do not gain any information that is not explicitly specified as output to them.[10] In particular, any structural information about the graph remains hidden. We prove this security property using the real-world/ideal-world simulation paradigm.

2.4 Secure Multiparty Computation (MPC)

Our protocols use secure multiparty computation (MPC) to keep all provided data private. To represent private data, we utilize arithmetic secret sharing over $\mathbb{Z}_{2^k}$, where a client shares its secret input $x \in \mathbb{Z}_{2^k}$ by sending random values $x_0, x_1 \in \mathbb{Z}_{2^k}$ to the servers P_0, P_1, respectively, s.t. $x_0 + x_1 = x$, as frequently used, e.g., in [22]. Notation $[[x]]$ denotes sharing of x. Note that secret sharing easily generalizes to, e.g., vectors by sharing each entry. Furthermore, we generalize secret sharing to permutations $\rho \in \mathcal{S}_n$ by representing ρ as a vector $[[(\rho(1), \ldots, \rho(n))]]$, as done, e.g., in [7]. We also use binary sharings $[[b]]_{\text{bin}}$ for a single bit $b \in \{0, 1\}$, which is the special case of setting $k = 1$ in the previous definition. Note that for $k = 1$, addition corresponds to a logical XOR, while multiplication corresponds to an AND. A shared value can be reconstructed (also called opened) to a server/client by handing it both shares.

Linear operations on sharings can be executed without interaction and hence are considered cheap. To compute $[[x + y]]$ given $[[x]], [[y]]$, both servers add their corresponding shares $x_0 + y_0$ and $x_1 + y_1$. A public constant c can be multiplied to $[[x]]$ by both servers locally multiplying their shares with c. Given a constant c, it is also simple to initialize a sharing $[[c]]$ without interaction by setting $c_0 = c, c_1 = 0$, also allowing to add constants to shared values.

Multiplying two shared values is more expensive, requiring communication between the servers. To multiply two sharings $[[x]], [[y]]$, we use Beaver's multiplication [9], requiring each party to send 2 ring elements in the online phase, consuming a triple $([[a]], [[b]], [[c]])$ with random $a, b \in \mathbb{Z}_{2^k}$ and $c = ab$ computed during preprocessing. Assuming that the helper samples 2 PRF keys [51, 55], sending one to each party once, together with server P_i for $i \in \{0, 1\}$ it can non-interactively sample shares $a_i, b_i \in \mathbb{Z}_{2^k}$ to set up random $[[a]], [[b]]$. It can then non-interactively and together with P_0 sample random $c_0 \in \mathbb{Z}_{2^k}$ and send $c_1 = (a_0 + a_1)(b_0 + b_1) - c_0$ to P_1 to provide the triple. Hence, the preprocessing of one multiplication only requires sending a single ring element.

2.5 Graphiti [35]

To design efficient, secure solutions for graph algorithms via MPC, the line of works [5, 35, 49] operates on a list representation when dealing with sparse graphs. We identify the state-of-the-art Graphiti framework [35] which shares our setting with two servers and a helper and that we use as the basis of our extended framework. We proceed by providing an overview of the techniques in Graphiti that we adopt and refer to [35] for further details.

Graphiti takes a data-augmented graph (DAG) as input, denoted as $G(\mathsf{V}, \mathsf{E}, \mathsf{data})$ where data denotes a user-defined set of values associated with each node and edge. To store the graph topology, Graphiti uses the following three vectors of length $|\mathsf{V}| + |\mathsf{E}|$[11]— source ($\vec{\mathbf{src}}$), destination ($\vec{\mathbf{dst}}$), is_Vertex ($\vec{\mathbf{isV}}$)—, collectively referred to as the DAG list. The i-th entry in each of the vectors corresponds to a specific node or edge in G such that for a node v, $\mathrm{src}_i = v, \mathrm{dst}_i = v$, and $\mathrm{isV}_i = 1$, while for an edge $(v, w) \in \mathsf{E}$, $\mathrm{src}_i = v$, $\mathrm{dst}_i = w$, and $\mathrm{isV}_i = 0$. Additionally, Graphiti maintains a vector $\vec{\mathbf{payload}}$ of the same dimension to represent the data component of each node/edge and other data items that may be required to store intermediate information generated during computation. The data associated to the node/edge at index i is stored in $\mathrm{payload}_i$.

Graphiti allows evaluating message-passing graph algorithms consisting of multiple iterations, where each iteration performs:[12]

- **Propagate:** Nodes transmit their current data to all their outgoing edges,
- **Gather:** Nodes gather the data from their incoming edges, aggregating it to the sum over all edges, and
- **Apply Update:** Each node's prior data and aggregated (gathered) data are used to determine a new value data, customizable via a function $\mathcal{F}_{\text{APPLY}}$ taking both as inputs.

Graphiti efficiently and securely realizes message-passing by carefully and privately reordering the DAG list and applying cheap, non-interactive manipulations of $\vec{\mathbf{payload}}$. Each iteration starts with the DAG list in *vertex order*, where all entries corresponding to nodes 0 to $|\mathsf{V}| - 1$ appear sequentially in the same order, followed by entries corresponding to edges. Propagate is then executed in two stages where we first apply an operation $\mathcal{F}_{\text{PROPAGATE-1}}$ in the vertex order itself. The DAG list is then brought to *source order* where entries corresponding to a node appear immediately before all entries of its outgoing edges, before finally applying the second stage of propagate $\mathcal{F}_{\text{PROPAGATE-2}}$. Following this, to perform gather, the DAG list is brought into *destination order*, where all entries corresponding to incoming edges precede immediately before the destination node entry. Gather is also executed in two stages by first applying an operation $\mathcal{F}_{\text{GATHER-1}}$ in the destination order. The DAG list is then brought back into vertex order, where gather is finalized using operation $\mathcal{F}_{\text{GATHER-2}}$. Node data is then updated by executing $\mathcal{F}_{\text{APPLY}}$ on the prior and newly aggregated data per node. The details of building blocks $\mathcal{F}_{\text{PROPAGATE-1}}$, $\mathcal{F}_{\text{PROPAGATE-2}}$, $\mathcal{F}_{\text{GATHER-1}}$, and $\mathcal{F}_{\text{GATHER-2}}$ are provided in [35]. For our work, it is only required to know that

[9] Padding can be used to obfuscate the exact input sizes as discussed in Appendix B.4.

[10] If the result is made available to the clients or some third party only, the servers gain no further information at all.

[11] The discussed approach is different from Graphiti in terms of notation but is done to account for the optimizations that we introduce in §4 while achieving the same goal.

[12] Graphiti also supports other linear aggregations for gather as well as applying a customizable function to data in-between propagate and gather. Yet, for simplicity, we do not introduce these options here as they are not required in our application, but they could easily be added.

they are entirely linear and can hence be realized non-interactively within MPC. Details regarding the message-passing will be clarified later in §4 as a component of our system.

In the context of our work, we make the crucial observation that the DAG list representation naturally also supports multigraphs by adding the same edge (v, w) to the list at multiple indices i.

3 Vertex-Centric Computation of Centrality

In the following, we phrase computations of centrality measures as novel message-passing algorithms, which we later securely evaluate with our privacy-preserving framework in §4. Expressing the centrality measures through message-passing is what later enables scalable secure computation whereas using adjacency matrices as in [6] would inevitably not be scalable. We start with the multilayer truncated Katz score π_{M}^D, as this is most expressive on multilayer graphs and most scalable to compute. This is followed by the truncated Katz score π_{K}^D, which follows easily from π_{M}^D, and finally the reach score π_{R}^D. Our approaches allow computing these measures for all nodes in parallel, as in [6].

3.1 Multilayer Truncated Katz Score π_{M}^D

To compute π_{M}^D for each node via message-passing, we utilize a state s_v for each node v that is iteratively updated until it coincides with $\pi_{\mathsf{M}}^D(v)$. Initially, we assume that all weights β_i (§2.2) are set to 1 (which counts the unweighted number of paths). In the first iteration of message-passing, each node propagates a '1' on its incoming edges. Then, each node v gathers information from all its outgoing edges[13], sums it and writes it to state s_v^1. Thus, s_v^1 stores the number of edges starting at v, i.e., the number of path scenarios of length ≤ 1 starting at v.

We continue up to D iterations to capture all path scenarios with length $\leq D$. For this, let state s_v^{i-1} of iteration $i-1$ be the number of path scenarios starting at v of length $\leq i-1$. Observe that this holds true for s_v^1. For iteration i, we carefully observe the structure of path scenarios of length $\leq i$ starting at node v, as depicted in Fig. 2. Each edge (v, w) is a path scenario of length 1 starting at v. In addition, there are s_w^{i-1} path scenarios of length $\leq i-1$ starting at w (left of Fig. 2). Thus, prepending each (v, w) to existing path scenarios of length $\leq i-1$ starting at w gives path scenarios of total length $\leq i$ starting at v (right of Fig. 2). Hence,

$$s_v^i = \sum_{(v,w)\in\mathcal{E}} \left(s_w^{i-1} + 1\right) \ \forall v \in \mathsf{V}. \tag{4}$$

The above computation can be realized by ensuring that for each outgoing edge (v, w), v receives $s_w^{i-1} + 1$, where the +1 accounts for the path scenario that consists of only the single edge (v, w). Since there can be multiple (v, w) edges, the sum in Eq. (4) is over multiset $\mathcal{E}$, counting edges multiple times according to their multiplicity, which is as required for path scenarios.

We now consider the general case of arbitrary weights β_i, where β_i is the weight assigned to each path scenario of length i. In the previous case of $\beta_i = 1$, each iteration extends existing path scenarios by passing values s_w^{i-1} and starting new ones of length 1 from edges (v, w), captured by the +1 in Eq. (4). For the weighted case, we incorporate weights into this +1 by multiplying it by β_j for some j before the path scenario gets extended in further iterations. The choice of j is determined as follows: Recall from the case of uniform weights of 1 that each future iteration uses path scenarios of a certain length to form new scenarios that are one edge longer. Hence, edge (v, w) added in iteration i will be part of path scenarios of length 2 in iteration $i+1$ and so on, until being part of path scenarios of length $j = 1 + D - i$ in iteration D. Thus, in the weighted case, instead of adding a +1 to account for (v, w) in iteration i, we will add $+(1 \cdot \beta_j)$ where $j = 1 + D - i$. Overall, we let $s_v^0 = 0 \ \forall v \in \mathsf{V}$ and then for $1 \leq i \leq D$,

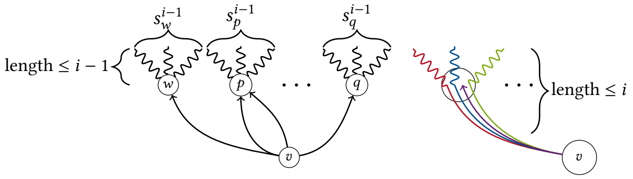


Figure 2: Inductively computing $\pi_{\mathsf{M}}^D(v)$, counting path scenarios up to length $\leq i$ by taking outgoing edges as a new path scenario, or by extending them by path scenarios of length $\leq i-1$ from the neighboring node. Given a multigraph, path scenarios can start with any instance of a specific edge, as in the case of (v, p) existing twice.

$$s_v^i = \sum_{(v,w)\in\mathcal{E}} \left(s_w^{i-1} + \beta_{1+D-i}\right) \ \forall v \in \mathsf{V}. \tag{5}$$

Hence, we obtain $\pi_{\mathsf{M}}^D(v) = s_v^D \ \forall v \in \mathsf{V}$ as formalized below.

Theorem 1. *Given that $s_v^0 = 0 \ \forall v \in \mathsf{V}$, applying update Eq. (5) to all nodes for iterations $i = 1, \dots, D$ yields that $\pi_{\mathsf{M}}^D(v) = s_v^D \ \forall v \in \mathsf{V}$.*

Proof. We prove that for $0 \leq i \leq D$, the following invariant holds:

$$s_v^i = \sum_{j=1}^{i} \beta_{D-i+j} \cdot (\#\text{path scenarios from } v \text{ of length } j) \tag{6}$$

Note that for $i = D$, it holds that $\pi_{\mathsf{M}}^D(v) = s_v^D$, concluding the proof.

We showcase that Eq. (5) satisfies the invariant from Eq. (6) via induction. For $i = 0$, the invariant trivially holds as per the initialization of all values. Now, assume that the invariant holds for an arbitrary but fixed $0 \leq i-1 < D$. All path scenarios starting at some $v \in \mathsf{V}$ first go over an edge $(v, w) \in \mathcal{E}$. They either have length 1, i.e., end immediately at w, or can be extended by a path scenario of length $\leq i-1$ starting at w. The number of such path scenarios is s_w^{i-1} for which Eq. (6) holds. Now, for path scenarios of

[13] While we usually consider propagate to outgoing edges and gather from incoming edges, we can use the opposite direction here, simply by swapping the interpretation of edge directions.

length i that start at v, we have

$$\begin{aligned}
&\sum_{j=1}^{i} \beta_{D-i+j} \cdot (\#\text{path scenarios from } v \text{ of length } j)\\
&= \beta_{D-i+1} \cdot (\#\text{path scenarios from } v \text{ of length } 1) +\\
&\sum_{(v,w)\in\mathcal{E}} \sum_{j=1}^{i-1} \beta_{D-(i-1)+j} \cdot (\#\text{path scenarios from } w \text{ of length } j)\\
&= \beta_{D-i+1} \cdot \left(\sum_{(v,w)\in\mathcal{E}} 1 \right) + \sum_{(v,w)\in\mathcal{E}} s_w^{i-1}\\
&= \sum_{(v,w)\in\mathcal{E}} \left(s_w^{i-1} + \beta_{1+D-i} \right) = s_v^i.
\end{aligned}$$

□

3.2 Truncated Katz Score π_{K}^{D}

As described in §2.2, the Katz score of a node v truncated at D counts the number of (weighted) paths, unlike path scenarios in π_{M}^{D}, from v of length at most D. Observe that computing π_{K}^{D} is similar to computing π_{M}^{D}. The only difference among the two measures is that π_{M}^{D} operates on multigraph $\mathcal{G} = (\mathsf{V}, \mathcal{E})$, while π_{K}^{D} operates on a unified graph $\mathsf{G} = (\mathsf{V}, \mathsf{E})$. E can be obtained from $\mathcal{E}$ by interpreting $\mathcal{E}$ as set E instead of a multiset (i.e., discarding the multiplicity of existing edges). As each path scenario on a simple G corresponds to just a path, we can thus reduce π_{K}^{D} to π_{M}^{D} by transforming $\mathcal{E}$ to a simple set E, followed by computing π_{M}^{D} on (V, E).

3.3 Reach Score π_{R}^{D}

As summarized in §2.2, the reach score counts the number of nodes that are reachable from v by a path of length *at most D* in the graph G. Computing the reach score in a vertex-centric manner on the DAG list representation of the graph can be realized by running a breadth-first-search (BFS) in a vertex-centric manner.

The iterative algorithm starts with v as the source node and searches (or counts) all nodes within distance D from v. Assuming s_w^i denotes the state of node w at iteration i, we wish to maintain the invariant that $s_w^i = 1$ for all nodes reachable by a path of length $\leq i$. To this end, we initialize $s_v^0 = 1$ while keeping all other values at zero, and then in each iteration i set

$$s_w^i = \mathrm{clip}\left(s_w^{i-1} + \sum_{(u,w)\in\mathcal{E}} s_u^{i-1} \right) \; \forall w \in \mathsf{V}, \tag{7}$$

where $\mathrm{clip}(x) = 1$ if $x > 0$ and 0 otherwise. Intuitively, $s_w^i = 1$ if w has already been reached in the previous iteration or if it can be reached in the current iteration through $(u, w) \in \mathcal{E}$ from a node u reached in the previous iteration. Realizing this in a vertex-centric manner is similarly done in Graphiti [35] and deferred to Appendix B.1. In contrast to the other centrality measures, we need to run $|\mathsf{V}|$ instances of the BFS in parallel to compute $\pi_{\mathsf{R}}^{D}(v)$ for all nodes $v \in \mathsf{V}$.

4 Secure Framework for Message-Passing

In our complete secure system, the clients first secret-share their inputs as DAG list among the servers (§4.1). The evaluation phase (§4.2) then securely computes the centrality measures via the message-passing algorithms from §3. Finally, the results are revealed to the intended output parties (§4.3).

4.1 Input Sharing Phase

Depending on the exact application, there are different approaches to providing the initial inputs. We first present a basic approach in a setting comparable to that of [6] as the only prior work for securely computing centrality measures on multilayer graphs. Then, we generalize this approach, discussing additional techniques providing more opportunities for customization.

4.1.1 *Basic Approach.*

Assume that clients agree on an ordering of the vertices in V, e.g., via network reconciliation [34]. Each client C_i holds the DAG list corresponding to its private graph $\mathsf{G}_i = (\mathsf{V}, \mathsf{E}_i)$. Without loss of generality, let C_1 additively secret share (§2.4) its entire DAG list with the servers, while the other clients $\mathsf{C}_2, \ldots, \mathsf{C}_\ell$ additively secret only the edge entries of their respective DAG list. Since the set of vertices is the same across all clients, this ensures that the servers receive precisely one copy of each vertex entry in the DAG list corresponding to the multigraph and that they receive all the edges depending on their multiplicity. Finally, the servers can concatenate the received DAG lists to obtain the complete DAG list comprising the vertex and all the edge entries in secret shared format ($[\![\vec{\mathbf{src}}]\!]$, $[\![\vec{\mathbf{dst}}]\!]$, $[\![\vec{\mathbf{isV}}]\!]$, $[\![\vec{\mathbf{payload}}]\!]$). The data size sent by a client leaks the number of edges in its private graph (but no further information). This can be avoided inexpensively if required by appending some padding entries as shown in Appendix B.4.

4.1.2 *Generalization.*

While [6] relies on a common vertex set V due to their adjacency matrix representation, our representation as DAG lists enables a far more general approach. Each client C_i can input an arbitrary number of entries that can be nodes and edges into the DAG list without the need for public V. We only need to ensure that different clients do not use the same vertex label. That can be achieved by each client reserving a range of vertex labels, e.g., C_1 gets a range $\{0, \ldots, 499\}$ to represent its nodes, C_2 gets $\{500, \ldots, 999\}$, etc. Similar to [6], we assume that the clients identify their common nodes in advance, say via a private set intersection protocol [33, 52]. For instance, if considering online social networks, common nodes/users may be identified by name, email addresses linked to their account, or logins over third parties such as Google. Yet, we only need each pair of clients $\mathsf{C}_i, \mathsf{C}_j$ to know their common nodes. They can then, before providing their inputs to the protocol, coordinate by themselves who inputs which of the common nodes represented by which vertex label. As an example for social networks, assume that $\mathsf{C}_1, \mathsf{C}_2$ both have users *John Doe* and *Jane Doe*. They can coordinate that C_1 inputs both, using labels $0, 1$. Then, C_1 notifies C_2 that 0 corresponds to *John Doe* and 1 to *Jane Doe*. Without information disclosure to C_1, C_2 may now also input an edge $(0, 1)$ if it has such a link. It may also input an edge $(1, 500)$ for a user *A. N. Other* (labeled 500) who is unknown to C_1.

4.2 Evaluation Phase

We proceed by introducing our improved secure protocol for privacy-preserving message-passing algorithms, based on Graphiti [35], and then present how our message-passing algorithms for centrality measures from §3 can be securely realized on top of it. We conclude with a complexity comparison against the only prior work on secure computation of centrality measures [6] and discussing the security of our system. A comparison of our system focusing on the improvements to Graphiti [35] is discussed later in §5.

Our solution relies on a range of sub-protocols, some of them from prior works that we translate to our setting, some based on ideas from prior works, and some new building blocks required specifically for our application. While we explain our overall system and crucial protocol components here, we defer detailed specification regarding all additionally used sub-protocols alongside an exact analysis of involved communication cost in Appendix A.

We begin by first providing our full, generic protocol in Prot. 1. The following parts lead through the different steps of the protocol, supported by the illustration of our secure approach to message-passing provided in Fig. 3.

Protocol 1 Our Secure Message-Passing Protocol

```
$[\![\rho_{\mathrm{src}}]\!] \leftarrow \mathcal{F}_{\textsc{GetSort}}([\![\vec{\mathbf{src}}]\!], [\![\vec{\mathbf{1}} - \vec{\mathbf{isV}}]\!])$ // equal src ⇒ vertex first
$[\![\rho_{\mathrm{dst}}]\!] \leftarrow \mathcal{F}_{\textsc{GetSort}}([\![\vec{\mathbf{dst}}]\!], [\![\vec{\mathbf{isV}}]\!])$ // equal dst ⇒ vertex last
$[\![\rho_{\mathrm{vert}}]\!] \leftarrow \mathcal{F}_{\textsc{GetSort}}([\![\vec{\mathbf{1}} - \vec{\mathbf{isV}}]\!], [\![\vec{\mathbf{src}}]\!], [\![\vec{\mathbf{1}} - \vec{\mathbf{isV}}]\!])$ // Vertices first, ordered
$[\![\vec{\mathbf{payload}}_{\mathrm{v}}]\!] \leftarrow \mathcal{F}_{\textsc{ApplyPerm}}([\![\rho_{\mathrm{vert}}]\!], [\![\vec{\mathbf{payload}}]\!])$
for $D$ iterations do
    $[\![\vec{\mathbf{payload}}'_{\mathrm{v}}]\!] \leftarrow \mathcal{F}_{\textsc{Propagate-1}}([\![\vec{\mathbf{payload}}_{\mathrm{v}}]\!])$
    $[\![\vec{\mathbf{payload}}_{\mathrm{src}}]\!] \leftarrow \mathcal{F}_{\textsc{swPerm}}([\![\rho_{\mathrm{vert}}]\!], [\![\rho_{\mathrm{src}}]\!], [\![\vec{\mathbf{payload}}'_{\mathrm{v}}]\!])$
    $[\![\vec{\mathbf{payload}}'_{\mathrm{src}}]\!] \leftarrow \mathcal{F}_{\textsc{Propagate-2}}([\![\vec{\mathbf{payload}}_{\mathrm{src}}]\!])$
    $[\![\vec{\mathbf{payload}}_{\mathrm{dst}}]\!] \leftarrow \mathcal{F}_{\textsc{swPerm}}([\![\rho_{\mathrm{src}}]\!], [\![\rho_{\mathrm{dst}}]\!], [\![\vec{\mathbf{payload}}'_{\mathrm{src}}]\!])$
    $[\![\vec{\mathbf{payload}}'_{\mathrm{dst}}]\!] \leftarrow \mathcal{F}_{\textsc{Gather-1}}([\![\vec{\mathbf{payload}}_{\mathrm{dst}}]\!])$
    $[\![\vec{\mathbf{payload}}'_{\mathrm{v}}]\!] \leftarrow \mathcal{F}_{\textsc{swPerm}}([\![\rho_{\mathrm{src}}]\!], [\![\rho_{\mathrm{dst}}]\!], [\![\vec{\mathbf{payload}}'_{\mathrm{v}}]\!])$
    $[\![\vec{\mathbf{update}}_{\mathrm{v}}]\!] \leftarrow \mathcal{F}_{\textsc{Gather-2}}([\![\vec{\mathbf{payload}}'_{\mathrm{v}}]\!])$
    $[\![\vec{\mathbf{payload}}_{\mathrm{v}}]\!] \leftarrow \mathcal{F}_{\textsc{Apply}}([\![\vec{\mathbf{payload}}_{\mathrm{v}}]\!], [\![\vec{\mathbf{update}}_{\mathrm{v}}]\!])$
end for
```

4.2.1 One-Time Initialization. For highly efficient message-passing in our protocol, it is necessary to alternately bring the DAG list into vertex order, source order, and destination order (cf. §2.5). Our protocol follows a novel approach here, contrasting Graphiti [35], where we first consider the *input order* (cf. Fig. 3) in which the rows of the DAG list are provided by the different clients. Then, we compute three permutations in parallel that, if applied to the DAG list in input order, would bring it into any of the other orders.

To efficiently compute the required permutations, we utilize the MPC radix sort of [7] to overcome limitations of Graphiti [35] using quicksort as we discuss in detail in §5. The sort of [7] implements functionality $\mathcal{F}_{\textsc{GetSort}}$ (Fig. 4) receiving as input a vector of size n and then providing a secret-shared permutation $[\![\rho]\!] = ([\![\rho(1)]\!], \ldots, [\![\rho(n)]\!])$ that, if applied on the input vector, would order it stably in ascending order. For performance reasons, we expect all entries of the input vectors to be available in bit-decomposed form. This could be implemented by letting all clients already share in this format or using additional protocols for bit decomposition. We defer the details regarding that to Appendix A.4. We also extend the definition of $\mathcal{F}_{\textsc{GetSort}}$ to, e.g., $\mathcal{F}_{\textsc{GetSort}}([\![\vec{\mathbf{dst}}]\!], [\![\vec{\mathbf{isV}}]\!])$, denoting that we sort by destinations dst_i with first priority and isV_i with second priority, i.e., sort by concatenated keys $\mathrm{dst}_i || \mathrm{isV}_i$, for $1 \leq i \leq n$ (cf. Appendix A.4).

Functionality $\mathcal{F}_{\textsc{GetSort}}$

Input: Secret-shared vector $[\![\vec{\mathbf{x}}]\!]$ of dimension n with entry x_i being bit-decomposed, consisting of shares $[\![x_i^{k-1}]\!], \ldots, [\![x_i^0]\!]$.
Output: Secret-shared permutation $[\![\rho]\!]$ with $\rho \in \mathcal{S}_n$ and $\rho(\vec{\mathbf{x}})$ being stably sorted in ascending order, i.e., equal elements remain in their original order from $\vec{\mathbf{x}}$.

Figure 4: Functionality to retrieve permutation to sort data.

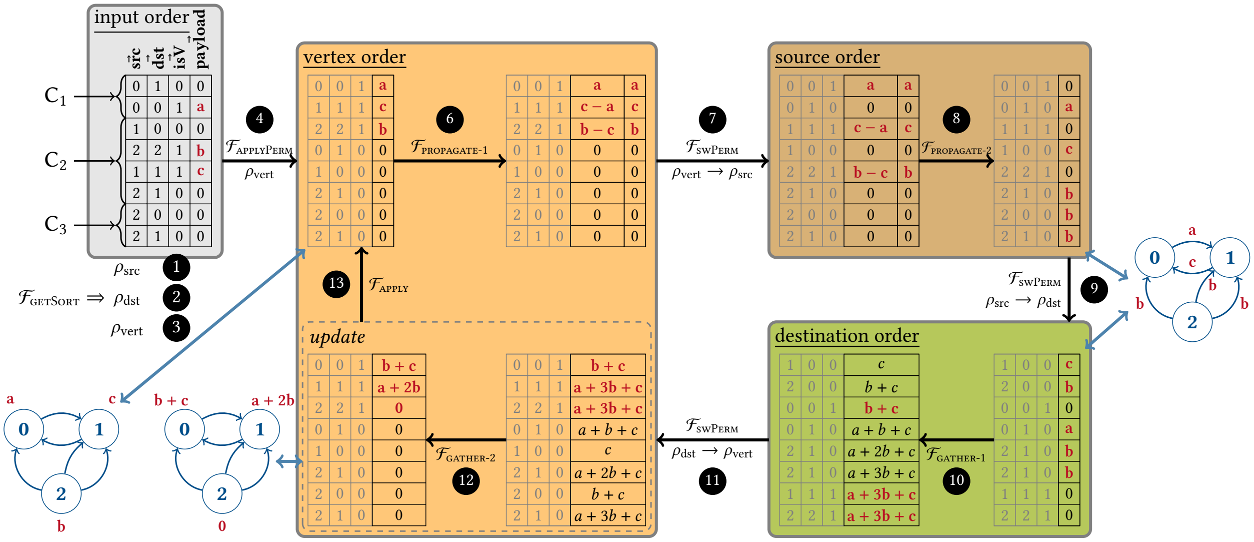
Now, regarding source order, we use $\mathcal{F}_{\textsc{GetSort}}([\![\vec{\mathbf{src}}]\!], [\![\vec{\mathbf{1}} - \vec{\mathbf{isV}}]\!])$ yielding a permutation $[\![\rho_{\mathrm{src}}]\!]$ that can order the DAG list into blocks of entries with the same source entry with nodes coming before edges inside a block, i.e., with each node appearing immediately before its outgoing edges. In parallel, we also use $\mathcal{F}_{\textsc{GetSort}}([\![\vec{\mathbf{dst}}]\!], [\![\vec{\mathbf{isV}}]\!])$ yielding $[\![\rho_{\mathrm{dst}}]\!]$ for the destination order where a block of all edges going into a specific node is immediately followed by said node. Finally, we could use $\mathcal{F}_{\textsc{GetSort}}([\![\vec{\mathbf{1}} - \vec{\mathbf{isV}}]\!], [\![\vec{\mathbf{src}}]\!])$ yielding permutation $[\![\rho_{\mathrm{vert}}]\!]$ for the vertex order, ensuring that all nodes come first and are ordered.

Optimization. We note that a radix sort works by first sorting by the least significant bit of all entries, then ordering by the second least significant bit in a stable manner, etc., until it finally stably sorts by the most significant bit. The same structure can also be found in the MPC version by [7]. We observe that instead of using $\mathcal{F}_{\textsc{GetSort}}([\![\vec{\mathbf{1}} - \vec{\mathbf{isV}}]\!], [\![\vec{\mathbf{src}}]\!])$, we could also use $\mathcal{F}_{\textsc{GetSort}}([\![\vec{\mathbf{1}} - \vec{\mathbf{isV}}]\!], [\![\vec{\mathbf{src}}]\!], [\![\vec{\mathbf{1}} - \vec{\mathbf{isV}}]\!])$ still yielding a valid permutation to reach vertex order. We already require $\mathcal{F}_{\textsc{GetSort}}([\![\vec{\mathbf{src}}]\!], [\![\vec{\mathbf{1}} - \vec{\mathbf{isV}}]\!])$ for the source order, and appending one single radix sort iteration for the bits in $[\![\vec{\mathbf{1}} - \vec{\mathbf{isV}}]\!]$ immediately leads over to the permutation for vertex order. Hence, the combination of radix sort and the correlation between the different required orderings enables us to *compute three sorting permutations for the price of two.* We defer details regarding the sorting to Appendix A.4.

4.2.2 Message-Passing Iterations. After the one-time initialization phase, each message-passing iteration follows Graphiti [35] on a high level. Recall from §2.5 that Graphiti provides non-interactive instantiations of the functionalities $\mathcal{F}_{\textsc{Propagate-1}}$ used while the DAG list is in vertex order and $\mathcal{F}_{\textsc{Propagate-2}}$ used while in source order that together realize the propagate of data from nodes to their outgoing edges. Similarly, it provides $\mathcal{F}_{\textsc{Gather-1}}$ used while in destination order and $\mathcal{F}_{\textsc{Gather-2}}$ used while in vertex order that jointly realize gather of data for nodes from their incoming edges.

Note that to represent a multigraph, the DAG list can simply contain m rows representing an edge of multiplicity m. Our input phase (§4.1) allows the same edge to be added multiple times. Furthermore, careful observation of building blocks $\mathcal{F}_{\textsc{Propagate-1}}$, $\mathcal{F}_{\textsc{Propagate-2}}$, $\mathcal{F}_{\textsc{Gather-1}}$, and $\mathcal{F}_{\textsc{Gather-2}}$ as provided in [35] reveals that they indeed require no modification to handle edges appearing in the DAG list multiple times.

Given the permutations $[\![\rho_{\mathrm{src}}]\!]$, $[\![\rho_{\mathrm{dst}}]\!]$, $[\![\rho_{\mathrm{vert}}]\!]$ from our novel initialization phase, we now need to apply them to the DAG list to switch between different orders. First, we apply permutation


input order
vertex order
source order
destination order
update
C_1
C_2
C_3
ρ_{src} 1
$\mathcal{F}_{\textsc{GetSort}} \Rightarrow \rho_{\text{dst}}$ 2
ρ_{vert} 3
4 $\mathcal{F}_{\textsc{ApplyPerm}}$ ρ_{vert}
6 $\mathcal{F}_{\textsc{propagate-1}}$
7 $\mathcal{F}_{\textsc{swPerm}}$ $\rho_{\text{vert}} \longrightarrow \rho_{\text{src}}$
8 $\mathcal{F}_{\textsc{propagate-2}}$
9 $\mathcal{F}_{\textsc{swPerm}}$ $\rho_{\text{src}} \longrightarrow \rho_{\text{dst}}$
10 $\mathcal{F}_{\textsc{gather-1}}$
11 $\mathcal{F}_{\textsc{swPerm}}$ $\rho_{\text{dst}} \longrightarrow \rho_{\text{vert}}$
12 $\mathcal{F}_{\textsc{gather-2}}$
13 $\mathcal{F}_{\textsc{apply}}$
a
c
b
b + c
a + 2b
0
0
1
2
0
1
2
0
1
2
a
c
b
b
b

input order

$\vec{\textbf{src}}$	$\vec{\textbf{dst}}$	$\vec{\textbf{isV}}$	$\overrightarrow{\textbf{payload}}$
0	1	0	0
0	0	1	a
1	0	0	0
2	2	1	b
1	1	1	c
2	1	0	0
2	0	0	0
2	1	0	0

vertex order

0	0	1	a
1	1	1	c
2	2	1	b
0	1	0	0
1	0	0	0
2	1	0	0
2	0	0	0
2	1	0	0

0	0	1	a	a
1	1	1	c − a	c
2	2	1	b − c	b
0	1	0	0	0
1	0	0	0	0
2	1	0	0	0
2	0	0	0	0
2	1	0	0	0

source order

0	0	1	a	a
0	1	0	0	0
1	1	1	c − a	c
1	0	0	0	0
2	2	1	b − c	b
2	1	0	0	0
2	0	0	0	0
2	1	0	0	0

0	0	1	0
0	1	0	a
1	1	1	0
1	0	0	c
2	2	1	0
2	1	0	b
2	0	0	b
2	1	0	b

destination order

1	0	0	c
2	0	0	$b + c$
0	0	1	b + c
0	1	0	$a + b + c$
2	1	0	$a + 2b + c$
2	1	0	$a + 3b + c$
1	1	1	a + 3b + c
2	2	1	a + 3b + c

1	0	0	c
2	0	0	b
0	0	1	0
0	1	0	a
2	1	0	b
2	1	0	b
1	1	1	0
2	2	1	0

update

0	0	1	b + c
1	1	1	a + 2b
2	2	1	0
0	1	0	0
1	0	0	0
2	1	0	0
2	0	0	0
2	1	0	0

0	0	1	b + c
1	1	1	a + 3b + c
2	2	1	a + 3b + c
0	1	0	$a + b + c$
1	0	0	c
2	1	0	$a + 2b + c$
2	0	0	$b + c$
2	1	0	$a + 3b + c$

Figure 3: Overview of our secure system for privacy-preserving message-passing algorithms, based on Graphiti [35]. Circled numbers correspond to the respective algorithm lines in Prot. 1. The DAG list consists of columns corresponding to $\vec{\textbf{src}}$, $\vec{\textbf{dst}}$, $\vec{\textbf{isV}}$, $\overrightarrow{\textbf{payload}}$. For instance, the row $(0, 0, 1, a)$ represents node 0 holding payload value a while row $(2, 1, 0, 0)$ represents edge $(2, 1)$ holding value 0. The DAG list in input order receives its entry rows by different clients and represents the multigraph on the left side. After finalizing propagate using $\mathcal{F}_{\textsc{propagate-1}}$ and $\mathcal{F}_{\textsc{propagate-2}}$, the resulting DAG list represents the multigraph on the right side where all data has been successfully propagated from nodes to their outgoing edges. Then, after gather, the resulting DAG list represents a new multigraph acting as an update that can be applied to the DAG list before the iteration, e.g., by replacing it or by payload values being added. $\vec{\textbf{src}}$, $\vec{\textbf{dst}}$, $\vec{\textbf{isV}}$ are in gray except for input order as these are never actually reordered by the protocol and are only provided here for better comprehensibility.

$[[\rho_{\text{vert}}]]$ to switch from input to vertex order. To apply a secret-shared permutation to the secret-shared columns of the DAG list, we use functionality $\mathcal{F}_{\textsc{ApplyPerm}}$ (Fig. 5). An efficient protocol for $\mathcal{F}_{\textsc{ApplyPerm}}$ is provided in [7] and also given in Appendix A.2.

Functionality $\mathcal{F}_{\textsc{ApplyPerm}}$

Input: Secret-shared vector $[[\vec{\mathbf{x}}]]$ of dimension n, $[[\rho]]$ for a permutation $\rho \in \mathcal{S}_n$.
Output: Secret-shared vector $[[\rho(\vec{\mathbf{x}})]]$.

Figure 5: Functionality to apply secret-shared permutation to a secret-shared vector.

When reordering the DAG list, we note that all message-passing operations of Graphiti operate on $\overrightarrow{\textbf{payload}}$ only. Hence, only the permutations computed on the input order during initialization depend on the other columns of the DAG list. We exploit that fact by, instead of reordering the entire DAG list during message-passing, only reordering the payload to improve communication.

After switching to vertex order, we apply $\mathcal{F}_{\textsc{propagate-1}}$ and switch from vertex order to source order. As permutation $[[\rho_{\text{src}}]]$ for the source order has been computed on the input order, we would have to first switch back to input order, i.e., applying ρ_{vert}^{-1}, to then go to source order, i.e., apply ρ_{src}. Instead, we introduce functionality $\mathcal{F}_{\textsc{swPerm}}$ (Fig. 6) that merges both steps into one. We hence apply $\mathcal{F}_{\textsc{swPerm}}([[\rho_{\text{vert}}]], [[\rho_{\text{src}}]], [[\overrightarrow{\textbf{payload}}]])$ which applies $\rho_{\text{src}} \circ \rho_{\text{vert}}^{-1}$ to $[[\overrightarrow{\textbf{payload}}]]$.[14] Appendix A.3 explains how $\mathcal{F}_{\textsc{swPerm}}$ could be naïvely realized based on [7]'s $\mathcal{F}_{\textsc{ApplyPerm}}$ and finally provides our own improved implementation of $\mathcal{F}_{\textsc{swPerm}}$ that improves the naïve approach by factor two, requiring only a single round. Propagate then is finalized by application of $\mathcal{F}_{\textsc{propagate-2}}$.

Functionality $\mathcal{F}_{\textsc{swPerm}}$

Input: Secret-shared vector $[[\vec{\mathbf{x}}]]$ of dimension n, $[[\rho_1]]$, $[[\rho_2]]$ for permutations $\rho_1, \rho_2 \in \mathcal{S}_n$.
Output: Secret-shared vector $[[\rho_2(\rho_1^{-1}(\vec{\mathbf{x}}))]]$.

Figure 6: Functionality to switch order of secret-shared vector from one secret-shared permutation to another.

As for Graphiti, gather consists of switching to destination order, in our protocol using $\mathcal{F}_{\textsc{swPerm}}$, applying $\mathcal{F}_{\textsc{gather-1}}$, switching to vertex order, and then applying $\mathcal{F}_{\textsc{gather-2}}$. The resulting payload serves as an *update* to the former one at the beginning of the message-passing iteration. By individually choosing $\mathcal{F}_{\textsc{apply}}$, one could apply this update by, e.g., using it to overwrite the prior payload, or adding prior payload and update, depending on the exact function to be computed (cf. §4.2.3). Then, the next message-passing iteration can commence.

[14] Only during the change from vertex to source order, we temporarily extend the payload to two entries per row instead of one, illustrated in Fig. 3, as per the design of the building blocks provided by Graphiti [35].

As a final remark, in Fig. 3 in-between the propagate respectively the gather steps, the semantics of the intermediate payloads associated to the DAG list entries might not appear directly apparent. Yet, it is exactly what allows to instantiate $\mathcal{F}_{\text{PROPAGATE-1}}$, $\mathcal{F}_{\text{PROPAGATE-2}}$, $\mathcal{F}_{\text{GATHER-1}}$, and $\mathcal{F}_{\text{GATHER-2}}$ non-interactively, utilizing the different orders of the DAG list to eventually come to the correct result. For details on this mechanism, we refer the interested reader to [35].

4.2.3 Computing Centrality Measures in our Framework. Secure computation of π^D_{M} using our previously introduced framework works as follows: As described in §3.1, in each iteration, a node propagates its prior state s^{i-1}_v and replaces it with the newly gathered data. We represent these states as the payload $\vec{\mathbf{payload}}$ which is secret-shared between the servers. Functionality $\mathcal{F}_{\text{APPLY}}$ now has to (a) overwrite the state of the node with the newly gathered data, i.e., the update, and (b) add the weights β_i to the states propagated by each node *before the iteration.*[15] Both steps run non-interactively. Furthermore, all entries of $\vec{\mathbf{payload}}$ are initialized to 0 representing the initial states s^0_v. The instantiation for the metric π^D_{M} within our general framework from Prot. 1 is provided in Prot. 2. Note that we do not give an input phase (for defining $[\![\vec{\mathbf{src}}]\!]$, $[\![\vec{\mathbf{dst}}]\!]$, $[\![\vec{\mathbf{isV}}]\!]$) or output phase (for revealing scores $\pi^D_{\mathsf{M}}(v)$ from the final $[\![\vec{\mathbf{payload}}]\!]$) here as they can be independently designed for a specific deployment setting, as described in §4.1 (inputs) and §4.3 (outputs).

Protocol 2 Secure Message-Passing for π^D_{M}

```
$[\![\vec{\mathbf{payload}}]\!] \leftarrow [\![\vec{0}]\!]$
$[\![\rho_{\text{src}}]\!] \leftarrow \mathcal{F}_{\text{GETSORT}}([\![\vec{\mathbf{src}}]\!], [\![\vec{1} - \vec{\mathbf{isV}}]\!])$ // equal src ⇒ vertex first
$[\![\rho_{\text{dst}}]\!] \leftarrow \mathcal{F}_{\text{GETSORT}}([\![\vec{\mathbf{dst}}]\!], [\![\vec{\mathbf{isV}}]\!])$ // equal dst ⇒ vertex last
$[\![\rho_{\text{vert}}]\!] \leftarrow \mathcal{F}_{\text{GETSORT}}([\![\vec{1} - \vec{\mathbf{isV}}]\!], [\![\vec{\mathbf{src}}]\!], [\![\vec{1} - \vec{\mathbf{isV}}]\!])$ // Vertices first, ordered
$[\![\vec{\mathbf{payload}}_{\mathsf{v}}]\!] \leftarrow \mathcal{F}_{\text{APPLYPERM}}([\![\rho_{\text{vert}}]\!], [\![\vec{\mathbf{payload}}]\!])$
for $1 \le i \le D$ do
    $[\![\vec{\mathbf{payload}}_{\mathsf{v}}]\!] \leftarrow [\![\vec{\mathbf{payload}}]\!] + (\underbrace{\beta_{1+D-i}, \ldots, \beta_{1+D-i}}_{|\mathsf{V}|\text{ times}}, \underbrace{0, \ldots, 0}_{|\mathcal{E}|\text{ times}})$ // $\mathcal{F}_{\text{APPLY}}$
    $[\![\vec{\mathbf{payload}}'_{\mathsf{v}}]\!] \leftarrow \mathcal{F}_{\text{PROPAGATE-1}}([\![\vec{\mathbf{payload}}_{\mathsf{v}}]\!])$
    $[\![\vec{\mathbf{payload}}_{\text{src}}]\!] \leftarrow \mathcal{F}_{\text{SWPERM}}([\![\rho_{\text{vert}}]\!], [\![\rho_{\text{src}}]\!], [\![\vec{\mathbf{payload}}'_{\mathsf{v}}]\!])$
    $[\![\vec{\mathbf{payload}}'_{\text{src}}]\!] \leftarrow \mathcal{F}_{\text{PROPAGATE-2}}([\![\vec{\mathbf{payload}}_{\text{src}}]\!])$
    $[\![\vec{\mathbf{payload}}_{\text{dst}}]\!] \leftarrow \mathcal{F}_{\text{SWPERM}}([\![\rho_{\text{src}}]\!], [\![\rho_{\text{dst}}]\!], [\![\vec{\mathbf{payload}}'_{\text{src}}]\!])$
    $[\![\vec{\mathbf{payload}}'_{\text{dst}}]\!] \leftarrow \mathcal{F}_{\text{GATHER-1}}([\![\vec{\mathbf{payload}}_{\text{dst}}]\!])$
    $[\![\vec{\mathbf{payload}}'_{\mathsf{v}}]\!] \leftarrow \mathcal{F}_{\text{SWPERM}}([\![\rho_{\text{dst}}]\!], [\![\rho_{\text{vert}}]\!], [\![\vec{\mathbf{payload}}'_{\text{dst}}]\!])$
    $[\![\vec{\mathbf{update}}_{\mathsf{v}}]\!] \leftarrow \mathcal{F}_{\text{GATHER-2}}([\![\vec{\mathbf{payload}}'_{\mathsf{v}}]\!])$
    $[\![\vec{\mathbf{payload}}_{\mathsf{v}}]\!] \leftarrow [\![\vec{\mathbf{update}}]\!]$ // $\mathcal{F}_{\text{APPLY}}$
end for
```

As we reduce π^D_{K} to π^D_{M} by transforming the multiset of edges $\mathcal{E}$ to a unified set E (§3.2), we require a one-time preparation step to run a deduplication sub-protocol before using our secure message-passing protocol. As the name suggests, this protocol securely removes duplicate edges by identifying edges that already occurred before and sets them to invalid such that the message-passing iterations will ignore them. Our secure deduplication follows standard techniques, which is why we defer details to Appendix A.6. For completeness, the formal protocol for π^D_{K} is provided in Appendix B.2.

To securely realize π^D_{R}, the information that is propagated during propagate in iteration i comprises s^{i-1}_w, and the subsequently gathered information is added to s^{i-1}_w before finally applying $\mathsf{clip}(\cdot)$ to the result as per Eq. (7) in §3.3. In this way, $\mathcal{F}_{\text{APPLY}}$ can be defined as $\mathcal{F}_{\text{APPLY}}([\![\vec{\mathbf{payload}}_{\mathsf{v}}]\!], [\![\vec{\mathbf{update}}_{\mathsf{v}}]\!]) = \mathsf{clip}([\![\vec{\mathbf{payload}}_{\mathsf{v}}]\!] + [\![\vec{\mathbf{update}}_{\mathsf{v}}]\!])$. We note that $\mathsf{clip}(\cdot)$ can be realized in MPC with standard techniques and provide details in Appendix A.5. Furthermore, it is possible to optimize $\mathcal{F}_{\text{APPLY}}$ to only use clipping in the last message-passing iteration (cf. Appendix B.1), rendering all but the last instance of $\mathcal{F}_{\text{APPLY}}$ non-interactive. Finally, recall that we need $|\mathsf{V}|$ parallel instances of BFS to compute the score for all nodes. For this, we simply extend the definition of the payload from a single vector $\vec{\mathbf{payload}}$ to $|\mathsf{V}|$ independent vectors $\vec{\mathbf{payload}}^{i}$, each for one of the required BFS instances. The verbose, complete protocol description for this embedding of π^D_{R} into our general framework from Prot. 1 is given in Appendix B.2.

4.2.4 Complexity. We report the asymptotic complexities of our protocols in Tab. 1. For a fair comparison, we report our complexity and that of [6] using the MPC setting introduced in §2.3, since the original protocols in [6] are based on less efficient BGW [10] without outsourcing (where clients themselves carry out the computation). Given a sparse graph, i.e., $|\mathcal{E}| \in \mathcal{O}(|\mathsf{V}|)$, observe from Tab. 1 that we reduce the communication per iteration from quadratic to linear for $\pi^D_{\mathsf{M}}, \pi^D_{\mathsf{K}}$, and from cubic to quadratic for π^D_{R}. The complexity of $\mathcal{O}((|\mathsf{V}| + |\mathcal{E}|) \log |\mathsf{V}|)$ is incurred only as a one-time cost for our protocols. Both, [6] and our work have local computation that is linear in the communication. While round complexity is slightly higher for our protocols, as shown in §6, rounds are not the bottleneck for any of the considered protocols.

Table 1: Asymptotic communication and round complexity of all metrics as implemented in [6], and as implemented by us, for $|\mathsf{V}|$ nodes, $|\mathcal{E}|$ edges, D iterations/depth and ℓ layers. Communication is in elements of $\mathbb{Z}_{2^k}$. Concrete values are given in Appendix B.3, Tab. 4.

	Cost	prior [6]	**ours**
π^D_{M}	comm.	$\mathcal{O}(\lvert\mathsf{V}\rvert^2 D)$	$\mathcal{O}((\lvert\mathsf{V}\rvert + \lvert\mathcal{E}\rvert) \log \lvert\mathsf{V}\rvert + (\lvert\mathsf{V}\rvert + \lvert\mathcal{E}\rvert)D)$
	rounds	$\mathcal{O}(D)$	$\mathcal{O}(\log \lvert\mathsf{V}\rvert + D)$
π^D_{K}	comm.	$\mathcal{O}(\lvert\mathsf{V}\rvert^2 \ell + \lvert\mathsf{V}\rvert^2 D)$	$\mathcal{O}((\lvert\mathsf{V}\rvert + \lvert\mathcal{E}\rvert) \log \lvert\mathsf{V}\rvert + (\lvert\mathsf{V}\rvert + \lvert\mathcal{E}\rvert)D)$
	rounds	$\mathcal{O}(\log(\ell) + D)$	$\mathcal{O}(\log \lvert\mathsf{V}\rvert + \log(k) + D)$
π^D_{R}	comm.	$\mathcal{O}(\lvert\mathsf{V}\rvert^3 D)$	$\mathcal{O}((\lvert\mathsf{V}\rvert^2 + \lvert\mathsf{V}\rvert \cdot \lvert\mathcal{E}\rvert)D)$
	rounds	$\mathcal{O}(\log(k) + D)$	$\mathcal{O}(\log \lvert\mathsf{V}\rvert + \log(k) + D)$

4.2.5 Security. The security of our protocols immediately follows from the correctness of the vertex-centric approach and the privacy guarantees of the MPC protocol for the same. In particular, our protocols are built from MPC building blocks that, on their own, are proven secure in the semi-honest setting. Some building blocks in Appendix A reconstruct intermediate values but only after masking, and we show their security in Appendix A. To prove the

[15] Adding weights *before* each iteration requires a slight and simple change from the generic template as provided by Prot. 1 by splitting $\mathcal{F}_{\text{APPLY}}$ into two different steps, one before and one after each iteration, each while in vertex order.

security of our overall protocols, it is sufficient to prove security in a hybrid model where the underlying building blocks are realized via a trusted third party using the standard real-world/ideal-world simulation paradigm. Note that instantiating these building blocks via their secure MPC protocols yields our secure protocol as security is directly inherited using standard composition theorems [17, 39]. We provide a full formal proof in Appendix B.5.

4.3 Output Phase

After the evaluation phase, the computed centrality scores are available as secret-shared data and hence can be disclosed to any entity by simply sending all output shares to that entity. Depending on the exact application, it can be freely decided who receives which output. For example, all clients could receive the scores for all nodes or, alternatively, only for the nodes they provided as input individually. Alternatively, the output could be kept private and utilized in further secure computation, making our protocol a secure building block for more complex systems.

5 Discussion of Our Improvements

In this section, we discuss the improvements of our approach to secure message-passing algorithms introduced in §4 over its basis provided by Graphiti [35].

First, note that our approach to input sharing (§4.1) improves the efficiency of not only the protocols for centrality measures from [6] that are entirely based on adjacency matrices, but also improves over the input sharing described in Graphiti [35]. Elaborately, while using DAG lists during the main computation, Graphiti's input sharing involves generating a secret-sharing of the adjacency matrix representation for the input graph and incurs $O(|\mathsf{V}|^2)$ communication. In contrast, in our case, all clients directly share their slice of the DAG list, resulting in $O(|\mathsf{V}|+|\mathcal{E}|)$ communication. Hence, while Graphiti manages to remove any $O(|\mathsf{V}|^2)$ factors during the protocol's main computation phase, our approach ensures the absence of such quadratic factors throughout the complete computation process, including the input phase. This is crucial to optimize not only run time and communication but also local memory requirements for sparse graphs with many nodes. When using adjacency matrices, scalability will be severely limited by the graph representation size, as shown later in §6. Yet, starting from an adjacency matrix allows Graphiti to then non-expensively build a DAG list immediately in vertex order so that sorting is only required for the other two orders. Starting from the new, arbitrary input order, our protocol also requires sorting to reach vertex order, but using the optimization from §4.2.1 that computes three sorting permutations at the price of two, essentially nullifies this supposed disadvantage. To summarize, we avoid quadratic overhead in the input phase, whereas Graphiti appears not to focus on making this phase efficient. The strategies from §4.1 furthermore offer possibilities for clients to discover common nodes which was not discussed in [35]. Finally, our solution supports multigraphs while Graphiti does not due to its input phase. On their own, the later message-passing primitives of Graphiti could otherwise also support edges of multiplicity > 1, which we exploit in §4.2.2. Hence, our alternative approach for the input phase can also be used to directly enhance Graphiti, rectifying its drawbacks concerning scalability of the input phase, strategies for discovering common nodes, and support for multilayer graphs.

While we use message-passing functionalities $\mathcal{F}_{\text{PROPAGATE-1}}$, $\mathcal{F}_{\text{PROPAGATE-2}}$, $\mathcal{F}_{\text{GATHER-1}}$, and $\mathcal{F}_{\text{GATHER-2}}$ as provided by Graphiti on the DAG list alternating between vertex order, source order, and destination order, our protocol substantially differs in how it switches between these orders. Graphiti uses the same approach as its predecessor [5] for that which is based on quicksort. In detail, sorting the DAG list entries for each of the specific orders is required as a one-time initialization, while revisiting the same ordering comes relatively cheap at the cost of one shuffling operation, which they implement using only a single communication round.

We identify multiple problems with the use of quicksort in the context of Graphiti:

1. The utilized MPC version of quicksort [5, 29] is secure only if all keys to sort are unique. To ensure that, [5] adds a unique padding to all keys. This results in higher asymptotic complexity and, hence, lower scalability than the MPC radix sort [7] that we apply (cf. §4.2.1).[16]
2. The program flow of quicksort is not oblivious, yielding a lack of predictability regarding run time and communication cost of the protocol execution. Furthermore, the dynamic program flow introduces engineering difficulties for a well-designed and efficient MPC implementation.
3. The non-constant communication requirements of MPC quicksort also translate to the preprocessing phase. The exact amount of required preprocessing material only becomes known in the protocol's online phase, necessitating overprovisioning during its preprocessing phase.[17]

Our solution overcomes these problems by using radix sort for the initialization phase (§4.2.1), which also changes how the DAG list is reordered in each of the message-passing iterations (§4.2.2). The change in reordering using $\mathcal{F}_{\text{swPerm}}$ stems from the fact that Graphiti's sorting immediately reorders data while the use of radix sort decouples computing a permutation vector (for sorting) from its application to the data. While protocols for applying permutations exist [7], we design new ones to switch between different permutations more efficiently (Appendix A.3). After the first message-passing iteration, our protocol requires one modified shuffle and local operations to change orderings, whereas Graphiti uses (standard) shuffle and local operations. The performances for Graphiti's and our approach to switching between permutations are equal, especially with both requiring just a single round. Beyond this similarity, the design for changing orderings is significantly different.

To summarize, we note that Graphiti's predecessor [5] focused mainly on optimizing the change between different orderings using sorting, while Graphiti's goal was to improve the propagate and gather steps to render them non-interactive. We remain in the same setting as Graphiti with two servers and a helper. Then, we aim at again improving the sorting as the approach from [5]

[16] This improvement is not based on hiding the size of keys in the asymptotic notation as it is sometimes done for plaintext radix sort. To sort n keys consisting of k bits each, the radix sort of [7] requires $O(k \cdot n \log n)$ bits communication while the quicksort in [5] requires $O((k + \log n) \cdot n \log n)$ bits in the average case.

[17] One may manage a pool of preprocessing material for multiple protocol runs to reuse unused material in later protocol runs. Yet, this does not resolve the problem as the preprocessing material has to be generated for a fixed size of the DAG list, rendering it partly unsuitable for later use on graphs with other sizes.

(in another setting) is less scalable and yields conflicts with the setting of Graphiti and our work. Furthermore, we provide scalable approaches for the input phase (§4.1) which previously were not provided by [5, 35] while also generalizing to multigraphs.

6 Experimental Evaluation

We evaluate the performance of our protocols, report concrete run time and communication cost, and compare it against [6].

Benchmark Environment. We fully implement our secure framework for message-passing, building on top of Graphiti's [35] codebase that only includes implementation for single building blocks, intended for micro-benchmarking and excluding the initialization phase. Using our framework, we implement our instantiation of the centrality measures from [6] as well as, for a fair comparison, the prior instantiations from [6] in our setting with two parties and one helper, thereby immediately improving [6]'s performance by using state-of-the-art primitives.[18] We use TLS to implement secure channels, which was missing in the original codebase of [35]. Our code is publicly available at https://encrypto.de/code/MultiCent.

We benchmark on three servers, each equipped with an Intel Core i9-7960X CPU @ 2.8 GHz, 128 GB of DDR4 RAM @ 2666 MHz. To simulate realistic real-world network behavior between the servers, we use the Linux tool `tc` (traffic control), focusing on a LAN setting with 1 ms RTT and 1 GBit/s bandwidth. WAN benchmarks and full benchmark data are provided in Appendix C. The benchmarks report concrete costs assuming the input has been shared among the servers since input sharing heavily depends on client-side capabilities, which are generally considered to be much less than the computing parties. However, note that input sharing runs in a single round, and communication therein can be derived from the graph representation size as will be introduced in Tab. 2.

Real-World Graph Instances. We consider several real-world sparse multigraphs, as used in [6], which are summarized in Tab. 2. Note that the performance of the secure protocols only relies on the general size parameters since MPC protocols for graph analysis must have a control flow independent of the graph structure so that no private information is leaked. For [6] this is the number of nodes |V| and layers ℓ, while our approach also requires knowledge of the total number of edges $|\mathcal{E}|$. Since some of the considered graphs are undirected, following [6], we reduce these instances to directed graphs by duplicating each edge, having it once in each direction.

Dataset "aarhus" [45] consists of relationships between employees of the computer science department of Aarhus University. Layers represent different kinds of relationships such as work, co-authorship, or social media connections. Interactions between significantly more people are captured by the "higgs" dataset [19] that considers different kinds of interactions (e.g., following, retweeting) between Twitter users related to the supposed discovery of the Higgs boson in 2012. While these datasets correspond to human interactions and especially "higgs" directly captures social media interaction, we also consider the "london" dataset [23] modeling public rail transport connections in London and the "hiv" and "arabi" datasets [58] representing genetic interaction networks, extending the scope to more diverse kinds of networks. The exact dataset parameter selection is kept as in [6] to ensure full comparability.

Table 2: Considered datasets, either undirected (U) or directed (D), with ℓ layers, |V| nodes, and $|\mathcal{E}| = \sum_{i=1}^{\ell} |\mathsf{E}_i|$ edges across all layers. Graph representation size denotes memory required by [6] and us when using 32-bit integers ($k = 32$).

dataset	U/D	ℓ	|V|	$\|\mathcal{E}\|$	graph representation size		
					prior [6]	**ours**	**improv.**
aarhus	U	5	61	620	14.54 KiB	15.25 KiB	0.95×
london	U	3	369	503	531.88 KiB	16.11 KiB	33.01×
hiv	D	3	1,005	2,688	3.85 MiB	43.28 KiB	91.17×
arabi	U	7	6,980	18,117	185.85 MiB	506.41 KiB	375.81×
higgs	D	3	304,691	1,110,962	345.84 GiB	16.20 MiB	21.86**k**×

The graph representation size captures $|\mathsf{V}|^2$-sized adjacency matrices for [6] vs. $3(|\mathsf{V}| + |\mathcal{E}|)$-sized DAG lists in our solution. This results in a smaller graph representation size of up to four orders of magnitude. That also directly corresponds to the amount of data that must be secret-shared during input sharing (i.e., total client-to-server communication cost) and eventually stored as input at the servers. Thus, memory constraints faced by [6] are evident, where some of the datasets cannot be represented in the RAM of realistic hardware. Hence, [6] only reports performance on the smallest three datasets when securely computing $\pi_{\mathsf{M}}^{D}, \pi_{\mathsf{K}}^{D}, \pi_{\mathsf{R}}^{D}$.

Experiments on Real-World Graphs. Following experiments in [6], we benchmark the protocols on graph instances in Tab. 2 for different numbers of iterations D. The protocols' run time and communication are provided in Fig. 7.

Suitability for Larger Graphs: The protocols [6] only run on the first three datasets due to memory constraints, also evident in [6]'s own benchmarks. This showcases the suitability of our protocols for larger graph instances, whereas handling graphs with as little as 7000 nodes for multiple iterations was infeasible for [6]. Notably, while [6] uses social networks as motivation too, their protocols are unsuitable for the large "higgs" dataset representing a social network graph and only run on smaller graphs from other domains. In contrast, our protocols for $\pi_{\mathsf{M}}^{D}, \pi_{\mathsf{K}}^{D}$ clearly enable efficient analysis of this social network graph. Although we cannot report concrete run time for [6] or π_{R}^{D} on the larger datasets, we report the communication cost that would be incurred by [6], as derived from the detailed analytical costs reported in Appendix B.3, Tab. 4. Further, we note that run times for our implementation of [6] are significantly lower than the original numbers from [6] (e.g., on "hiv" they report $>$ 300 s run time at 10 iterations for all protocols, whereas our re-implementation takes below 5 s for $\pi_{\mathsf{M}}^{D}, \pi_{\mathsf{K}}^{D}$), showing that our instantiation in the outsourced setting and efficient computation on rings in contrast to n-party protocols over fields significantly improves their performance.

Improved Performance with Increasing D: The cost for all protocols scales linearly with D, as expected, given the asymptotic complexities in Tab. 2. Note that the protocols in [6] incur the same *quadratic* cost (for $\pi_{\mathsf{M}}^{D}, \pi_{\mathsf{K}}^{D}$) and *cubic* cost (for π_{R}^{D}) for each iteration

[18] There exists no public reference implementation of [6] in any setting.

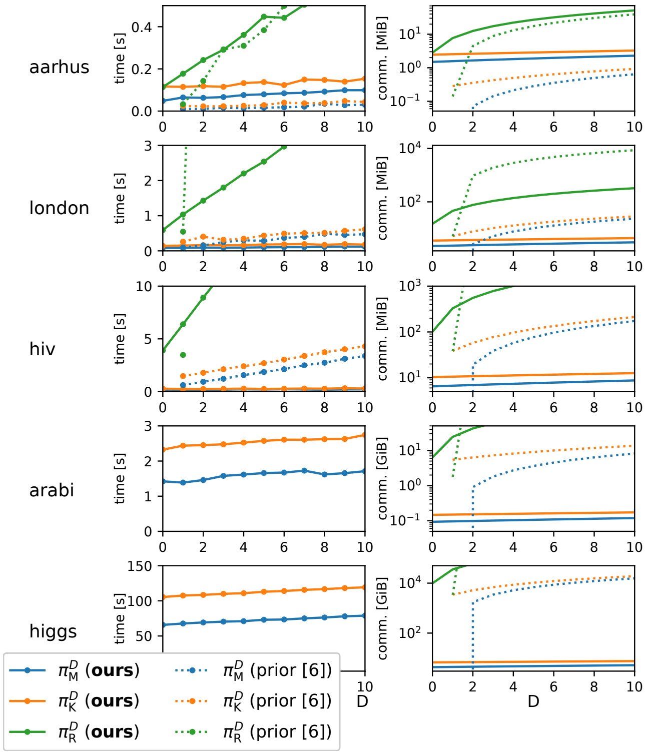


Figure 7: Total run times and communication (preprocessing + online) of our protocols and our implementation of prior protocols [6] for different metrics on graph instances in Tab. 2 for varying numbers of iterations D.

D. On the other hand, our protocols incur a one-time cost for the initialization phase (including the cost of clip$(\cdot)$ for π_{R}^{D}), captured via $D = 0$, which is our bottleneck. The subsequent message-passing iterations, captured by $D > 0$, are highly efficient incurring only *linear* cost for $\pi_{\mathrm{M}}^{D}, \pi_{\mathrm{K}}^{D}$, and *quadratic* cost for π_{R}^{D}, thereby improving [6] by a factor of $|\mathsf{V}|$. Despite the one-time initialization cost being a bottleneck, we observe that our protocols for all metrics outperform those of [6], for all instances where [6] is reported, except the "aarhus" dataset. This is because the graph is small and dense while our constant factors are higher.

Observations Specific to $\pi_{\mathrm{M}}^{D}, \pi_{\mathrm{K}}^{D}, \pi_{\mathrm{R}}^{D}$: Concerning π_{M}^{D} and π_{K}^{D}, trends are similar, with the only difference in cost attributed to the additional computations required for our π_{K}^{D} to transform the multilayer graph to a unified graph, which is a one-time operation.

Although no explicit initialization is required for any of the protocols in [6], there is a difference in the first iteration ($D = 1$). For π_{M}^{D}, note that this computation does not require any interactions among the parties since the operations comprise simple additions performed on the input matrix that is already shared. For π_{K}^{D}, this captures the cost of performing some aggregation using interactive MPC operations on the input. Hence, the cost of π_{K}^{D} is slightly higher than π_{M}^{D}. However, note that this is a one-time cost. Similarly, for π_{R}^{D} this denotes the one-time cost of performing clipping to bring down the values to $\{0, 1\}$. Hence, across all the protocols, the quadratic ($\pi_{\mathrm{M}}^{D}, \pi_{\mathrm{K}}^{D}$) and cubic ($\pi_{\mathrm{R}}^{D}$) growth w.r.t. the graph size starts at $D > 1$ after a cheap, first iteration.

Note that computing π_{R}^{D}, in the case of both ours and [6], becomes infeasible with increasing D, due to significantly higher complexity than the other metrics. This is clearly evident for "hiv", where the prior [6] protocol for π_{R}^{D} only runs for $D = 1$ and further iterations immediately exhaust the available memory.

Scalability for Larger Graphs. To showcase the scalability of our protocols for much larger graphs (in addition to the real-world datasets considered above), we investigate the performance of our protocols by varying $|\mathsf{V}|$ and compare it against [6]. Moreover, to account for different levels of sparsity[19], we consider graphs with $|\mathcal{E}| = 10|\mathsf{V}|$, $|\mathcal{E}| = 50|\mathsf{V}|$, and $|\mathcal{E}| = 100|\mathsf{V}|$. Recall that the cost per iteration remains the same for the protocols in [6] as well as ours (except for a one-time initialization cost). Hence, for varying $|\mathsf{V}|$ and $|\mathcal{E}|$, Fig. 8 explicitly reports the one-time cost and the cost per iteration (allowing extrapolation to arbitrarily many iterations).

Observations for $\pi_{\mathrm{M}}^{D}, \pi_{\mathrm{K}}^{D}, \pi_{\mathrm{R}}^{D}$: Regarding our $\pi_{\mathrm{M}}^{D}, \pi_{\mathrm{K}}^{D}$, as also reported above, the one-time cost is higher than the per iteration cost.[20] This is consistent with our $O(|\mathsf{V}| \log |\mathsf{V}|)$ one-time asymptotic complexity and $O(|\mathsf{V}|)$ per iteration for $|\mathcal{E}| \in O(|\mathsf{V}|)$. Interestingly, despite one-time communication for the prior π_{M}^{D} protocol [6] being 0, their run time is worse than our protocol as their local computation still scales with the quadratic size of the adjacency matrix, showcasing the adverse impact of quadratic computation cost. The poor scalability of [6] also immediately becomes apparent for all metrics from the cost per iteration. It is not feasible to run the prior protocols [6] on more than 8k nodes at all, whereas our protocols for π_{M}^{D} and π_{K}^{D} scale well. Our protocol for π_{R}^{D} also outperforms the prior one [6] due to better cost per iteration, yet it is not very efficient on larger graphs.

Scalability for Huge Graphs: In Fig. 9, we investigate the scalability of π_{M}^{D} for even larger graphs with up to 500k nodes using $|\mathcal{E}| = 10|\mathsf{V}|$. Recall that the online run time determines the run time of the protocol once the input is made available. Due to its significance, we also investigate the split between online and preprocessing run times and communication here. Even for 500k nodes and 5 million edges, the total one-time cost is 300 s, only 124 s of them being online, with communication of at most 17 GiB total. The cost per iteration is significantly cheaper, staying at 8 s and 400 MiB total, with only 3 s in the online phase, making our approach highly scalable for varying numbers of iterations.

7 Related Work

To the best of our knowledge, [6] is the only work that designs secure solutions for analyzing multilayer graphs. It completely relies on encoding graphs as adjacency matrices $\mathbf{A}$. To compute paths of length i, it exploits that $(\mathbf{A}^i)_{v,w}$ equals the number of paths from node v to w. While this approach via matrix multiplication is simple on a conceptional level, it requires $|\mathsf{V}|^3$ multiplications for each but the first iteration. For $\pi_{\mathrm{M}}^{D}, \pi_{\mathrm{K}}^{D}$, [6] optimize this to quadratic complexity, still ruling out scalability as we can also concretely see

[19] Due to operating on adjacency matrices, the cost of [6] is independent of the sparsity.
[20] The discontinuities in communication are at powers of two, stemming from the $O(|\mathsf{V}| \log |\mathsf{V}|)$ complexity which concretely contains a $\lceil \log_2 |\mathsf{V}| \rceil$ term.

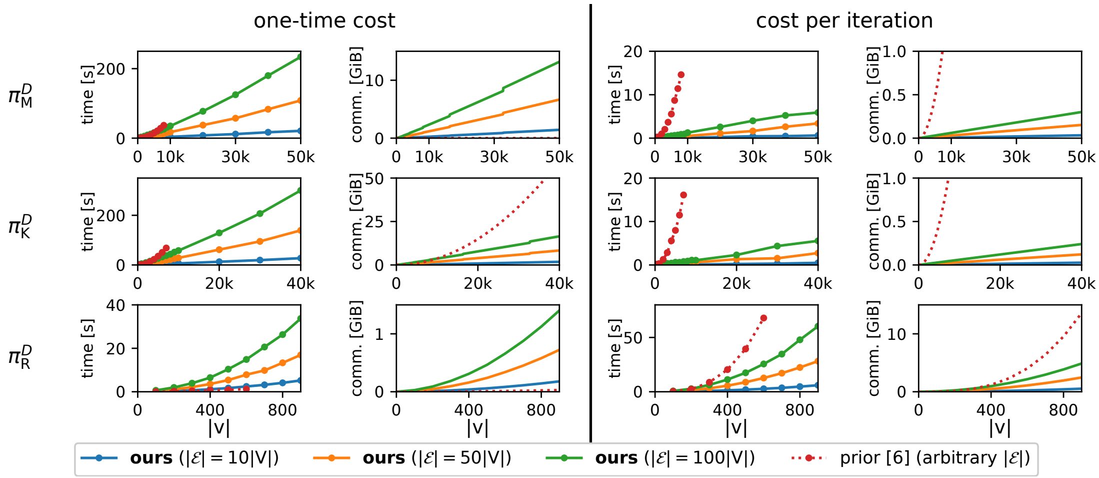


Figure 8: Total run times and communication (preprocessing + online) of our protocols vs. prior protocols [6] ($\ell = 3$ layers).

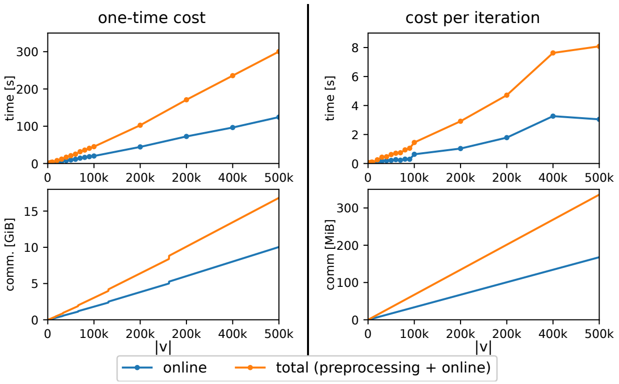


Figure 9: Run times and communication (over all parties) for our π^D_M with $|\mathcal{E}| = 10|V|$, split into online and total (preprocessing + online) cost ($\ell = 3$ layers).

in §6. All required operations in [6] are instantiated using the BGW protocol [10], additionally reducing practical performance. Due to the lack of other solutions for multilayer graphs, in what follows, we focus our attention on works that design secure solutions for graph analysis, despite them being on simple graphs.

There have been several works in the literature that design secure MPC-based solutions specific to the considered graph algorithms [2–4, 11, 12, 16]. Apart from considering traditional graph algorithms, a few works specifically look at securely computing centrality measures on simple graphs [31, 36, 37, 57, 60]. However, with scalability being the primary concern in all of these works, GraphSC [49] was the first work to design a generic framework that facilitated secure realizing any graph algorithm that is expressed as a message-passing algorithm. GraphSC proposed a vertex-centric approach where data is passed between nodes over their connecting edges in multiple iterations, thereby operating on a DAG list (cf. §2.5) representation of the graph. Further, to efficiently realize message passing, GraphSC proposed transitioning between different orderings of the DAG list. Performance of this approach was greatly improved in [5] by (1) switching from a two-party to a three-party honest majority setting, and (2) improving the reordering of the DAG list by only requiring to run sorting algorithms once during an initialization phase while [49] needs to sort in each iteration. Graphiti [35] is the state of the art in this line of work. It uses [5]'s approach for efficient reordering, but improves upon the propagate and gather steps of message-passing themselves (cf. §2.5). Furthermore, it allows one party to act as a helper only, assisting in a preprocessing phase that was not utilized in [5]. There also exist message-passing works leaking structural information on the graph [46, 47] predating the work of [5] that also discusses the limitations yielded by the leakage.

We discuss our conceptual improvements over [35] as state of the art for analysis of simple graphs using message-passing in §5. We provide no concrete efficiency comparison due to the lack of details on and implementation of an end-to-end system in [35].

8 Conclusion

We design secure and efficient protocols for computing centrality measures in multilayer graphs. In contrast to the prior work [6], we develop practical and scalable solutions by making several conscious design choices. Most notably, we switch from a matrix representation to a list representation, a paradigm shift that significantly improves efficiency. However, this shift requires carefully designed, data-oblivious protocols to ensure no information about the underlying graph is leaked. To address this, we introduce novel vertex-centric protocols for computing centrality measures based on the list representation. As a basis for our protocols, we develop a secure graph analysis framework for multigraphs, extending and improving Graphiti [35] while also providing a functional end-to-end implementation. Implementing our protocols for centrality measures on top of this new framework, we show that they are, by orders of magnitude, more efficient than prior work [6] and now scale to handle graph sizes in the millions.

Acknowledgments

The authors thank Joachim Schmidt (TU Darmstadt) and Maximilian Stillger (TU Darmstadt) for contributing to the codebase the clipping operation as well as TLS channels and their integration.

This project received funding from the European Research Council (ERC) under the European Union's research and innovation programs Horizon 2020 (PSOTI/850990) and Horizon Europe (PRIVTOOLS/101124778). It was co-funded by the Deutsche Forschungsgemeinschaft (DFG) within SFB 1119 CROSSING/236615297.

References

[1] Pranav Shriram A, Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, and Bhavish Raj Gopal. 2023. Find Thy Neighbourhood: Privacy-Preserving Local Clustering. *Proceedings on Privacy Enhancing Technologies (PoPETs)* 2023, 2 (2023), 23–39.

[2] Abdelrahaman Aly, Edouard Cuvelier, Sophie Mawet, Olivier Pereira, and Mathieu Van Vyve. 2013. Securely Solving Simple Combinatorial Graph Problems. In *Financial Cryptography and Data Security (FC)*. Springer, 239–257.

[3] Abdelrahaman Aly and Mathieu Van Vyve. 2015. Securely Solving Classical Network Flow Problems. In *International Conference on Information Security and Cryptology (ICISC)*. Springer, 205–221.

[4] Mohammad Anagreh, Peeter Laud, and Eero Vainikko. 2021. Parallel Privacy-Preserving Shortest Path Algorithms. *Cryptography* 5, 4 (2021), 27.

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

[6] Gilad Asharov, Francesco Bonchi, David Garcia-Soriano, and Tamir Tassa. 2017. Secure Centrality Computation Over Multiple Networks. In *International World Wide Web Conference (WWW)*. ACM, 957–966.

[7] Gilad Asharov, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Ariel Nof, Benny Pinkas, Katsumi Takahashi, and Junichi Tomida. 2022. Efficient Secure Three-Party Sorting with Applications to Data Analysis and Heavy Hitters. In *Conference on Computer and Communications Security (CCS)*. ACM, 125–138.

[8] Muhammad A Awad, Saman Ashkiani, Serban D Porumbescu, and John D Owens. 2020. Dynamic Graphs on the GPU. In *International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 739–748.

[9] Donald Beaver. 1991. Efficient Multiparty Protocols Using Circuit Randomization. In *International Cryptology Conference (CRYPTO)*. Springer, 420–432.

[10] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. 1988. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract). In *Symposium on Theory of Computing (STOC)*. ACM, 1–10.

[11] Marina Blanton and Siddharth Saraph. 2015. Secure and Oblivious Maximum Bipartite Matching Size Algorithm with Applications to Secure Fingerprint Identification. In *European Symposium on Research in Computer Security (ESORICS)*. Springer, 384–406.

[12] Marina Blanton, Aaron Steele, and Mehrdad Alisagari. 2013. Data-Oblivious Graph Algorithms for Secure Computation and Outsourcing. In *ASIA Conference on Computer and Communications Security (ASIACCS)*. ACM, 207–218.

[13] Elette Boyle, Nishanth Chandran, Niv Gilboa, Divya Gupta, Yuval Ishai, Nishant Kumar, and Mayank Rathee. 2021. Function Secret Sharing for Mixed-Mode and Fixed-Point Secure Computation. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*. Springer, 871–900.

[14] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2019. Secure Computation With Preprocessing via Function Secret Sharing. In *Theory of Cryptography Conference (TCC)*. Springer, 341–371.

[15] Guillaume Braun, Hemant Tyagi, and Christophe Biernacki. 2021. Clustering Multilayer Graphs with Missing Nodes. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*. PMLR, 2260–2268.

[16] Justin Brickell and Vitaly Shmatikov. 2005. Privacy-Preserving Graph Algorithms in the Semi-Honest Model. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*. Springer, 236–252.

[17] Ran Canetti. 2000. Security and Composition of Multiparty Cryptographic Protocols. *Journal of Cryptology* 13, 1 (2000), 143–202.

[18] Sharma Chakravarthy, Abhishek Santra, and Kanthi Sannappa Komar. 2019. Why Multilayer Networks Instead of Simple Graphs? Modeling Effectiveness and Analysis Flexibility and Efficiency!. In *International Conference on Big Data Analytics (ICBDA)*. Springer, 227–244.

[19] Manlio De Domenico, Antonio Lima, Paul Mougel, and Mirco Musolesi. 2013. The Anatomy of a Scientific Rumor. *Scientific Reports* 3, 1 (2013), 2980.

[20] Manlio De Domenico, Albert Solé-Ribalta, Elisa Omodei, Sergio Gómez, and Alex Arenas. 2015. Ranking in Interconnected Multilayer Networks Reveals Versatile Nodes. *Nature Communications* 6, 1 (2015), 6868.

[21] Charo I Del Genio, Thilo Gross, and Kevin E Bassler. 2011. All Scale-Free Networks are Sparse. *Physical Review Letters* 107, 17 (2011), 178701.

[22] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In *Network and Distributed System Security Symposium (NDSS)*. IEEE.

[23] Manlio De Domenico, Albert Solé-Ribalta, Sergio Gómez, and Alex Arenas. 2014. Navigability of Interconnected Networks Under Random Failures. *Proceedings of the National Academy of Sciences* 111, 23 (2014), 8351–8356.

[24] Kurt C Foster, Stephen Q Muth, John J Potterat, and Richard B Rothenberg. 2001. A Faster Katz Status Score Algorithm. *Computational & Mathematical Organization Theory* 7 (2001), 275–285.

[25] Hildreth Robert Frost. 2023. Eigenvector Centrality for Multilayer Networks with Dependent Node Importance. In *International Conference on Complex Networks and Their Applications*. Springer, 3–14.

[26] H Robert Frost. 2024. A Generalized Eigenvector Centrality for Multilayer Networks with Inter-Layer Constraints on Adjacent Node Importance. *Applied Network Science* 9, 1 (2024), 14.

[27] Zhie Gao and Amin Rezaeipanah. 2023. A Novel Link Prediction Model in Multilayer Online Social Networks Using the Development of Katz Similarity Metric. *Neural Processing Letters* 55, 4 (2023), 4989–5011.

[28] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *Symposium on Theory of Computing (STOC)*. ACM, 218–229.

[29] Koki Hamada, Ryo Kikuchi, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. 2012. Practically Efficient Multi-party Sorting Protocols from Comparison Sort Algorithms. In *International Conference on Information Security and Cryptology (ICISC)*. Springer, 202–216.

[30] Yan Huang, David Evans, and Jonathan Katz. 2012. Private Set Intersection: Are Garbled Circuits Better than Custom Protocols?. In *Network and Distributed System Security Symposium (NDSS)*. IEEE.

[31] Florian Kerschbaum and Andreas Schaad. 2008. Privacy-Preserving Social Network Analysis for Criminal Investigations. In *Workshop on Privacy in the Electronic Society (WPES)*. ACM, 9–14.

[32] Mikko Kivelä, Alex Arenas, Marc Barthelemy, James P Gleeson, Yamir Moreno, and Mason A Porter. 2014. Multilayer Networks. *Journal of Complex Networks* 2, 3 (2014), 203–271.

[33] Vladimir Kolesnikov, Naor Matania, Benny Pinkas, Mike Rosulek, and Ni Trieu. 2017. Practical Multi-Party Private Set Intersection from Symmetric-Key Techniques. In *Conference on Computer and Communications Security (CCS)*. ACM, 1257–1272.

[34] Nitish Korula and Silvio Lattanzi. 2014. An Efficient Reconciliation Algorithm for Social Networks. *International Conference on Very Large Data Bases (VLDB)* 7, 5 (2014), 377–388.

[35] Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, and Bhavish Raj Gopal. 2024. Graphiti: Secure Graph Computation Made More Scalable. In *Conference on Computer and Communications Security (CCS)*. ACM, 4017–4031.

[36] Varsha Bhat Kukkala and S. R. S. Iyengar. 2018. Computing Betweenness Centrality: An Efficient Privacy-Preserving Approach. In *International Conference on Cryptology and Network Security (CANS)*. Springer, 23–42.

[37] Varsha Bhat Kukkala and S. R. S. Iyengar. 2020. Identifying Influential Spreaders in a Social Network (While Preserving Privacy). *Proceedings on Privacy Enhancing Technologies (PoPETs)* 2020, 2 (2020), 537–557.

[38] Tarun Kumar, Ramanathan Sethuraman, Sanga Mitra, Balaraman Ravindran, and Manikandan Narayanan. 2023. MultiCens: Multilayer Network Centrality Measures to Uncover Molecular Mediators of Tissue-Tissue Communication. *PLOS Computational Biology* 19, 4 (2023), e1011022.

[39] Eyal Kushilevitz, Yehuda Lindell, and Tal Rabin. 2006. Information-Theoretically Secure Protocols and Security Under Composition. In *Symposium on Theory of Computing (STOC)*. ACM, 109–118.

[40] David Liben-Nowell and Jon Kleinberg. 2003. The Link Prediction Problem for Social Networks. In *International Conference on Information and Knowledge Management (CIKM)*. ACM, 556–559.

[41] Mingkai Lin, Wenzhong Li, Lynda J Song, Cam-Tu Nguyen, Xiaoliang Wang, and Sanglu Lu. 2021. SAKE: Estimating Katz Centrality Based on Sampling for Large-Scale Social Networks. *Transactions on Knowledge Discovery from Data (TKDD)* 15, 4 (2021), 1–21.

[42] Boge Liu, Fan Zhang, Chen Zhang, Wenjie Zhang, and Xuemin Lin. 2019. CoreCube: Core Decomposition in Multilayer Graphs. In *International Conference on Web Information Systems Engineering (WISE)*. Springer, 694–710.

[43] Dandan Liu and Zhaonian Zou. 2023. gCore: Exploring Cross-layer Cohesiveness in Multi-layer Graphs. *International Conference on Very Large Data Bases (VLDB)* 16, 11 (2023), 3201–3213.

[44] Zhengdong Lu, Berkant Savas, Wei Tang, and Inderjit S Dhillon. 2010. Supervised Link Prediction Using Multiple Sources. In *International Conference on Data Mining (ICDM)*. IEEE, 923–928.

[45] Matteo Magnani, Barbora Micenkova, and Luca Rossi. 2013. Combinatorial Analysis of Multiple Networks. arXiv:1303.4986 https://arxiv.org/abs/1303.4986

[46] Sahar Mazloom and S. Dov Gordon. 2018. Secure Computation with Differentially Private Access Patterns. In *Conference on Computer and Communications Security (CCS)*. ACM, 490–507.

[47] Sahar Mazloom, Phi Hung Le, Samuel Ranellucci, and S. Dov Gordon. 2020. Secure Parallel Computation on National Scale Volumes of Data. In *USENIX Security*. 2487–2504.

[48] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2015. High-Performance and Scalable GPU Graph Traversal. *Transactions on Parallel Computing (TOPC)* 1, 2 (2015), 1–30.

[49] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. 2015. GraphSC: Parallel Secure Computation Made Easy. In *Symposium on Security and Privacy (S&P)*. IEEE, 377–394.

[50] Vanni Noferini and Ryan Wood. 2024. Efficient Computation of Katz Centrality for Very Dense Networks via Negative Parameter Katz. *Journal of Complex Networks* 12, 5 (2024), cnae036.

[51] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. 2021. ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation. In *USENIX Security*. 2165–2182.

[52] Benny Pinkas, Thomas Schneider, Oleksandr Tkachenko, and Avishay Yanai. 2019. Efficient Circuit-Based PSI with Linear Communication. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*. Springer, 122–153.

[53] Matthew J Rattigan and David Jensen. 2005. The Case for Anomalous Link Discovery. *SIGKDD Explorations Newsletter* 7, 2 (2005), 41–47.

[54] Hunter Rehm, Mona Matar, Puck Rombach, and Lauren McIntyre. 2023. The Effect of the Katz Parameter on Node Ranking, with a Medical Application. *Social Network Analysis and Mining* 13, 1 (2023), 132.

[55] M Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M Songhori, Thomas Schneider, and Farinaz Koushanfar. 2018. Chameleon: A Hybrid Secure Computation Framework for Machine Learning Applications. In *ASIA Conference on Computer and Communications Security (ASIACCS)*. ACM, 707–721.

[56] Théo Ryffel, Pierre Tholoniat, David Pointcheval, and Francis Bach. 2022. ARIANN: Low-Interaction Privacy-Preserving Deep Learning via Function Secret Sharing. *Proceedings on Privacy Enhancing Technologies (PoPETs)* 2022, 1 (2022), 291–316.

[57] 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 *Financial Cryptography and Data Security (FC)*. Springer, 605–623.

[58] Chris Stark, Bobby-Joe Breitkreutz, Teresa Reguly, Lorrie Boucher, Ashton Breitkreutz, and Mike Tyers. 2006. BioGRID: A General Repository for Interaction Datasets. *Nucleic Acids Research* 34 (2006), D535–D539.

[59] Longxu Sun, Xin Huang, Zheng Wu, and Jianliang Xu. 2024. Efficient Cross-layer Community Search in Large Multilayer Graphs. In *International Conference on Data Engineering (ICDE)*. IEEE, 2959–2971.

[60] Tamir Tassa and Francesco Bonchi. 2014. Privacy Preserving Estimation of Social Influence. In *International Conference on Extending Database Technology (EDBT)*. OpenProceedings.org, 559–570.

[61] Eugenio Valdano, Luca Ferreri, Chiara Poletto, and Vittoria Colizza. 2015. Analytical Computation of the Epidemic Threshold on Temporal Networks. *Physical Review X* 5, 2 (2015), 021005.

[62] Sara Venturini, Andrea Cristofari, Francesco Rinaldi, and Francesco Tudisco. 2023. Learning the Right Layers a Data-Driven Layer-Aggregation Strategy for Semi-Supervised Learning on Multilayer Graphs. In *International Conference on Machine Learning (ICML)*. PMLR, 35006–35023.

[63] Sameer Wagh, Divya Gupta, and Nishanth Chandran. 2019. SecureNN: 3-Party Secure Computation for Neural Network Training. *Proceedings on Privacy Enhancing Technologies (PoPETs)* 2019, 3 (2019), 26–49.

[64] Chao Wang, Venu Satuluri, and Srinivasan Parthasarathy. 2007. Local Probabilistic Models for Link Prediction. In *International Conference on Data Mining (ICDM)*. IEEE, 322–331.

[65] Dan Wang, Feng Tian, and Daijun Wei. 2023. A New Centrality Ranking Method for Multilayer Networks. *Journal of Computational Science* 66 (2023), 101924.

[66] Run-An Wang, Dandan Liu, and Zhaonian Zou. 2024. FocusCore Decomposition of Multilayer Graphs. In *International Conference on Data Engineering (ICDE)*. IEEE, 2792–2804.

[67] Tianming Zhang, Junkai Fang, Zhengyi Yang, Bin Cao, and Jing Fan. 2024. TATKC: A Temporal Graph Neural Network for Fast Approximate Temporal Katz Centrality Ranking. In *International World Wide Web Conference (WWW)*. ACM, 527–538.

A Protocol Building Blocks

This section provides details on all building blocks used throughout this work. Note that multiple building blocks that were not defined before act as additional sub-building blocks here. As an orientation, Tab. 3 provides an overview of all building blocks introduced or used in §4 and this appendix.

A.1 Shuffling

A fundamental building block to scalable graph analysis is efficient shuffling [5, 35]. For our protocols, we use the shuffling of [35] that matches our setting with some modifications.

The first modification is that we allow the helper $\mathcal{H}$ to know the permutations used for shuffling, saving the computational cost of both parties applying an additional blinding permutation. This does not violate the security of our protocols as these permutations are sampled at random and are used only to mask information revealed to the parties in intermediate steps of the protocol. The helper has no access to this revealed information and does not collude with the parties. Hence, the security proofs of [35] still hold true when using this modification.

In addition, [35] allows to reuse some correlations between different instances of shuffling if the same random permutation is used. We make this explicit, yielding first functionality $\mathcal{F}_{\text{GETSHUFFLE}}$ (Fig. 10) that prepares a random permutation for shuffling and second functionality $\mathcal{F}_{\text{SHUFFLE}}$ (Fig. 11) which then applies a previously prepared permutation. For a random shuffling permutation π, we utilize a new sharing semantic $\ll \pi \gg$ that is defined as follows: P_i holds $\pi_i, \pi_i' \in \mathcal{S}_n$ for $i \in \{0, 1\}$ with $\pi_0, \pi_0', \pi_1, \pi_1'$ being sampled uniformly at random from $\mathcal{S}_n$ subject to $\pi_0 \circ \pi_1 = \pi_1' \circ \pi_0' = \pi$. The helper $\mathcal{H}$ additionally holds all of the mentioned permutations.

Functionality $\mathcal{F}_{\text{GETSHUFFLE}}$

Input: $n \in \mathbb{N}$.
Output: $\ll \pi \gg$ for a randomly sampled $\pi \in \mathcal{S}_n$.

Figure 10: Functionality for generating a new secure shuffle.

Functionality $\mathcal{F}_{\text{SHUFFLE}}$

Input: Secret-shared vector $[[\vec{\mathbf{x}}]]$ of dimension n and $\ll \pi \gg$ for a $\pi \in \mathcal{S}_n$.
Output: Secret-shared vector $[[\pi(\vec{\mathbf{x}})]]$.

Figure 11: Secure shuffling functionality.

$\mathcal{F}_{\text{GETSHUFFLE}}, \mathcal{F}_{\text{SHUFFLE}}$ are implemented using Prot. 3 respectively Prot. 4. Note that the protocols require $\mathcal{H}$ and a party to sample the same random values, which can be done non-interactively using pre-shared PRF keys.

Protocol 3 Secure Shuffle Generation, based on [35]

```
Preprocessing Phase
    For $i \in \{0, 1\}$, $P_i, \mathcal{H}$ sample random $\pi_i \in \mathcal{S}_n$.
    $\mathcal{H}$ sets $\pi \leftarrow \pi_0 \circ \pi_1$.
    $P_0, \mathcal{H}$ sample random $\pi_0' \in \mathcal{S}_n$.
    $\mathcal{H}$ computes $\pi_1' \leftarrow \pi \circ {\pi_0'}^{-1}$ and sends it to $P_1$.
```

If there is a need to *unshuffle*, i.e., applying the inverse of a shuffling permutation, we use $\mathcal{F}_{\text{UNSHUFFLE}}$ (Fig. 12) which we implement by Prot. 5. This is a modification of Prot. 4 that, by swapping and

Table 3: Overview of all used building blocks.

Building Block	Description	Functionality	Protocol
$\ll\pi\gg \leftarrow \mathcal{F}_{\textsc{GetShuffle}}(n)$	prepare new shuffling permutation	Fig. 10 (p. 958)	Prot. 3 (p. 958)
$\ll\omega\circ\pi^{-1}\gg \leftarrow \mathcal{F}_{\textsc{GetMergedShuffle}}(\ll\pi\gg, \ll\omega\gg)$	compose two shuffling permutations	Fig. 14 (p. 960)	Prot. 9 (p. 960)
$[\![\pi(\vec{\mathbf{x}})]\!] \leftarrow \mathcal{F}_{\textsc{shuffle}}([\![\vec{\mathbf{x}}]\!], \ll\pi\gg)$	apply shuffling permutation	Fig. 11 (p. 958)	Prot. 4 (p. 959)
$[\![\pi^{-1}(\vec{\mathbf{x}})]\!] \leftarrow \mathcal{F}_{\textsc{unshuffle}}([\![\vec{\mathbf{x}}]\!], \ll\pi\gg)$	apply inverse shuffling permutation	Fig. 12 (p. 959)	Prot. 5 (p. 959)
$[\![\rho(\vec{\mathbf{x}})]\!] \leftarrow \mathcal{F}_{\textsc{ApplyPerm}}([\![\vec{\mathbf{x}}]\!], [\![\rho]\!])$	apply secret-shared permutation	Fig. 5 (p. 951)	Prot. 6 (p. 959)
$[\![\rho^{-1}(\vec{\mathbf{x}})]\!] \leftarrow \mathcal{F}_{\textsc{reversePerm}}([\![\vec{\mathbf{x}}]\!], [\![\rho]\!])$	apply inverse secret-shared permutation	Fig. 13 (p. 960)	Prot. 7 (p. 960)
$[\![\rho_2(\rho_1^{-1}(\vec{\mathbf{x}}))]\!] \leftarrow \mathcal{F}_{\textsc{swPerm}}([\![\vec{\mathbf{x}}]\!], [\![\rho_1]\!], [\![\rho_2]\!])$	switch from one permutation to another	Fig. 6 (p. 951)	Prot. 10 (p. 960)
$[\![\rho]\!] \leftarrow \mathcal{F}_{\textsc{getCompaction}}([\![\vec{\mathbf{x}}]\!])$	get permutation to sort 0/1-entries	Fig. 15 (p. 961)	[7]
$[\![\rho]\!] \leftarrow \mathcal{F}_{\textsc{sortIteration}}([\![\vec{\mathbf{x}}]\!], [\![\sigma]\!])$	radix sort iteration after applying σ	Fig. 16 (p. 961)	Prot. 11 (p. 961)
$[\![\rho]\!] \leftarrow \mathcal{F}_{\textsc{getSort}}([\![\vec{\mathbf{x}}]\!])$	full radix sort with $\vec{\mathbf{x}}$ bit-decomposed	Fig. 4 (p. 950)	Prot. 12 (p. 961)
$[\![b]\!]_{\text{bin}} \leftarrow \mathcal{F}_{\textsc{eqz}}([\![x]\!])$	equal-zero check, $x = 0 \Leftrightarrow b = 1$	Fig. 17 (p. 962)	circuit in §A.5
$[\![x]\!] \leftarrow \mathcal{F}_{\textsc{bit2A}}([\![x]\!]_{\text{bin}})$	bit-to-arithmetic conversion	Fig. 18 (p. 962)	[51]
$([\![\vec{\mathbf{src}}^d]\!], [\![\vec{\mathbf{dst}}^d]\!]) \leftarrow \mathcal{F}_{\textsc{deduplication}}([\![\vec{\mathbf{src}}]\!], [\![\vec{\mathbf{dst}}]\!])$	set duplicate entries to invalid values	Fig. 19 (p. 962)	Prot. 13 (p. 962)

Protocol 4 Secure Shuffling, based on [35]

```
Preprocessing Phase
For $i \in \{0,1\}$, $P_i, \mathcal{H}$ sample random $\vec{\mathbf{R}}_\mathbf{i} \in \mathbb{Z}_{2^k}^n$.
$\mathcal{H}$ samples random $\vec{\mathbf{R}} \in \mathbb{Z}_{2^k}^n$.
$\mathcal{H}$ computes $\vec{\mathbf{B}}_\mathbf{0} \leftarrow \pi(\vec{\mathbf{R}}_\mathbf{1}) - \vec{\mathbf{R}}$ and sends it to $P_0$.
$\mathcal{H}$ computes $\vec{\mathbf{B}}_\mathbf{1} \leftarrow \pi(\vec{\mathbf{R}}_\mathbf{0}) + \vec{\mathbf{R}}$ and sends it to $P_1$.
Online Phase
$P_0$ computes $\vec{\mathbf{t}}_\mathbf{0} \leftarrow \pi'_0(\vec{\mathbf{x}}_\mathbf{0} + \vec{\mathbf{R}}_\mathbf{0})$ and sends it to $P_1$.
$P_1$ computes $\vec{\mathbf{t}}_\mathbf{1} \leftarrow \pi_1(\vec{\mathbf{x}}_\mathbf{1} + \vec{\mathbf{R}}_\mathbf{1})$ and sends it to $P_0$.
$P_0$ computes $\vec{\mathbf{y}}_\mathbf{0} \leftarrow \pi_0(\vec{\mathbf{t}}_\mathbf{1}) - \vec{\mathbf{B}}_\mathbf{0}$.
$P_1$ computes $\vec{\mathbf{y}}_\mathbf{1} \leftarrow \pi'_1(\vec{\mathbf{t}}_\mathbf{0}) - \vec{\mathbf{B}}_\mathbf{1}$.
Return $[\![\vec{\mathbf{y}}]\!]$.
```

Protocol 5 Secure Unshuffling

```
Preprocessing Phase
For $i \in \{0,1\}$, $P_i, \mathcal{H}$ sample random $\vec{\mathbf{R}}_\mathbf{i} \in \mathbb{Z}_{2^k}^n$.
$\mathcal{H}$ samples random $\vec{\mathbf{R}} \in \mathbb{Z}_{2^k}^n$.
$\mathcal{H}$ computes $\vec{\mathbf{B}}_\mathbf{0} \leftarrow \pi^{-1}(\vec{\mathbf{R}}_\mathbf{1}) - \vec{\mathbf{R}}$ and sends it to $P_0$.
$\mathcal{H}$ computes $\vec{\mathbf{B}}_\mathbf{1} \leftarrow \pi^{-1}(\vec{\mathbf{R}}_\mathbf{0}) + \vec{\mathbf{R}}$ and sends it to $P_1$.
Online Phase
$P_0$ computes $\vec{\mathbf{t}}_\mathbf{0} \leftarrow {\pi_0}^{-1}(\vec{\mathbf{x}}_\mathbf{0} + \vec{\mathbf{R}}_\mathbf{0})$ and sends it to $P_1$.
$P_1$ computes $\vec{\mathbf{t}}_\mathbf{1} \leftarrow {\pi'}_1^{-1}(\vec{\mathbf{x}}_\mathbf{1} + \vec{\mathbf{R}}_\mathbf{1})$ and sends it to $P_0$.
$P_0$ computes $\vec{\mathbf{y}}_\mathbf{0} \leftarrow {\pi'}_0^{-1}(\vec{\mathbf{t}}_\mathbf{1}) - \vec{\mathbf{B}}_\mathbf{0}$.
$P_1$ computes $\vec{\mathbf{y}}_\mathbf{1} \leftarrow {\pi_1}^{-1}(\vec{\mathbf{t}}_\mathbf{0}) - \vec{\mathbf{B}}_\mathbf{1}$.
Return $[\![\vec{\mathbf{y}}]\!]$.
```

inverting permutations, applies π^{-1} instead of π:

$$\begin{aligned}
\vec{\mathbf{y}} &= {\pi'}_0^{-1}(\vec{\mathbf{t}}_\mathbf{1}) - \vec{\mathbf{B}}_\mathbf{0} + {\pi_1}^{-1}(\vec{\mathbf{t}}_\mathbf{0}) - \vec{\mathbf{B}}_\mathbf{1} \\
&= {\pi'}_0^{-1}({\pi'}_1^{-1}(\vec{\mathbf{x}}_\mathbf{1} + \vec{\mathbf{R}}_\mathbf{1})) - \vec{\mathbf{B}}_\mathbf{0} + {\pi_1}^{-1}({\pi_0}^{-1}(\vec{\mathbf{x}}_\mathbf{0} + \vec{\mathbf{R}}_\mathbf{0})) - \vec{\mathbf{B}}_\mathbf{1} \\
&= \pi^{-1}(\vec{\mathbf{x}}) + \pi^{-1}(\vec{\mathbf{R}}_\mathbf{1}) + \pi^{-1}(\vec{\mathbf{R}}_\mathbf{0}) - \pi^{-1}(\vec{\mathbf{R}}_\mathbf{1}) - \vec{\mathbf{R}} - \pi^{-1}(\vec{\mathbf{R}}_\mathbf{0}) + \vec{\mathbf{R}} \\
&= \pi^{-1}(\vec{\mathbf{x}})
\end{aligned}$$

Functionality $\mathcal{F}_{\textsc{unshuffle}}$

Input: Secret-shared vector $[\![\vec{\mathbf{x}}]\!]$ of dimension n and $\ll\pi\gg$ for a $\pi \in \mathcal{S}_n$.
Output: Secret-shared vector $[\![\pi^{-1}(\vec{\mathbf{x}})]\!]$.

Figure 12: Secure unshuffling functionality, inverse of $\mathcal{F}_{\textsc{shuffle}}$.

Complexity. Instantiating $\mathcal{F}_{\textsc{GetShuffle}}$ costs n ring elements of communication in preprocessing. Instantiating $\mathcal{F}_{\textsc{shuffle}}$ or $\mathcal{F}_{\textsc{unshuffle}}$ each costs $2n$ ring elements communication in preprocessing, n ring elements online communication per party, and a single round in the online phase.

A.2 Applying Permutations

Another essential building block of our protocols is the application of a secret-shared permutation to secret-shared data. This is achieved by $\mathcal{F}_{\textsc{ApplyPerm}}$ (Fig. 5 in §4.2.2), which is efficiently implemented in [7] for the three-party setting. We translate it to our setting, also including optimizations in case the same permutation[21] is applied to different data vectors. The resulting protocol is given in Prot. 6, correctness and security are as in [7].

Protocol 6 Secure Application of a Secret Permutation, based on [7]

```
If $[\![\rho]\!]$ has not been used by instance of $\mathcal{F}_{\textsc{ApplyPerm}}/\mathcal{F}_{\textsc{reversePerm}}$ before
    $\ll\pi\gg \leftarrow \mathcal{F}_{\textsc{GetShuffle}}(n)$
    $[\![\pi(\rho)]\!] \leftarrow \mathcal{F}_{\textsc{shuffle}}([\![\rho]\!], \ll\pi\gg)$
    Open $\pi(\rho) = \rho \circ \pi^{-1}$ // Observation 2.4 in [7]
Always (use $\ll\pi\gg, \pi(\rho)$ from last instantiation with $[\![\rho]\!]$ if exists)
    $[\![\pi(\vec{\mathbf{x}})]\!] \leftarrow \mathcal{F}_{\textsc{shuffle}}([\![\vec{\mathbf{x}}]\!], \ll\pi\gg)$
    $[\![\rho(\vec{\mathbf{x}})]\!] \leftarrow (\rho \circ \pi^{-1})([\![\pi(\vec{\mathbf{x}})]\!])$
```

Based on this protocol, we also implement another functionality $\mathcal{F}_{\textsc{reversePerm}}$ (Fig. 13) applying the inverse of a secret-shared permutation in Prot. 7. Regarding security, the same arguments as for

[21] Same as in the same variable/wire, not as in different variables/wires with the same (but secret) value

Prot. 6 apply as the same masked data is revealed. Considering correctness, note that $\pi^{-1}((\rho \circ \pi^{-1})^{-1}(\vec{\mathbf{x}})) = (\pi^{-1} \circ \pi \circ \rho^{-1})(\vec{\mathbf{x}}) = \rho^{-1}(\vec{\mathbf{x}})$.

Functionality $\mathcal{F}_{\textsc{reversePerm}}$

Input: Secret-shared vector $[\![\vec{\mathbf{x}}]\!]$ of dimension n, $[\![\rho]\!]$ for a permutation $\rho \in \mathcal{S}_n$.
Output: Secret-shared vector $[\![\rho^{-1}(\vec{\mathbf{x}})]\!]$.

Figure 13: Functionality to apply the inverse of a secret-shared permutation to a secret-shared vector.

Protocol 7 Secure Inverse Application of a Secret Permutation

```
If $[\![\rho]\!]$ has not been used by instance of $\mathcal{F}_{\textsc{applyPerm}}/\mathcal{F}_{\textsc{reversePerm}}$ before
    $\ll \pi \gg \leftarrow \mathcal{F}_{\textsc{getShuffle}}(n)$
    $[\![\pi(\rho)]\!] \leftarrow \mathcal{F}_{\textsc{shuffle}}([\![\rho]\!], \ll \pi \gg)$
    Open $\pi(\rho) = \rho \circ \pi^{-1}$ // Observation 2.4 in [7]
Always (use $\ll \pi \gg, \pi(\rho)$ from last instantiation with $[\![\rho]\!]$ if exists)
    $[\![\pi(\rho^{-1}(\vec{\mathbf{x}}))]\!] \leftarrow (\rho \circ \pi^{-1})^{-1}([\![\vec{\mathbf{x}}]\!])$
    $[\![\rho^{-1}(\vec{\mathbf{x}})]\!] \leftarrow \mathcal{F}_{\textsc{unshuffle}}([\![\pi(\rho^{-1}(\vec{\mathbf{x}}))]\!], \ll \pi \gg)$
```

Complexity. Regarding the instantiation of $\mathcal{F}_{\textsc{applyPerm}}$: If $[\![\rho]\!]$ has been used before: $2n$ ring elements communication in preprocessing, n ring elements per party online (due to $\mathcal{F}_{\textsc{shuffle}}$), and 1 online round as the final step runs locally. If $[\![\rho]\!]$ is used the first time, $3n$ ring elements communication in preprocessing, $2n$ ring elements per party online, and one online round more (from $\mathcal{F}_{\textsc{getShuffle}}, \mathcal{F}_{\textsc{shuffle}}$ and opening n elements, but both instances of $\mathcal{F}_{\textsc{shuffle}}$ can be run in parallel). The complexity for $\mathcal{F}_{\textsc{reversePerm}}$ is the same where $\mathcal{F}_{\textsc{applyPerm}}$ and $\mathcal{F}_{\textsc{reversePerm}}$ have cost savings after any of them used $[\![\rho]\!]$ once.

A.3 Switching Between Permutations

Recall from Prot. 1 (§4.2) that besides $\mathcal{F}_{\textsc{apply}}$ if applicable, the only interaction in message-passing iterations stems from switching between permutations, emphasizing the importance of optimizing this operation. To change the ordering of some secret-shared vector from permutation ρ_1 to ρ_2, we use functionality $\mathcal{F}_{\textsc{swPerm}}$ (Fig. 6 in §4.2.2). Functionality $\mathcal{F}_{\textsc{swPerm}}$ could be simply implemented by composing $\mathcal{F}_{\textsc{reversePerm}}$ with input ρ_1 and $\mathcal{F}_{\textsc{applyPerm}}$ with input ρ_2 which would result in Prot. 8.

Protocol 8 Secure Permutation Switching, first attempt

```
Obtain $\ll \pi \gg, \ll \omega \gg, \pi(\rho_1) = \rho_1 \circ \pi^{-1}, \sigma(\rho_2) = \rho_2 \circ \sigma^{-1}$ for $\pi, \omega \in \mathcal{S}_n$ as in Prot. 6.
$[\![\pi(\rho_1^{-1}(\vec{\mathbf{x}}))]\!] \leftarrow (\rho_1 \circ \pi^{-1})^{-1}([\![\vec{\mathbf{x}}]\!])$
$[\![\rho_1^{-1}(\vec{\mathbf{x}})]\!] \leftarrow \mathcal{F}_{\textsc{unshuffle}}([\![\pi(\rho_1^{-1}(\vec{\mathbf{x}}))]\!], \ll \pi \gg)$
$[\![\omega(\rho_1^{-1}(\vec{\mathbf{x}}))]\!] \leftarrow \mathcal{F}_{\textsc{shuffle}}([\![\rho_1^{-1}(\vec{\mathbf{x}})]\!], \ll \omega \gg)$
$[\![\rho_2(\rho_1^{-1}(\vec{\mathbf{x}}))]\!] \leftarrow (\rho_2 \circ \omega^{-1})([\![\omega(\rho_1^{-1}(\vec{\mathbf{x}}))]\!])$
```

Observe that the only interactive steps (except when the permutations are used for the first time) are unshuffling and shuffling in direct succession. This can be optimized by shuffling only once with a composed permutation $\ll \omega \circ \pi^{-1} \gg$. Utilizing the sharing semantics of $\ll \cdot \gg$, specifically that $\mathcal{H}$ knows the entire permutation, allows to implement the shuffle composition functionality $\mathcal{F}_{\textsc{getMergedShuffle}}$ (Fig. 14) as provided in protocol Prot. 9. A similar shuffle composition is used in [35] but not provided as explicit functionality/protocol there which we provide here. Regarding correctness, note that $\sigma_0 \circ \sigma_1 = \sigma_1' \circ \sigma_0' = \omega \circ \pi^{-1}$. Furthermore, regarding security, note that each message sent by $\mathcal{H}$ is perfectly masked by a randomly sampled permutation only known to $\mathcal{H}$ and the other respective party. The final protocol for $\mathcal{F}_{\textsc{swPerm}}$ is provided in Prot. 10 with correctness and security directly following from the prior considerations and the properties of $\mathcal{F}_{\textsc{applyPerm}}$ and $\mathcal{F}_{\textsc{reversePerm}}$.

Functionality $\mathcal{F}_{\textsc{getMergedShuffle}}$

Input: $\ll \pi \gg, \ll \omega \gg$ for $\pi, \omega \in \mathcal{S}_n$.
Output: $\ll \sigma \gg$ for $\sigma = \omega \circ \pi^{-1}$.

Figure 14: Functionality for composing two secure shuffles.

Protocol 9 Secure Shuffle Composition

```
Preprocessing Phase
    $P_0, \mathcal{H}$ sample random $\sigma_0 \in \mathcal{S}_n$; $P_1, \mathcal{H}$ sample random $\sigma_1' \in \mathcal{S}_n$
    $\mathcal{H}$ computes $\sigma_0' \leftarrow \sigma_1'^{-1} \circ \omega \circ \pi^{-1}$ and sends it to $P_0$.
    $\mathcal{H}$ computes $\sigma_1 \leftarrow \sigma_0^{-1} \circ \omega \circ \pi^{-1}$ and sends it to $P_1$.
```

Protocol 10 Secure Permutation Switching, optimized version

```
If $[\![\rho_1]\!]$ has not been used by instance of $\mathcal{F}_{\textsc{applyPerm}}/\mathcal{F}_{\textsc{reversePerm}}/\mathcal{F}_{\textsc{swPerm}}$ before
    $\ll \pi \gg \leftarrow \mathcal{F}_{\textsc{getShuffle}}(n)$
    $[\![\pi(\rho_1)]\!] \leftarrow \mathcal{F}_{\textsc{shuffle}}([\![\rho_1]\!], \ll \pi \gg)$
    Open $\pi(\rho_1) = \rho_1 \circ \pi^{-1}$ // Observation 2.4 in [7]
If $[\![\rho_2]\!]$ has not been used by instance of $\mathcal{F}_{\textsc{applyPerm}}/\mathcal{F}_{\textsc{reversePerm}}/\mathcal{F}_{\textsc{swPerm}}$ before
    $\ll \omega \gg \leftarrow \mathcal{F}_{\textsc{getShuffle}}(n)$
    $[\![\omega(\rho_2)]\!] \leftarrow \mathcal{F}_{\textsc{shuffle}}([\![\rho_2]\!], \ll \omega \gg)$
    Open $\omega(\rho_2) = \rho_2 \circ \omega^{-1}$ // Observation 2.4 in [7]
If $[\![\rho_1]\!], [\![\rho_2]\!]$ have not been used in this combination by instance of $\mathcal{F}_{\textsc{swPerm}}$ before
    $\ll \omega \circ \pi^{-1} \gg \leftarrow \mathcal{F}_{\textsc{getMergedShuffle}}(\ll \pi \gg, \ll \omega \gg)$
Always (use $\ll \pi \gg, \ll \omega \gg, \ll \omega \circ \pi^{-1} \gg, \pi(\rho_1), \omega(\rho_2)$ from prior existing instantiations if available)
    $[\![\pi(\rho_1^{-1}(\vec{\mathbf{x}}))]\!] \leftarrow (\rho_1 \circ \pi^{-1})^{-1}([\![\vec{\mathbf{x}}]\!])$
    $[\![\omega(\rho_1^{-1}(\vec{\mathbf{x}}))]\!] \leftarrow \mathcal{F}_{\textsc{shuffle}}([\![\pi(\rho_1^{-1}(\vec{\mathbf{x}}))]\!], \ll \omega \circ \pi^{-1} \gg)$
    $[\![\rho_2(\rho_1^{-1}(\vec{\mathbf{x}}))]\!] \leftarrow (\rho_2 \circ \omega^{-1})([\![\omega(\rho_1^{-1}(\vec{\mathbf{x}}))]\!])$
```

Communication. Our instantiation of $\mathcal{F}_{\textsc{swPerm}}$, if used for the same permutations before, costs $2n$ ring elements communication in preprocessing, n ring elements communication per party online, and a single round which it directly inherits from $\mathcal{F}_{\textsc{shuffle}}$. For an instantiation with $\ll \rho_1 \gg$ or $\ll \rho_2 \gg$ never used before in $\mathcal{F}_{\textsc{applyPerm}}$, $\mathcal{F}_{\textsc{reversePerm}}$, or $\mathcal{F}_{\textsc{swPerm}}$ there is an additionally cost of $3n$ ring elements communication in preprocessing, $2n$ ring elements communication per party online and one round per new permutation (rounds may be parallelized if multiple). There also is an additional cost of $2n$ ring elements communication in preprocessing the first time that $\mathcal{F}_{\text{swPerm}}$ is instantiated with two specific permutations. A naïve implementation simply using $\mathcal{F}_{\text{reversePerm}}$ and $\mathcal{F}_{\text{applyPerm}}$ would instead have double the cost in all metrics if repeated; only the additional cost for the first instantiation would be lower by $2n$ ring elements in preprocessing otherwise used for $\mathcal{F}_{\text{getMergedShuffle}}$.

A.4 Sorting

To instantiate our protocols, we need access to functionality $\mathcal{F}_{\text{getSort}}$ generating a permutation that is necessary to apply to bring data into a sorted order. This is formalized as $\mathcal{F}_{\text{getSort}}$ (Fig. 4 in §4.2.1), and the resulting permutation can then be applied for the final sorting using $\mathcal{F}_{\text{applyPerm}}$ (cf. §A.2).

We implement such stable sorting based on the radix sort construction of [7]. It assumes the input elements to be decomposed into their bit representation, i.e., in the format $(\llbracket \vec{\mathbf{x}}^{\mathbf{k-1}} \rrbracket, \ldots, \llbracket \vec{\mathbf{x}}^{\mathbf{0}} \rrbracket)$ where $x_i = \sum_{j=0}^{k-1} x_i^j \cdot 2^j$. The decomposition sharings still are assumed to be over $\mathbb{Z}_{2^k}$ and can be obtained, e.g., by running conversion protocols as in [22]. We also use notation $\mathcal{F}_{\text{getSort}}(\llbracket \vec{\mathbf{x}} \rrbracket, \llbracket \vec{\mathbf{y}} \rrbracket)$ to indicate that we sort elements $x_i || y_i$ where concatenation of the components is trivial in decomposed format.

As one atomic step of sorting, [7] computes the permutation to sort a vector $\vec{\mathbf{x}}$ where all entries are 0 or 1 which we formalize as $\mathcal{F}_{\text{getCompaction}}$ (cf. Fig. 15). They show how $\mathcal{F}_{\text{getCompaction}}$ can be instantiated using n multiplications in parallel.

Functionality $\mathcal{F}_{\text{getCompaction}}$

Input: Secret-shared vector $\llbracket \vec{\mathbf{x}} \rrbracket$ of dimension n with $x_i \in \{0,1\}$ $\forall 1 \leq i \leq n$.
Output: Secret-shared permutation $\llbracket \rho \rrbracket$ with $\rho \in \mathcal{S}_n$ and $\rho(\vec{\mathbf{x}})$ being stably sorted in ascending order.

Figure 15: Functionality to retrieve permutation to sort data with only $0/1$**-entries.**

Sorting in [7] is constructed from multiple iterations that we formalize as $\mathcal{F}_{\text{sortIteration}}$ (Fig. 16). For completeness, we provide its instantiation in Prot. 11 that follows from [7], including their proposed optimizations. Note that $\rho = \sigma' \circ \sigma \circ \pi^{-1} \circ (\pi^{-1})^{-1} = \sigma' \circ \sigma$ by Observation 2.4 of [7]. As σ' stably sorts $\sigma(\vec{\mathbf{x}})$, correctness immediately follows.

Functionality $\mathcal{F}_{\text{sortIteration}}$

Input: Secret-shared vector $\llbracket \vec{\mathbf{x}} \rrbracket$ of dimension n with $x_i \in \{0,1\}$ $\forall 1 \leq i \leq n$, $\llbracket \sigma \rrbracket$ with $\sigma \in \mathcal{S}_n$.
Output: Secret-shared permutation $\llbracket \rho \rrbracket$ with $\rho \in \mathcal{S}_n$ and $\rho(\vec{\mathbf{x}})$ being a stably sorted version in ascending order of $\sigma(\vec{\mathbf{x}})$.

Figure 16: Iteration of sorting from [7]

Now, we can give an instantiation of $\mathcal{F}_{\text{getSort}}$ in Prot. 12 which is a restructured version of its original in [7]. Here, we denote the j'th bits of all entries of $\vec{\mathbf{x}}$ by $\vec{\mathbf{x}}^{\mathbf{j}}$. Note how the protocol precisely displays the structure of a plaintext radix sort: In iteration j, we compute a permutation of the j'th bits to sort them while maintaining for equal bits their order resulting from the prior permutation necessary to sort w.r.t. bits $j-1$ to 0.

Protocol 11 Secure Sorting Iteration, based on [7]

```
$\ll \pi \gg \leftarrow \mathcal{F}_{\text{getShuffle}}(n)$
$\llbracket \pi(\sigma) \rrbracket \leftarrow \mathcal{F}_{\text{shuffle}}(\llbracket \sigma \rrbracket, \ll \pi \gg)$
Open $\pi(\sigma) = \sigma \circ \pi^{-1}$ // Observation 2.4 in [7]
$\llbracket \pi(\vec{\mathbf{x}}) \rrbracket \leftarrow \mathcal{F}_{\text{shuffle}}(\llbracket \vec{\mathbf{x}} \rrbracket, \ll \pi \gg)$
$\llbracket \sigma(\vec{\mathbf{x}}) \rrbracket \leftarrow (\sigma \circ \pi^{-1})(\llbracket \pi(\vec{\mathbf{x}}) \rrbracket)$
$\llbracket \sigma' \rrbracket \leftarrow \mathcal{F}_{\text{getCompaction}}(\llbracket \sigma(\vec{\mathbf{x}}) \rrbracket)$
$\llbracket \sigma' \circ \sigma \circ \pi^{-1} \rrbracket \leftarrow (\sigma \circ \pi^{-1})^{-1}(\llbracket \sigma' \rrbracket)$ // Observation 2.4 in [7]
$\llbracket \rho \rrbracket \leftarrow \mathcal{F}_{\text{unshuffle}}(\llbracket \sigma' \circ \sigma \circ \pi^{-1} \rrbracket, \ll \pi \gg)$
```

Protocol 12 Secure Computation of Sorting Permutation, restructured from [7]

```
$\llbracket \rho \rrbracket \leftarrow \mathcal{F}_{\text{getCompaction}}(\llbracket \vec{\mathbf{x}}^{\mathbf{0}} \rrbracket)$
for $j = 1, \ldots, k-1$ do
    $\llbracket \rho \rrbracket \leftarrow \mathcal{F}_{\text{sortIteration}}(\llbracket \vec{\mathbf{x}}^{\mathbf{j}} \rrbracket, \llbracket \rho \rrbracket)$
end for
```

Complexity. Instantiating $\mathcal{F}_{\text{getCompaction}}$ as in [7] costs n ring elements communication in preprocessing and $2n$ ring elements communication per party in a single round online. Each instantiation of $\mathcal{F}_{\text{sortIteration}}$ costs $8n$ ring elements in preprocessing, $6n$ ring elements per party online, and 4 rounds (note that both instances of $\mathcal{F}_{\text{shuffle}}$ can be parallelized). Hence, generating the full sorting permutation costs $8kn - 7n$ ring elements in preprocessing, $6kn - 4n$ ring elements per party online, and $4k - 3$ online rounds.

Optimizations. We note that if the sorting keys do not utilize the whole domain $\mathbb{Z}_{2^k}$, we can optimize the sorting further. If we sort by nodes and assign ids $0, 1, \ldots, |\mathsf{V}| - 1$ to them, only the $\lceil \log_2 |\mathsf{V}| \rceil$ lowest bits are not zero. Hence, we can decrease from $k-1$ iterations down to $\lceil \log_2 |\mathsf{V}| \rceil - 1$ iterations.

As already sketched in §4.2.1, we exploit correlations between the permutations to sort by concatenated keys $\mathsf{src}||(1 - \mathsf{isV})$, $\mathsf{dst}||\mathsf{isV}$, and $1 - \mathsf{isV}||\mathsf{src}||(1 - \mathsf{isV})$. We compute the first two permutations using the aforementioned approach on $\lceil \log_2 |\mathsf{V}| \rceil + 1$ bit keys. To compute the third permutation, it suffices to use that for $\mathsf{src}||(1 - \mathsf{isV})$ and append one more radix sort iteration for adding $1 - \mathsf{isV}$ as the new most significant bit. Hence, computing the permutation for vertex ordering only requires an additional $8n$ ring elements in preprocessing, $6n$ ring elements per party online, and 4 rounds instead of running another full-fletched radix sort.

Assuming that $\vec{\mathbf{x}}$ is not provided in a bit-decomposed fashion, we could prepend decomposition protocols as in [22]. Yet, we opt for instead letting the input parties provide their inputs for src, dst already in a bit-decomposed fashion which increases the input size by a factor of $\lceil \log_2 |\mathsf{V}| \rceil$, but overall is significantly cheaper than running decomposition protocols. In a scenario where the input parties' network connection is significantly worse than the network between the computing parties, reducing the input data size and instead running a decomposition protocol might be the better choice.

A.5 Clipping ($\mathsf{clip}(\cdot)$)

For the clipping operation for π_{R}^{D} (§3.3) as well as for secure deduplication (§A.6), we need to check if a secret-shared value is equal to zero. To this end, we first use functionality $\mathcal{F}_{\text{EQZ}}$ (Fig. 17) computing a binary sharings that represents 1 if the input is 0. Furthermore, as most of our computation is carried out over the arithmetic domain, we use $\mathcal{F}_{\text{BIT2A}}$ (Fig. 18) to convert single binary sharings back to arithmetic ones.

Functionality $\mathcal{F}_{\text{EQZ}}$

Input: $[\![x]\!]$ with $x \in \mathbb{Z}_{2^k}$.
Output: $[\![1]\!]_{\text{bin}}$ if $x = 0$, $[\![0]\!]_{\text{bin}}$ otherwise.

Figure 17: Equal-Zero Check

Functionality $\mathcal{F}_{\text{BIT2A}}$

Input: $[\![x]\!]_{\text{bin}}$ with $x \in \{0, 1\}$.
Output: $[\![x]\!]$.

Figure 18: Bit-to-Arithmetic Conversion

For $\mathcal{F}_{\text{BIT2A}}$, we take the conversion technique from [51] where the parties secret share their shares x_0, x_1 in the arithmetic domain and then compute $[\![x]\!] = [\![x_0]\!] + [\![x_1]\!] - 2[\![x_0]\!][\![x_1]\!]$. For $\mathcal{F}_{\text{EQZ}}$, we observe that $x = 0 \Leftrightarrow x_0 + x_1 = 0 \Leftrightarrow x_0 = -x_1$. We let the parties decompose x_0 respectively $-x_1$ into bits which they then share in the binary domain as $[\![(x_0)_{k-1}]\!]_{\text{bin}}, \ldots, [\![(x_0)_0]\!]_{\text{bin}}$ respectively $[\![(-x_1)_{k-1}]\!]_{\text{bin}}, \ldots, [\![(-x_1)_0]\!]_{\text{bin}}$. Then, they check if $x_0 = -x_1$ by running a binary equality circuit: $x_0 = -x_1 \Leftrightarrow ((x_1)_{k-1} = (-x_2)_{k-1}) \wedge \cdots \wedge ((x_1)_0 = (-x_2)_0) \Leftrightarrow \overline{(x_1)_{k-1} \oplus (-x_2)_{k-1}} \wedge \cdots \wedge \overline{(x_1)_0 \oplus (-x_2)_0}$.

Complexity. Note that a server can share values non-interactively by both servers sampling the other server's share using a pre-shared PRF key. Our instantiation of $\mathcal{F}_{\text{EQZ}}$ requires $k - 1$ bits $= 1 - 1/k$ ring elements communication in preprocessing, $2k - 2$ bits $= 2 - 2/k$ ring elements per party online, and $\lceil \log_2 k \rceil$ online rounds.[22] The instantiation for $\mathcal{F}_{\text{BIT2A}}$ comes at the cost of one multiplication, i.e., 1 ring element communication in preprocessing, 2 ring elements per party online, and one round.

A.6 Deduplication

For computing π^D_{K}, we rely on deduplication of the edges, i.e., removing duplicates so that each specific edge only remains in the DAG list once. To not leak the number of duplicates, we opt to not completely remove entries but instead mark them so that they do not influence the later computation. Our strategy for that is to set the least significant bit among the bits that are 0 for all DAG list entries (i.e., $x_{\lceil \log_2 |\mathsf{V}| \rceil}$) to 1 for all duplicates (assuming node labels $0, 1, \ldots, |\mathsf{V}| - 1$). This will force the duplicates to be at the end of the DAG list in source order, destination order, and vertex order, where they have no effect on any of the other active entries as per the design of [35]. Our procedure for deduplication $\mathcal{F}_{\text{DEDUPLICATION}}$ (Fig. 19) is given in Prot. 13, which is based on ideas in [30].

Functionality $\mathcal{F}_{\text{DEDUPLICATION}}$

Input: Secret-shared vectors $[\![\vec{\mathbf{src}}]\!]$, $[\![\vec{\mathbf{dst}}]\!]$ of dimension $n = |\mathsf{V}| + |\mathcal{E}|$.
Output: $[\![\vec{\mathbf{src}}^d]\!]$, $[\![\vec{\mathbf{dst}}^d]\!]$ with $\text{src}^d_i = \text{src}_i + 2^{\lceil \log_2 |\mathsf{V}| \rceil}$, $\text{dst}^d_i = \text{dst}_i + 2^{\lceil \log_2 |\mathsf{V}| \rceil}$ if $\exists j < i : \text{src}_j = \text{src}_i \wedge \text{dst}_j = \text{dst}_i$ and $\text{src}^d_i = \text{src}_i$, $\text{dst}^d_i = \text{dst}_i$ otherwise, for $1 \le i \le |\mathsf{V}| + |\mathcal{E}|$.

Figure 19: Functionality for Deduplication

Protocol 13 Secure Deduplication, based on [30]

```
$[\![\rho]\!] \leftarrow \mathcal{F}_{\text{GETSORT}}([\![\vec{\mathbf{src}}]\!], [\![\vec{\mathbf{dst}}]\!])$
$[\![\vec{\mathbf{src}}']\!] \leftarrow \mathcal{F}_{\text{APPLYPERM}}([\![\rho]\!], [\![\vec{\mathbf{src}}]\!])$
$[\![\vec{\mathbf{dst}}']\!] \leftarrow \mathcal{F}_{\text{APPLYPERM}}([\![\rho]\!], [\![\vec{\mathbf{dst}}]\!])$ // DAG list is now ordered so that duplicates are grouped together
for $i = 2, \ldots, n = |\mathsf{V}| + |\mathcal{E}|$ do
    $[\![\delta'_1]\!] \leftarrow [\![\text{src}'_i]\!] - [\![\text{src}'_{i-1}]\!]$ // 0 iff same as predecessor
    $[\![\delta'_2]\!] \leftarrow [\![\text{dst}'_i]\!] - [\![\text{dst}'_{i-1}]\!]$
    $[\![\Delta'_1]\!]_{\text{bin}} \leftarrow \mathcal{F}_{\text{EQZ}}([\![\delta'_1]\!])$ // 1 if same as predecessor
    $[\![\Delta'_2]\!]_{\text{bin}} \leftarrow \mathcal{F}_{\text{EQZ}}([\![\delta'_2]\!])$ // 0 otherwise
    $[\![\Delta']\!]_{\text{bin}} \leftarrow [\![\Delta'_1]\!]_{\text{bin}} \wedge [\![\Delta'_2]\!]_{\text{bin}}$
    $[\![\Delta']\!] \leftarrow \mathcal{F}_{\text{BIT2A}}([\![\Delta']\!]_{\text{bin}})$ // 1 if duplicate
end for
$[\![\Delta]\!] \leftarrow \mathcal{F}_{\text{REVERSEPERM}}([\![\rho]\!], [\![\Delta']\!])$ // back to original order
$[\![\vec{\mathbf{src}}^d]\!] \leftarrow [\![\vec{\mathbf{src}}]\!] + (2^{\lceil \log_2 |\mathsf{V}| \rceil}) \cdot [\![\Delta]\!]$
$[\![\vec{\mathbf{dst}}^d]\!] \leftarrow [\![\vec{\mathbf{dst}}]\!] + (2^{\lceil \log_2 |\mathsf{V}| \rceil}) \cdot [\![\Delta]\!]$
```

Optimizations. Recall that the improvements from §A.4 require the entries of $\vec{\mathbf{src}}, \vec{\mathbf{dst}}$ to be bit-decomposed. As this would cause a significant overhead when reordering these vectors by $\mathcal{F}_{\text{APPLYPERM}}$, we here instead compose the single bits back using non-interactive linear combinations $\sum_{j=0}^{k-1} [\![x^j_i]\!] \cdot 2^j$ for each vector entry x_i.

Furthermore, we consider the deduplication in the context of the overall graph analysis that, as discussed in §A.4, later computes permutations to sort by keys $\text{src}^d||(1 - \text{isV})$, $\text{dst}^d||\text{isV}$, and $1 - \text{isV}||\text{src}^d||(1 - \text{isV})$ after deduplication. Instead of sorting by keys src||dst here, the protocol's correctness is also preserved when sorting by src||dst||isV instead. This allows to exploit a correlation to the later required permutation ordering for destination order $\text{dst}^d||\text{isV}$: When sorting by src||dst||isV, we keep the radix sort intermediate result considering dst||isV to then alternatively appending one final iteration sorting by Δ describing exactly the new most significant non-zero bit in dst^d, also yielding the permutation for sorting by $\text{dst}^d||\text{isV}$.

Complexity. Each iteration of the loop in Prot. 13 costs $3 - 1/k$ ring elements communication in preprocessing, $6 - 2/k$ ring elements communication per party online, and $\lceil \log_2 k \rceil + 2$ rounds. Then, the overall *standalone* complexity of the protocol for $n = |\mathsf{V}| + |\mathcal{E}|$, considering the sorting optimizations from §A.4, is

- $(16n \cdot \lceil \log_2 |\mathsf{V}| \rceil - 7n) + 7n + (n - 1) \cdot (3 - 1/k) + 2n = 16n \cdot \lceil \log_2 |\mathsf{V}| \rceil + (5 - 1/k)n - 3 + 1/k$ ring elements communication in preprocessing,
- $(12n \cdot \lceil \log_2 |\mathsf{V}| \rceil - 4n) + 4n + (n - 1) \cdot (6 - 2/k) + n = 12n \cdot \lceil \log_2 |\mathsf{V}| \rceil + (7 - 2/k)n - 6 + 2/k$ ring elements communication per party online,
- and $8\lceil \log_2 |\mathsf{V}| \rceil - 3 + 2 + \lceil \log_2 k \rceil + 2 + 1 = 8\lceil \log_2 |\mathsf{V}| \rceil + \lceil \log_2 k \rceil + 2$ rounds.

[22] Our implementation packs these bits into 32 bit integers, not fully utilizing their capacity, leading to slightly worse actual performance than what is theoretically given.

B Full Protocol Details

This section contains details regarding our overall computation of centrality measures from §4. §B.1 provides additional information about the vertex-centric computation for π^D_{R}, followed by the formal specification of the complete protocols for all centrality measures in §B.2. §B.3 analyzes the exact and concrete communication cost of our protocols, and §B.4 elaborates on how even the input sizes could be hidden from the parties if needed.

B.1 Vertex-Centric Approach for Realizing π^D_{R}

In a vertex-centric approach, Eq. (7) is computed as follows: Each node w propagates s^{i-1}_w on its outgoing edges, following which it gathers the propagated data on all its incoming edges. Taking as input the prior state $x = s^{i-1}_w$ and the collected data $y = \sum_{(u,w)\in\mathcal{E}} s^{i-1}_u$, node w sets $s^i_w = \text{clip}(x + y)$. After D iterations, we compute the sum over s^D_w for all $w \in \mathsf{V}$ to get the number of reached nodes $\pi^D_{\mathsf{R}}(v)$ for a given source node v. If we wish to compute $\pi^D_{\mathsf{R}}(v)$ for all nodes v, we compute BFS for all v as source in parallel and require maintaining states s^i_w for each starting node v separately. This unfortunately yields an overhead with a factor $|\mathsf{V}|$ increase, the same overhead also occurring in [6].

As an optimization also mentioned in [35] for BFS starting at a single node, we can relax our invariant from $s^i_w = 1$ to $s^i_w \geq 1$ for all nodes reachable by a path of length $\leq i$. This makes applying $\text{clip}(\cdot)$ unnecessary, so we omit it to improve performance (since clip is expensive when performed via MPC). However, after the last iteration, to compute the number of reachable nodes, we clip s^D_w *once* and then compute the sum of the clipped s^D_w over all nodes $w \in \mathsf{V}$.

B.2 Full Secure Protocols for π^D_{K} and π^D_{R}

We provide the complete secure message-passing protocols for centrality measures $\pi^D_{\mathsf{K}}, \pi^D_{\mathsf{R}}$ here which follow the template already provided by Prot. 1 in §4.2. The protocol for π^D_{M} can be found in §4.2, Prot. 2.

The complete protocol for π^D_{K} is given in Prot. 14. Recall that this reduces π^D_{K} to π^D_{M} by deduplication, that is provided in §A.6.

Protocol 14 Secure Message-Passing for π^D_{K}

```
$([\![\vec{\mathbf{src}}^d]\!], [\![\vec{\mathbf{dst}}^d]\!]) \leftarrow \mathcal{F}_{\text{DEDUPLICATION}}([\![\vec{\mathbf{src}}]\!], [\![\vec{\mathbf{dst}}]\!])$ // cf. §A.6
Run Prot. 2 on $[\![\vec{\mathbf{src}}^d]\!]$, $[\![\vec{\mathbf{dst}}^d]\!]$, $[\![\vec{\mathbf{isV}}]\!]$.
```

Furthermore, Prot. 15 describes how π^D_{R} can be computed securely. From §4.2.3, recall that we use $|\mathsf{V}|$ independent instances of BFS by working on $|\mathsf{V}|$ vectors $\vec{\mathbf{payload}}^j$ in parallel. For each BFS, we set $\vec{\mathbf{payload}}$ to 1 at the position representing the BFS starting point. $\mathcal{F}_{\text{APPLY}}$ considers both, the prior state $\vec{\mathbf{payload}}^j_{\mathsf{v}}$ as well as $\vec{\mathbf{update}}^j$ resulting from a message-passing round, updating $\vec{\mathbf{payload}}$ to the sum of both. The results after all iterations are clipped using a comparison $\mathcal{F}_{\text{EQZ}}$, which outputs a single bit that we then convert back to arithmetic using $\mathcal{F}_{\text{BIT2A}}$. This process is elaborated on in §A.5. Then, $\vec{\mathbf{payload}}^j$ contains 1 for each reached node, allowing us to count reached nodes by simply computing a sum. Note that while we use $|\mathsf{V}|$ vectors $\vec{\mathbf{payload}}^j$, our permutations $\rho_{\text{src}}, \rho_{\text{dst}}, \rho_{\text{vert}}$ still need to be computed only once s.t. there is no overhead in the one-time initialization.

Protocol 15 Secure Message-Passing for π^D_{R}

```
$[\![\vec{\mathbf{payload}}^j]\!] \leftarrow [\![\vec{\mathbf{0}}]\!]\ \forall 1 \leq j \leq |\mathsf{V}|$
$[\![\rho_{\text{src}}]\!] \leftarrow \mathcal{F}_{\text{GETSORT}}([\![\vec{\mathbf{src}}]\!], [\![\vec{\mathbf{1}} - \vec{\mathbf{isV}}]\!])$ // equal src $\Rightarrow$ vertex first
$[\![\rho_{\text{dst}}]\!] \leftarrow \mathcal{F}_{\text{GETSORT}}([\![\vec{\mathbf{dst}}]\!], [\![\vec{\mathbf{isV}}]\!])$ // equal dst $\Rightarrow$ vertex last
$[\![\rho_{\text{vert}}]\!] \leftarrow \mathcal{F}_{\text{GETSORT}}([\![\vec{\mathbf{1}} - \vec{\mathbf{isV}}]\!], [\![\vec{\mathbf{src}}]\!], [\![\vec{\mathbf{1}} - \vec{\mathbf{isV}}]\!])$ // Vertices first, ordered
for $1 \leq j \leq |\mathsf{V}|$ (in parallel) do
    $[\![\vec{\mathbf{payload}}^j_{\mathsf{v}}]\!] \leftarrow \mathcal{F}_{\text{APPLYPERM}}([\![\rho_{\text{vert}}]\!], [\![\vec{\mathbf{payload}}^j]\!])$
    $[\![\vec{\mathbf{payload}}^j_{\mathsf{v}}]\!] \leftarrow [\![\vec{\mathbf{e}}_{\mathrm{j}}]\!]$ // initial state: node can reach itself, $\vec{\mathbf{e}}_{\mathrm{j}}$ denotes the $j$'th unit vector
    for $1 \leq i \leq D$ do
        $[\![\vec{\mathbf{payload}}'^j_{\mathsf{v}}]\!] \leftarrow \mathcal{F}_{\text{PROPAGATE-1}}([\![\vec{\mathbf{payload}}^j_{\mathsf{v}}]\!])$
        $[\![\vec{\mathbf{payload}}^j_{\text{src}}]\!] \leftarrow \mathcal{F}_{\text{SWPERM}}([\![\rho_{\text{vert}}]\!], [\![\rho_{\text{src}}]\!], [\![\vec{\mathbf{payload}}'^j]\!])$
        $[\![\vec{\mathbf{payload}}'^j_{\text{src}}]\!] \leftarrow \mathcal{F}_{\text{PROPAGATE-2}}([\![\vec{\mathbf{payload}}^j_{\text{src}}]\!])$
        $[\![\vec{\mathbf{payload}}^j_{\text{dst}}]\!] \leftarrow \mathcal{F}_{\text{SWPERM}}([\![\rho_{\text{src}}]\!], [\![\rho_{\text{dst}}]\!], [\![\vec{\mathbf{payload}}'^j]\!])$
        $[\![\vec{\mathbf{payload}}'^j_{\text{dst}}]\!] \leftarrow \mathcal{F}_{\text{GATHER-1}}([\![\vec{\mathbf{payload}}^j_{\text{dst}}]\!])$
        $[\![\vec{\mathbf{payload}}'^j_{\mathsf{v}}]\!] \leftarrow \mathcal{F}_{\text{SWPERM}}([\![\rho_{\text{dst}}]\!], [\![\rho_{\text{vert}}]\!], [\![\vec{\mathbf{payload}}'^j]\!])$
        $[\![\vec{\mathbf{update}}^j_{\mathsf{v}}]\!] \leftarrow \mathcal{F}_{\text{GATHER-2}}([\![\vec{\mathbf{payload}}'^j_{\mathsf{v}}]\!])$
        $[\![\vec{\mathbf{payload}}^j_{\mathsf{v}}]\!] \leftarrow [\![\vec{\mathbf{payload}}'^j_{\mathsf{v}}]\!] + [\![\vec{\mathbf{update}}^j_{\mathsf{v}}]\!]$ // $\mathcal{F}_{\text{APPLY}}$
    end for
    $[\![\text{payload}^j_{\mathsf{V}_i}]\!] \leftarrow 1 - \mathcal{F}_{\text{BIT2A}}(\mathcal{F}_{\text{EQZ}}([\![\text{payload}^j_{\mathsf{V}_i}]\!]))\ \forall 1 \leq i \leq |\mathsf{V}|$ // apply clipping
    $[\![\text{payload}_j]\!] \leftarrow \sum_{i=1}^{|\mathsf{V}|} [\![\text{payload}^j_{\mathsf{V}_i}]\!]$ // final output
end for
```

B.3 Communication Complexity Analysis

For brevity, we use $n := |\mathsf{V}| + |\mathcal{E}|$ as a shorthand notation for the length of the DAG list throughout this section.

Overall, our protocols require to first set up the permutations $[\![\rho_{\text{src}}]\!], [\![\rho_{\text{dst}}]\!], [\![\rho_{\text{vert}}]\!]$ including all optimizations from §A.4. We need one instance of $\mathcal{F}_{\text{GETSORT}}$ over $\lceil \log_2 |\mathsf{V}| \rceil + 1$ bits/iterations and one over $\lceil \log_2 |\mathsf{V}| \rceil + 2$ bits/iterations to retrieve all three permutations. This costs a total of

- $n + \lceil \log_2 |\mathsf{V}| \rceil \cdot 8n + n + (\lceil \log_2 |\mathsf{V}| \rceil + 1) \cdot 8n = 16n \cdot \lceil \log_2(|\mathsf{V}|) \rceil + 10n$ ring elements communication in preprocessing,
- $2n + \lceil \log_2 |\mathsf{V}| \rceil \cdot 6n + 2n + (\lceil \log_2 |\mathsf{V}| \rceil + 1) \cdot 6n = 12n \cdot \lceil \log_2(|\mathsf{V}|) \rceil + 10n$ ring elements communication per party online,
- and $1 + (\lceil \log_2(|\mathsf{V}|) \rceil + 1) \cdot 4 = 4 \cdot \lceil \log_2(|\mathsf{V}|) \rceil + 5$ rounds.

Furthermore, each of the message-passing iteration instantiates $\mathcal{F}_{\text{SWPERM}}$ three times for order changes $\rho_{\text{vert}} \rightarrow \rho_{\text{src}}$, $\rho_{\text{src}} \rightarrow \rho_{\text{dst}}$, and $\rho_{\text{dst}} \rightarrow \rho_{\text{vert}}$. Besides potentially $\mathcal{F}_{\text{APPLY}}$, these switches require the only interaction per iteration as the other operations run non-interactively [35]. In addition, once before the first iteration, we need to instantiate $\mathcal{F}_{\text{APPLYPERM}}$ once to reach vertex order. We finally note that as per [35], when switching from vertex order to source order, the intermediate payload has two entries per row instead of one,

Table 4: Communication and round complexity of all metrics as implemented in [6], translated to our setting, and as implemented by us. Communication is in elements of $\mathbb{Z}_{2^k}$. Fractionals exist due to binary domain computation used to compute $\mathcal{F}_{\text{EQZ}}$ (§A.5). Preprocessing communication is what is sent by the dealer, online communication is what is sent by each of the online parties.

Metric	Cost	prior [6]	**ours**
π_{M}^{D}	preproc. com	$\lvert\mathsf{V}\rvert^2 \cdot (D-1)$	$16(\lvert\mathsf{V}\rvert + \lvert\mathcal{E}\rvert) \cdot \lceil \log_2 \lvert\mathsf{V}\rvert \rceil + 27(\lvert\mathsf{V}\rvert + \lvert\mathcal{E}\rvert) + 8(\lvert\mathsf{V}\rvert + \lvert\mathcal{E}\rvert) \cdot D$
	online com	$2\lvert\mathsf{V}\rvert^2 \cdot (D-1)$	$12(\lvert\mathsf{V}\rvert + \lvert\mathcal{E}\rvert) \cdot \lceil \log_2 \lvert\mathsf{V}\rvert \rceil + 17(\lvert\mathsf{V}\rvert + \lvert\mathcal{E}\rvert) + 4(\lvert\mathsf{V}\rvert + \lvert\mathcal{E}\rvert) \cdot D$
	online rounds	$D-1$	$4 \cdot \lceil \log_2 \lvert\mathsf{V}\rvert \rceil + 7 + 3 \cdot D$
π_{K}^{D}	preproc. com	$\lvert\mathsf{V}\rvert^2 \cdot (D+\ell-2)$	$24(\lvert\mathsf{V}\rvert + \lvert\mathcal{E}\rvert) \cdot \lceil \log_2 \lvert\mathsf{V}\rvert \rceil + (55-1/k)(\lvert\mathsf{V}\rvert + \lvert\mathcal{E}\rvert) - 3 + 1/k + 8(\lvert\mathsf{V}\rvert + \lvert\mathcal{E}\rvert) \cdot D$
	online com	$2\lvert\mathsf{V}\rvert^2 \cdot (D+\ell-2)$	$18(\lvert\mathsf{V}\rvert + \lvert\mathcal{E}\rvert) \cdot \lceil \log_2 \lvert\mathsf{V}\rvert \rceil + (40-2/k)(\lvert\mathsf{V}\rvert + \lvert\mathcal{E}\rvert) - 6 + 2/k + 4(\lvert\mathsf{V}\rvert + \lvert\mathcal{E}\rvert) \cdot D$
	online rounds	$\lceil \log_2(\ell) \rceil + D - 1$	$8 \cdot \lceil \log_2 \lvert\mathsf{V}\rvert \rceil + 15 + \lceil \log_2(k) \rceil + 3 \cdot D$
π_{R}^{D}	preproc. com	$(2-1/k)\lvert\mathsf{V}\rvert^2 + \lvert\mathsf{V}\rvert^3 \cdot (D-1)$	$16(\lvert\mathsf{V}\rvert + \lvert\mathcal{E}\rvert) \cdot \lceil \log_2 \lvert\mathsf{V}\rvert \rceil + (4-1/k)\lvert\mathsf{V}\rvert^2 + 2\lvert\mathsf{V}\rvert\lvert\mathcal{E}\rvert + 25(\lvert\mathsf{V}\rvert + \lvert\mathcal{E}\rvert) + 8(\lvert\mathsf{V}\rvert + \lvert\mathcal{E}\rvert) \cdot \lvert\mathsf{V}\rvert \cdot D$
	online com	$(4-2/k)\lvert\mathsf{V}\rvert^2 + 2\lvert\mathsf{V}\rvert^3 \cdot (D-1)$	$12(\lvert\mathsf{V}\rvert + \lvert\mathcal{E}\rvert) \cdot \lceil \log_2 \lvert\mathsf{V}\rvert \rceil + (5-2/k)\lvert\mathsf{V}\rvert^2 + \lvert\mathsf{V}\rvert\lvert\mathcal{E}\rvert + 16(\lvert\mathsf{V}\rvert + \lvert\mathcal{E}\rvert) + 4(\lvert\mathsf{V}\rvert + \lvert\mathcal{E}\rvert) \cdot \lvert\mathsf{V}\rvert \cdot D$
	online rounds	$\lceil \log_2(k) \rceil + D$	$4 \cdot \lceil \log_2 \lvert\mathsf{V}\rvert \rceil + 8 + \lceil \log_2(k) \rceil + 3 \cdot D$

doubling the payload size for this step only.[23] Hence, for a total of D message-passing iterations, the collective cost (without $\mathcal{F}_{\text{APPLY}}$) is

- $3 \cdot 3n + 3 \cdot 2n + 2n + 4 \cdot 2n \cdot D = 17n + 8n \cdot D$ ring elements communication in preprocessing,
- $3 \cdot 2n + n + 4 \cdot n \cdot D = 7n + 4n \cdot D$ ring elements communication per party online,
- and $2 + 3 \cdot D$ rounds,

noting that three permutations are used in three distinct combinations of two and carefully observing that during the two rounds of instantiating $\mathcal{F}_{\text{APPLYPERM}}$, the one-time operations for later instances of $\mathcal{F}_{\text{SWPERM}}$ are executed in parallel.

B.3.1 Complexity for π_{M}^{D}. The protocol for π_{M}^{D} (Prot. 2) uses a non-interactive $\mathcal{F}_{\text{APPLY}}$. Hence, the total complexity immediately results from adding together the previously discussed costs, yielding the numbers in Tab. 4.

B.3.2 Complexity for π_{K}^{D}. Recall from §A.6 that we only need to incorporate an additional deduplication step, partially merging it with the computation of the permutations $\llbracket \rho_{\text{src}} \rrbracket$, $\llbracket \rho_{\text{dst}} \rrbracket$, $\llbracket \rho_{\text{vert}} \rrbracket$ for optimization. If we add one initial sorting iteration by $\vec{\textbf{isV}}$ to the deduplication, it already computes $\llbracket \rho_{\text{dst}} \rrbracket$ except for the last iteration, saving some cost to set up the permutations. Furthermore, the other sorting procedure needs to be extended by one iteration too to incorporate the newly added bits $\vec{\boldsymbol{\Delta}}$ shifting duplicates to the end of the list. Combining the cost from §A.6 with the savings from correlated permutations yields an overhead compared to π_{M}^{D} of

- $(16n \cdot \lceil \log_2 |\mathsf{V}| \rceil + (5 - 1/k)n - 3 + 1/k) - (n + \lceil \log_2 |\mathsf{V}| \rceil \cdot 8n) + 3 \cdot 8n = 8n \cdot \lceil \log_2 |\mathsf{V}| \rceil + (28 - 1/k)n - 3 + 1/k$ ring elements communication in preprocessing,
- and $(12n \cdot \lceil \log_2 |\mathsf{V}| \rceil + (7 - 2/k)n - 6 + 2/k) - (2n + \lceil \log_2 |\mathsf{V}| \rceil \cdot 6n) + 3 \cdot 6n = 6n \cdot \lceil \log_2 |\mathsf{V}| \rceil + (23 - 2/k)n - 6 + 2/k$ ring elements communication per party online.

The $8 \lceil \log_2 |\mathsf{V}| \rceil + \lceil \log_2 k \rceil + 2$ rounds for deduplication get increased by 4 for the additional iteration w.r.t. $\vec{\textbf{isV}}$. Other operations mostly run in parallel, but the deduplication needs to be finalized in order to run the additional sorting iteration based on the newly added bits[24] which still needs to be followed by another iteration to reach $\llbracket \rho_{\text{vert}} \rrbracket$. Then, the message-passing phase follows, yielding the results given in Tab. 4.

B.3.3 Complexity for π_{R}^{D}. From §B.2, recall that all iterations are executed on $|\mathsf{V}|$ many payloads in parallel. As the parallel instances use the same permutation, this yields an overhead of $2n + 8n \cdot D$ communication during preprocessing and $n + 4n \cdot D$ communication per party online for each node but the first. The final clipping operation on $|\mathsf{V}|$ values yields the remaining overhead (§A.5) leading to the numbers reported in Tab. 4.

B.3.4 Complexity for Prior Work [6]. To compare to the only prior work computing centrality measures on multilayer graphs using MPC [6], for fairness reasons, we translate their protocols to our setting. The cost analysis for our instantiation of [6] then is simple: For $\pi_{\mathsf{M}}^{D}, \pi_{\mathsf{K}}^{D}$, all but the first iteration require $|\mathsf{V}|^2$ multiplications, π_{R}^{D} requires $|\mathsf{V}|^3$ multiplications. In addition, π_{K}^{D} initially computes the OR over all input matrices using $\ell - 1$ multiplications per entry in $\lceil \log_2(\ell) \rceil$ rounds, as documented in [6], where ℓ is the number of graph layers. For π_{R}^{D}, we need to apply clipping (§A.5) to $|\mathsf{V}|^2$ matrix entries.

B.4 Preventing Information Leakage from Input Dimensions

Our protocol's efficient input phase comes with the downside of requiring each client to disclose their number of submitted entries to the DAG list to the servers. Depending on the application scenario, if such disclosure is not appropriate, clients may input a randomized number of additional invalid entries as a padding, or agree upon an upper bound b with $|\mathsf{E}_i| \leq b$ for all layers $1 \leq i \leq \ell$ that they pad their entries to. Padding elements can be marked by adding an additional flag $\in \{0, 1\}$ to each DAG list entry which is 1 for all padding elements. On the server side, the flag values can be used to set entries to invalid elements that are ignored by the message-passing, which is done as for duplicate entries as described in §A.6. Another option is to run a secure compaction[25], pushing all padding

[23] In more detail, we apply two instances of $\mathcal{F}_{\text{SWPERM}}$ for the same permutations on two vectors, each containing one of the two entries per row.

[24] This takes 3 rounds as the iteration is only waiting for the input vector, not the input permutation.

[25] Secure sorting by single bit keys, which matches a single iteration of the radix sort described in §A.4.

entries to the end of the DAG list. Revealing the ordered flag values then allows discarding padding elements, decreasing the size of the DAG list and hence increasing performance at the cost of leaking how many padding elements were used in total only instead of the padding used per party.

B.5 Security Proof

Recall that we assume the number of DAG list entries, the number of nodes, and the input sizes of all clients to be public. We can formally prove the security of our protocol in the simulation paradigm by utilizing its modular architecture. The overview is as follows.

The protocol (Prot. 1) uses functionalities $\mathcal{F}_{\text{GETSORT}}$, $\mathcal{F}_{\text{APPLYPERM}}$, $\mathcal{F}_{\text{PROPAGATE-1}}$, $\mathcal{F}_{\text{PROPAGATE-2}}$, $\mathcal{F}_{\text{GATHER-1}}$, $\mathcal{F}_{\text{GATHER-2}}$, and $\mathcal{F}_{\text{APPLY}}$. Concrete secure instantiations of $\mathcal{F}_{\text{PROPAGATE-1}}$, $\mathcal{F}_{\text{PROPAGATE-2}}$, $\mathcal{F}_{\text{GATHER-1}}$, and $\mathcal{F}_{\text{GATHER-2}}$ are provided by Graphiti [35]. Hence, also the security of these instantiations follows from [35]. Furthermore, $\mathcal{F}_{\text{GETSORT}}$ is instantiated as described in §A.4, based on the sorting protocol of [7], but replacing their building block for shuffling with the protocol from [35] that matches our setting. Inserting the security argument for the shuffling given in [35] into the security proof of [7] immediately yields the security for our sorting instantiation. Our novel instantiation for $\mathcal{F}_{\text{SWPERM}}$ builds upon $\mathcal{F}_{\text{APPLYPERM}}$ which is proven secure in [35]. As discussed in §A.3, our instantiation inherits the security of $\mathcal{F}_{\text{APPLYPERM}}$. Finally, $\mathcal{F}_{\text{APPLY}}$ depends on the exact application. For π^D_{M} and π^D_{K}, its instantiation requires no interaction and hence is secure. For π^D_{R}, it can be securely instantiated as demonstrated in §A.5.

Recall from §4 that our protocol provides flexibility regarding how exactly inputs are provided and outputs are to be disclosed. Hence, we focus on the abstract functionality $\mathcal{F}_\pi$ given in Fig. 19 which, for a centrality measure $\pi \in \{\pi^D_{\mathsf{M}}, \pi^D_{\mathsf{K}}, \pi^D_{\mathsf{R}}\}$ (and accordingly chosen $\mathcal{F}_{\text{APPLY}}$), receives the graph already in secret-shared format and computes shares of the output without disclosing them to any server/client. We can now proceed by proving that our protocol securely instantiates $\mathcal{F}_\pi$ in a hybrid model where the servers have access to a trusted third party for computing any of the hybrid functionalities.

Functionality $\mathcal{F}_\pi$

Input: DAG list of a multigraph, represented by secret-shared vectors $[[\vec{\mathbf{src}}]]$, $[[\vec{\mathbf{dst}}]]$, $[[\vec{\mathbf{isV}}]]$, $[[\vec{\mathbf{payload}}]]$ of dimension $|\mathsf{V}| + |\mathcal{E}|$ for public parameters $|\mathsf{V}|, |\mathcal{E}|$.

Output: $[[\vec{\mathbf{payload}}_{\mathsf{V}}]]$ with the i'th entry being $[[\pi(u)]]$ for the i'th vertex $u \in \mathsf{V}$.

Figure 20: Functionality for computing a $\pi \in \{\pi^D_{\mathsf{M}}, \pi^D_{\mathsf{K}}, \pi^D_{\mathsf{R}}\}$

Theorem 2. *Prot. 1 securely computes $\mathcal{F}_\pi$ for $\pi \in \{\pi^D_{\mathsf{M}}, \pi^D_{\mathsf{K}}, \pi^D_{\mathsf{R}}\}$ in the ($\mathcal{F}_{\text{GETSORT}}$, $\mathcal{F}_{\text{APPLYPERM}}$, $\mathcal{F}_{\text{PROPAGATE-1}}$, $\mathcal{F}_{\text{PROPAGATE-2}}$, $\mathcal{F}_{\text{GATHER-1}}$, $\mathcal{F}_{\text{GATHER-2}}$, $\mathcal{F}_{\text{APPLY}}$)-hybrid model in the presence of a semi-honest adversary corrupting one server (among the two online servers and the helper server $\mathcal{H}$).*

Proof. We design a simulator that, provided the protocol input and output shares of the corrupted server, generates a transcript of this server that is indistinguishable from the corrupted server's view during a real protocol execution. The simulator works as follows, keeping track of all shares held by the corrupted party:

Each computation step of protocol 1 is a call to some ideal functionality or a linear and hence non-interactive operation. Non-interactive operations can easily be simulated by being skipped by the simulator as no server receives any message. For a call to an ideal functionality, the simulator emulates the ideal functionality, utilizing our hybrid model.

This simulation is indistinguishable from a real protocol execution, directly following from the fact that the simulator only emulates ideal functionalities provided by the hybrid model. □

Finally, we conclude the standalone security of our protocol:

Theorem 3. *Prot. 1 securely computes $\mathcal{F}_\pi$ for $\pi \in \{\pi^D_{\mathsf{M}}, \pi^D_{\mathsf{K}}, \pi^D_{\mathsf{R}}\}$ in the presence of a semi-honest adversary corrupting one server (among the two online servers and the helper server $\mathcal{H}$).*

Proof. This directly follows from the security in the prior hybrid model, together with the secure instantiations of all functionalities, using the sequential modular composition theorem from [17]. Furthermore, parallelizing steps of the protocol is not violating security as the simulators for the protocol and all building blocks are black-box simulators and do not use rewinding while also the clients' inputs are fixed before the start of the execution [39]. □

In an exact deployment scenario, it is up to the deploying entities to exactly define the input and output phases in between which functionality $\mathcal{F}_\pi$ should be computed. Clients could, e.g., secret share entries of the DAG list without disclosing their content, and final scores $\pi(u)$ can be disclosed to any specific entity (server/client/other) without disclosing anything else. The security of this directly follows from the security of our underlying secret sharing scheme, see, e.g., [22, 28]. Also, recall from §2.4 that the sharing $[[x]]$ of a value $x \in \mathbb{Z}_{2^k}$ consists of $x_0 \in \mathbb{Z}_{2^k}$ held by server P_0 and $x_1 \in \mathbb{Z}_{2^k}$ held by server P_1 with x_0, x_1 random subject to $x_0 + x_1 = x$. This secret sharing scheme is unconditionally secure as each individual share, without knowledge of the other share, is uniformly random. Utilizing pre-shared PRF keys to generate random shares instead of sending truly random shares, saving on communication as in [51, 55], reduces security from unconditional/information-theoretic to computational security, using the security of the utilized PRF as cryptographic assumption.

C Additional Evaluation Results

Besides considering a LAN setting assuming excellent connections between our servers, we also benchmark the protocols for π_{M}^{D} for a substantially slower WAN with only 100 MBit/s links and 100 ms RTT instead of 1 GBit/s and 1 ms RTT. The results are depicted in Fig. 21. For larger instances, the run times for WAN are 6× to 9× higher than LAN, regarding both one-time cost and cost per iteration, for our as well as for the prior protocols from [6]. Hence, with the significantly slower network, the protocols are slowed down by one order of magnitude, revealing a corridor for moderate network settings where, even in the worst case, one iteration for 50k nodes and 5 million edges still takes less than a minute.

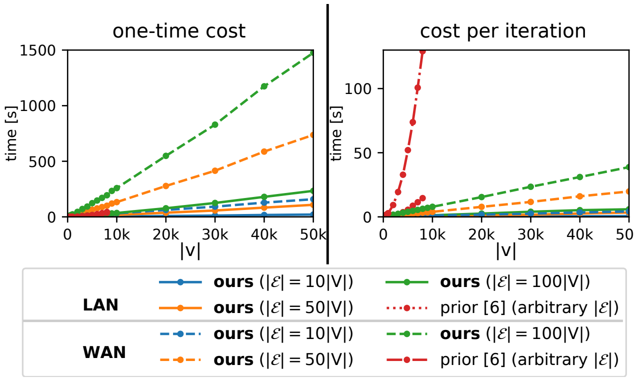


Figure 21: Total run times and communication (preprocessing + online) of our protocol for π_{M}^{D} vs. the prior protocol from [6], using different network settings. LAN corresponds to 1ms RTT, 1GBit/s and WAN corresponds to 100ms RTT, 100MBit/s links between the parties. ($\ell = 3$ layers)

Furthermore, for all benchmark results featured within this paper, we provide full benchmark results regarding run time and observed maximum allocation of virtual memory in Tab. 5, Tab. 6, Tab. 7, Tab. 8, Tab. 9, and Tab. 10. Recall that allocated memory is of interest here as this was the limiting factor for our benchmarks, especially for the protocols from [6]. We do not provide the communication here as it can be directly derived from Tab. 4.

Table 5: Run times and maximal observed memory allocation for our protocols for $\pi_{\mathrm{M}}^{D}, \pi_{\mathrm{K}}^{D}$, in a LAN setting, on the larger datasets from §6, Tab. 2. Run times are split into preprocessing + online.

		π_{M}^{D}		π_{K}^{D}	
		ours		**ours**	
	D	run time	memory	run time	memory
arabi	0	0.9+0.5 s	1.2 GB	1.3+1.0 s	1.8 GB
	1	0.8+0.6 s	1.5 GB	1.5+1.0 s	1.8 GB
	2	0.9+0.6 s	1.5 GB	1.5+1.0 s	1.7 GB
	3	1.0+0.6 s	1.6 GB	1.5+1.0 s	1.7 GB
	4	1.0+0.6 s	1.6 GB	1.5+1.0 s	1.9 GB
	5	1.0+0.7 s	1.6 GB	1.5+1.0 s	1.9 GB
	6	1.0+0.7 s	1.5 GB	1.5+1.1 s	1.9 GB
	7	1.0+0.7 s	1.6 GB	1.5+1.1 s	1.9 GB
	8	0.9+0.7 s	1.6 GB	1.6+1.1 s	1.8 GB
	9	0.9+0.7 s	1.4 GB	1.5+1.1 s	1.9 GB
	10	1.0+0.7 s	1.7 GB	1.6+1.1 s	2.0 GB
higgs	0	36.6+29.2 s	28.9 GB	59.7+45.8 s	40.4 GB
	1	38.0+29.8 s	29.7 GB	60.9+46.7 s	41.2 GB
	2	38.8+30.4 s	30.1 GB	61.3+47.2 s	41.8 GB
	3	39.6+30.8 s	30.8 GB	62.0+48.0 s	42.5 GB
	4	39.9+31.1 s	31.4 GB	62.8+48.0 s	43.2 GB
	5	41.1+31.9 s	32.0 GB	63.9+49.0 s	43.8 GB
	6	41.2+32.2 s	32.7 GB	64.7+49.3 s	44.5 GB
	7	42.1+33.0 s	33.4 GB	65.4+50.3 s	45.1 GB
	8	42.5+33.8 s	34.0 GB	66.3+50.4 s	45.8 GB
	9	43.5+34.5 s	34.4 GB	66.8+51.3 s	46.4 GB
	10	44.3+34.7 s	35.4 GB	67.2+52.0 s	47.1 GB

Table 6: Run times and maximal observed memory allocation for our and prior [6] protocols for $\pi_{\mathrm{M}}^{D}, \pi_{\mathrm{K}}^{D}, \pi_{\mathrm{R}}^{D}$, in a LAN setting, on the smaller datasets from §6, Tab. 2. Run times are split into preprocessing + online.

		π_{M}^{D}				π_{K}^{D}				π_{R}^{D}			
		ours		prior [6]		**ours**		prior [6]		**ours**		prior [6]	
	D	run time	memory	run time	memory	run time	memory	run time	memory	run time	memory	run time	memory
aarhus	0	20.5+28.0 ms	0.9 GB			52.8+63.7 ms	0.8 GB			61.5+52.0 ms	0.9 GB		
	1	31.3+33.3 ms	0.8 GB	6.8+2.9 ms	0.8 GB	49.1+65.7 ms	0.8 GB	14.2+1,0.0 ms	0.8 GB	98.4+78.6 ms	0.8 GB	21.0+11.6 ms	0.8 GB
	2	28.3+34.5 ms	0.9 GB	7.4+2.7 ms	0.9 GB	47.2+71.0 ms	0.9 GB	13.3+9.1 ms	0.8 GB	142.9+98.9 ms	1.0 GB	82.4+60.2 ms	1.0 GB
	3	30.4+36.0 ms	0.9 GB	10.2+5.3 ms	0.9 GB	43.8+71.1 ms	0.8 GB	13.8+9.2 ms	0.9 GB	159.6+131.3 ms	1.1 GB	193.9+98.4 ms	1.1 GB
	4	35.8+40.2 ms	0.8 GB	9.0+4.5 ms	0.8 GB	56.8+75.7 ms	0.9 GB	15.1+9.8 ms	0.9 GB	212.6+148.5 ms	1.1 GB	178.9+130.6 ms	1.0 GB
	5	36.4+43.2 ms	0.7 GB	9.3+6.6 ms	0.9 GB	60.8+76.6 ms	0.9 GB	16.6+11.5 ms	0.8 GB	263.9+183.6 ms	1.0 GB	220.5+163.2 ms	1.3 GB
	6	37.6+46.6 ms	0.8 GB	9.7+8.4 ms	0.9 GB	44.5+78.9 ms	0.9 GB	27.3+12.6 ms	0.9 GB	244.4+197.8 ms	1.2 GB	299.1+199.3 ms	1.3 GB
	7	36.6+49.9 ms	0.9 GB	11.5+9.4 ms	0.9 GB	64.3+85.0 ms	0.9 GB	26.1+10.7 ms	0.7 GB	289.3+214.8 ms	1.1 GB	320.7+228.8 ms	1.4 GB
	8	38.4+53.8 ms	0.7 GB	22.3+12.3 ms	0.9 GB	61.5+86.0 ms	0.9 GB	22.2+15.8 ms	0.8 GB	313.3+237.1 ms	1.1 GB	308.2+292.2 ms	1.2 GB
	9	42.1+56.9 ms	0.8 GB	17.1+10.9 ms	0.6 GB	49.9+89.6 ms	0.9 GB	30.3+17.2 ms	0.9 GB	411.8+260.3 ms	1.3 GB	367.3+293.1 ms	1.2 GB
	10	42.7+56.3 ms	0.6 GB	18.4+11.6 ms	0.8 GB	59.9+93.5 ms	0.9 GB	26.6+16.5 ms	0.9 GB	437.4+276.0 ms	1.2 GB	352.3+341.0 ms	1.6 GB
london	0	42.0+40.5 ms	0.9 GB			54.6+86.3 ms	0.9 GB			0.3+0.3 s	1.3 GB		
	1	36.8+46.7 ms	0.9 GB	59.8+34.6 ms	1.1 GB	61.7+88.7 ms	0.9 GB	167.5+92.0 ms	1.1 GB	0.6+0.4 s	1.5 GB	0.4+0.2 s	1.3 GB
	2	46.3+47.8 ms	0.9 GB	89.0+65.1 ms	1.0 GB	70.0+92.5 ms	0.8 GB	270.2+133.8 ms	1.2 GB	0.8+0.6 s	1.7 GB	8.6+8.1 s	20.7 GB
	3	37.7+50.5 ms	0.7 GB	169.6+80.0 ms	1.0 GB	60.0+98.7 ms	0.9 GB	168.4+144.7 ms	1.2 GB	1.1+0.7 s	2.0 GB	16.7+16.1 s	38.9 GB
	4	42.7+51.1 ms	0.8 GB	179.8+120.7 ms	1.2 GB	63.2+97.7 ms	0.9 GB	183.9+160.2 ms	1.3 GB	1.3+0.9 s	2.1 GB	24.8+23.9 s	57.0 GB
	5	44.7+55.3 ms	0.8 GB	142.3+142.2 ms	1.3 GB	78.0+101.1 ms	0.9 GB	257.4+176.5 ms	1.1 GB	1.5+1.0 s	2.2 GB	32.7+31.9 s	75.6 GB
	6	46.3+58.6 ms	0.8 GB	202.3+164.9 ms	1.2 GB	80.6+105.5 ms	0.6 GB	288.4+200.3 ms	1.4 GB	1.8+1.2 s	2.6 GB	40.4+40.1 s	93.8 GB
	7	43.7+61.1 ms	0.9 GB	202.7+193.2 ms	1.2 GB	84.4+108.1 ms	0.8 GB	264.2+241.1 ms	1.4 GB	2.0+1.3 s	2.8 GB	–	–
	8	46.9+65.1 ms	0.9 GB	278.9+207.1 ms	1.5 GB	58.4+112.4 ms	0.9 GB	274.4+238.7 ms	1.5 GB	2.1+1.5 s	3.1 GB	–	–
	9	54.7+67.5 ms	0.9 GB	238.9+214.6 ms	1.4 GB	81.7+110.0 ms	0.8 GB	306.6+263.2 ms	1.5 GB	2.4+1.6 s	3.3 GB	–	–
	10	45.3+70.7 ms	0.6 GB	244.1+228.8 ms	1.5 GB	62.1+115.7 ms	0.7 GB	329.5+282.5 ms	1.6 GB	2.6+1.8 s	3.5 GB	–	–
hiv	0	0.1+0.1 s	0.9 GB			0.1+0.1 s	1.0 GB			2.1+1.8 s	3.6 GB		
	1	0.1+0.1 s	1.0 GB	0.4+0.2 s	2.2 GB	0.1+0.1 s	1.0 GB	0.8+0.6 s	2.8 GB	3.6+2.8 s	5.3 GB	2.1+1.3 s	3.3 GB
	2	0.1+0.1 s	1.0 GB	0.5+0.4 s	2.6 GB	0.1+0.1 s	0.9 GB	1.0+0.8 s	3.0 GB	5.1+3.8 s	6.9 GB	–	–
	3	0.1+0.1 s	1.0 GB	0.7+0.6 s	3.0 GB	0.1+0.1 s	0.8 GB	1.2+0.9 s	3.5 GB	6.5+4.8 s	8.5 GB	–	–
	4	0.1+0.1 s	0.8 GB	0.8+0.7 s	3.2 GB	0.1+0.1 s	0.8 GB	1.3+1.1 s	3.9 GB	8.0+5.8 s	10.2 GB	–	–
	5	0.1+0.1 s	0.9 GB	1.0+0.9 s	3.7 GB	0.1+0.1 s	1.0 GB	1.4+1.2 s	4.2 GB	9.5+6.8 s	11.8 GB	–	–
	6	0.1+0.1 s	0.9 GB	1.1+1.0 s	4.0 GB	0.1+0.1 s	1.0 GB	1.6+1.4 s	4.6 GB	10.9+7.8 s	13.5 GB	–	–
	7	0.1+0.1 s	0.9 GB	1.3+1.2 s	4.4 GB	0.1+0.1 s	1.0 GB	1.8+1.6 s	5.0 GB	12.4+8.8 s	15.3 GB	–	–
	8	0.1+0.1 s	0.9 GB	1.4+1.3 s	4.7 GB	0.1+0.2 s	1.0 GB	2.0+1.7 s	5.3 GB	13.8+9.8 s	17.1 GB	–	–
	9	0.1+0.1 s	0.9 GB	1.6+1.5 s	5.1 GB	0.2+0.2 s	0.9 GB	2.1+1.9 s	5.6 GB	15.5+10.7 s	18.8 GB	–	–
	10	0.1+0.1 s	1.0 GB	1.8+1.6 s	5.5 GB	0.1+0.2 s	1.0 GB	2.3+2.0 s	5.9 GB	16.9+11.8 s	20.6 GB	–	–

Table 7: Run times and maximal observed memory allocation for our and prior [6] protocols for π_{M}^{D} with different sparsity, in a LAN setting. Run times are split into preprocessing + online.

	ours ($\lvert\mathcal{E}\rvert = 10\lvert\mathrm{V}\rvert$)				**ours** ($\lvert\mathcal{E}\rvert = 50\lvert\mathrm{V}\rvert$)				**ours** ($\lvert\mathcal{E}\rvert = 100\lvert\mathrm{V}\rvert$)				prior [6] (arbitrary $\lvert\mathcal{E}\rvert$)			
	one-time cost		cost per iteration		one-time cost		cost per iteration		one-time cost		cost per iteration		one-time cost		cost per iteration	
$\lvert\mathrm{V}\rvert$	run time	memory	run time	memory	run time	memory	run time	memory	run time	memory	run time	memory	run time	memory	run time	memory
100	29.0+33.3 ms	0.8 GB	3.2+2.3 ms	0.0 GB	92.5+58.7 ms	0.9 GB	18.3+2.8 ms	0.0 GB	160.7+94.2 ms	1.0 GB	2.5+3.8 ms	0.0 GB	11.2+2.8 ms	0.9 GB	0.0+3.0 ms	0.0 GB
200	65.2+44.9 ms	0.8 GB	0.0+1.6 ms	0.1 GB	166.8+101.9 ms	0.9 GB	0.0+3.6 ms	0.0 GB	270.7+170.9 ms	1.1 GB	35.5+8.0 ms	0.0 GB	16.9+7.8 ms	0.9 GB	25.1+9.3 ms	0.0 GB
300	86.7+53.0 ms	0.8 GB	0.0+4.2 ms	0.0 GB	219.1+148.1 ms	1.0 GB	0.0+9.2 ms	0.0 GB	383.9+280.9 ms	1.1 GB	47.6+9.4 ms	0.0 GB	61.7+22.6 ms	0.9 GB	61.9+19.4 ms	0.0 GB
400	130.1+68.0 ms	0.9 GB	0.0+0.5 ms	0.0 GB	343.4+195.7 ms	1.1 GB	0.0+5.0 ms	0.0 GB	524.4+368.0 ms	1.3 GB	23.4+10.8 ms	0.0 GB	106.4+42.9 ms	1.1 GB	23.5+39.0 ms	0.1 GB
500	124.0+73.2 ms	0.9 GB	0.0+0.7 ms	0.0 GB	418.7+244.5 ms	1.1 GB	0.0+6.5 ms	0.0 GB	780.9+498.0 ms	1.2 GB	0.0+18.6 ms	0.0 GB	121.4+55.3 ms	1.2 GB	107.4+52.2 ms	0.1 GB
600	148.1+90.2 ms	0.9 GB	0.0+4.6 ms	0.1 GB	441.2+315.2 ms	1.2 GB	9.7+9.2 ms	0.0 GB	832.4+676.1 ms	1.6 GB	38.2+19.9 ms	0.0 GB	136.9+80.5 ms	1.4 GB	167.6+63.7 ms	0.1 GB
700	164.2+105.6 ms	0.9 GB	0.0+3.3 ms	0.0 GB	571.1+362.4 ms	1.3 GB	7.9+15.9 ms	0.0 GB	1.1+0.8 s	1.6 GB	23.0+35.0 ms	0.0 GB	201.9+109.2 ms	1.5 GB	164.8+81.4 ms	0.2 GB
800	147.7+115.2 ms	1.0 GB	0.4+2.0 ms	0.0 GB	601.1+412.5 ms	1.4 GB	7.6+16.8 ms	0.0 GB	1.2+0.9 s	1.8 GB	35.4+35.0 ms	0.0 GB	248.6+144.7 ms	1.6 GB	86.0+106.2 ms	0.0 GB
900	207.7+125.6 ms	1.0 GB	0.0+3.3 ms	0.0 GB	775.2+468.8 ms	1.3 GB	0.0+9.4 ms	0.1 GB	1.5+1.0 s	1.8 GB	31.2+33.6 ms	0.0 GB	312.9+184.2 ms	2.0 GB	98.2+133.5 ms	0.2 GB
1000	228.4+139.4 ms	0.8 GB	2.5+3.4 ms	0.1 GB	863.7+570.3 ms	1.4 GB	11.9+27.6 ms	0.0 GB	1.5+1.1 s	1.8 GB	94.5+46.2 ms	0.1 GB	378.3+225.2 ms	2.2 GB	113.7+157.9 ms	0.4 GB
2000	341.1+245.0 ms	1.1 GB	48.1+10.2 ms	0.0 GB	1.6+1.3 s	1.8 GB	63.6+29.6 ms	0.2 GB	3.2+2.5 s	3.3 GB	147.6+89.3 ms	0.1 GB	1.5+0.9 s	6.1 GB	319.3+647.1 ms	1.5 GB
3000	571.9+392.0 ms	1.3 GB	25.3+18.1 ms	0.0 GB	2.7+2.1 s	2.6 GB	85.5+33.0 ms	0.2 GB	5.0+4.1 s	4.5 GB	192.3+135.6 ms	0.3 GB	3.3+2.0 s	12.8 GB	0.6+1.5 s	3.5 GB
4000	779.8+532.3 ms	1.4 GB	12.3+16.5 ms	0.1 GB	3.4+2.7 s	3.4 GB	125.3+93.1 ms	0.1 GB	6.6+5.4 s	5.9 GB	262.2+165.5 ms	0.1 GB	5.8+3.5 s	21.7 GB	1.0+2.6 s	5.9 GB
5000	1.1+0.8 s	1.6 GB	0.0+16.3 ms	0.1 GB	4.4+3.7 s	4.2 GB	144.2+105.0 ms	0.2 GB	8.8+7.2 s	7.5 GB	339.4+202.5 ms	0.2 GB	9.1+5.5 s	34.7 GB	1.5+4.1 s	9.6 GB
6000	1.4+0.9 s	1.7 GB	51.7+11.2 ms	0.1 GB	5.3+4.4 s	5.1 GB	222.4+127.7 ms	0.2 GB	10.7+8.7 s	9.3 GB	400.3+238.3 ms	0.3 GB	12.9+7.9 s	48.8 GB	2.5+6.1 s	14.6 GB
7000	1.4+1.1 s	1.9 GB	49.7+38.4 ms	0.1 GB	6.2+5.2 s	5.7 GB	253.7+137.1 ms	0.2 GB	12.4+10.3 s	10.4 GB	473.6+249.8 ms	0.3 GB	17.5+10.8 s	67.6 GB	3.1+8.3 s	19.1 GB
8000	1.6+1.2 s	1.8 GB	50.6+38.6 ms	0.1 GB	7.1+5.8 s	6.4 GB	276.4+177.3 ms	0.2 GB	14.1+11.7 s	11.7 GB	606.4+270.1 ms	0.4 GB	23.0+14.0 s	84.5 GB	3.7+10.9 s	24.3 GB
9000	1.9+1.4 s	2.3 GB	50.2+46.0 ms	0.0 GB	8.6+7.0 s	7.5 GB	315.0+204.1 ms	0.2 GB	17.2+14.0 s	14.2 GB	629.5+301.2 ms	0.4 GB				
10000	2.2+1.6 s	2.2 GB	28.4+49.0 ms	0.2 GB	9.6+7.9 s	8.2 GB	312.3+196.9 ms	0.2 GB	19.4+15.7 s	15.4 GB	942.1+320.0 ms	0.5 GB				
20000	4.4+3.5 s	4.2 GB	162.2+92.9 ms	0.1 GB	20.7+16.8 s	16.5 GB	760.6+335.9 ms	0.4 GB	43.6+33.7 s	32.0 GB	1.8+0.8 s	0.9 GB				
30000	6.4+5.3 s	5.9 GB	278.1+158.2 ms	0.2 GB	31.8+25.5 s	24.5 GB	1.1+0.6 s	0.7 GB	71.9+53.0 s	48.5 GB	2.7+1.3 s	1.5 GB				
40000	9.3+7.7 s	8.0 GB	271.9+195.4 ms	0.2 GB	46.6+36.6 s	35.7 GB	1.5+1.1 s	1.0 GB	1.7+1.3 min	70.4 GB	3.4+1.8 s	2.1 GB				
50000	11.5+9.5 s	10.1 GB	405.4+213.6 ms	0.3 GB	62.2+46.3 s	44.4 GB	2.1+1.2 s	1.2 GB	2.3+1.6 min	86.9 GB	4.9+1.0 s	2.1 GB				
60000	13.8+11.3 s	11.8 GB	442.4+266.1 ms	0.3 GB												
70000	17.3+14.4 s	14.3 GB	516.3+227.0 ms	0.4 GB												
80000	19.6+16.4 s	16.2 GB	639.3+300.6 ms	0.3 GB												
90000	22.5+18.3 s	18.4 GB	759.8+292.2 ms	0.5 GB												
100000	25.0+20.1 s	20.8 GB	814.7+635.8 ms	0.5 GB												
200000	58.1+44.5 s	43.2 GB	1.9+1.0 s	1.1 GB												
300000	98.4+72.6 s	65.4 GB	2.9+1.8 s	1.5 GB												
400000	2.3+1.6 min	89.4 GB	4.4+3.3 s	1.9 GB												
500000	2.9+2.1 min	111.1 GB	5.0+3.0 s	2.4 GB												

Table 8: Run times and maximal observed memory allocation for our and prior [6] protocols for π_{M}^{D} with different sparsity, in a WAN setting. Run times are split into preprocessing + online.

	ours ($\lvert\mathcal{E}\rvert = 10\lvert V\rvert$)				**ours** ($\lvert\mathcal{E}\rvert = 50\lvert V\rvert$)				**ours** ($\lvert\mathcal{E}\rvert = 100\lvert V\rvert$)				prior [6] (arbitrary $\lvert\mathcal{E}\rvert$)			
	one-time cost		cost per iteration		one-time cost		cost per iteration		one-time cost		cost per iteration		one-time cost		cost per iteration	
$\lvert V\rvert$	run time	memory	run time	memory	run time	memory	run time	memory	run time	memory	run time	memory	run time	memory	run time	memory
100	0.9+2.1 s	0.7 GB	62.9+153.8 ms	0.1 GB	0.9+2.4 s	0.9 GB	45.0+177.8 ms	0.0 GB	1.3+2.7 s	0.9 GB	0.0+144.0 ms	0.0 GB	259.9+102.0 ms	0.9 GB	109.0+202.1 ms	0.0 GB
200	1.3+2.4 s	0.9 GB	45.9+154.9 ms	0.0 GB	1.3+2.9 s	0.9 GB	217.8+161.1 ms	0.0 GB	2.2+3.4 s	1.1 GB	80.4+160.7 ms	0.0 GB	281.4+115.1 ms	0.9 GB	311.2+336.0 ms	0.1 GB
300	0.9+2.7 s	0.9 GB	35.8+150.6 ms	0.0 GB	2.0+3.4 s	1.0 GB	64.2+186.4 ms	0.0 GB	2.9+4.7 s	1.2 GB	153.2+184.4 ms	0.0 GB	282.3+127.2 ms	0.9 GB	439.0+433.2 ms	0.1 GB
400	1.0+2.9 s	1.0 GB	54.7+161.4 ms	0.0 GB	2.3+3.7 s	1.1 GB	87.9+170.0 ms	0.0 GB	3.6+5.4 s	1.3 GB	170.8+200.5 ms	0.0 GB	317.1+140.4 ms	1.0 GB	515.4+637.2 ms	0.2 GB
500	0.8+2.9 s	0.9 GB	36.8+170.0 ms	0.0 GB	2.6+4.5 s	1.2 GB	130.1+171.1 ms	0.0 GB	4.3+7.9 s	1.3 GB	196.4+232.4 ms	0.0 GB	339.6+143.1 ms	1.2 GB	636.6+852.2 ms	0.2 GB
600	1.1+3.1 s	0.9 GB	103.0+190.9 ms	0.0 GB	3.1+5.1 s	1.2 GB	160.4+191.6 ms	0.0 GB	5.2+9.1 s	1.5 GB	232.1+283.9 ms	0.1 GB	398.6+202.6 ms	1.3 GB	0.5+1.1 s	0.1 GB
700	1.3+3.3 s	0.9 GB	0.0+129.4 ms	0.1 GB	3.6+5.5 s	1.2 GB	115.5+190.8 ms	0.0 GB	6.0+9.8 s	1.7 GB	299.1+295.5 ms	0.0 GB	416.1+216.8 ms	1.5 GB	0.6+1.2 s	0.2 GB
800	1.3+3.3 s	1.0 GB	207.0+188.8 ms	0.0 GB	3.9+5.8 s	1.4 GB	178.6+198.8 ms	0.0 GB	6.7+11.0 s	1.8 GB	280.8+315.8 ms	0.0 GB	501.8+230.6 ms	1.7 GB	0.7+1.5 s	0.3 GB
900	1.6+3.4 s	1.0 GB	45.6+158.2 ms	0.0 GB	4.2+6.2 s	1.4 GB	207.9+213.4 ms	0.0 GB	7.4+11.6 s	1.8 GB	349.3+341.2 ms	0.1 GB	553.3+273.5 ms	2.0 GB	0.7+1.9 s	0.3 GB
1000	1.7+3.4 s	1.0 GB	134.3+164.4 ms	0.0 GB	4.6+8.6 s	1.4 GB	226.4+287.3 ms	0.0 GB	8.1+14.6 s	2.0 GB	472.0+521.8 ms	0.0 GB	687.2+314.3 ms	2.2 GB	0.8+2.1 s	0.4 GB
2000	2.6+4.4 s	1.1 GB	175.2+174.0 ms	0.0 GB	8.8+15.9 s	2.1 GB	437.7+492.6 ms	0.0 GB	16.2+28.7 s	3.3 GB	832.9+882.5 ms	0.1 GB	2.0+0.9 s	6.1 GB	1.9+7.4 s	1.4 GB
3000	3.7+6.1 s	1.3 GB	148.1+182.6 ms	0.0 GB	13.5+24.0 s	2.8 GB	625.4+657.4 ms	0.0 GB	25.9+45.1 s	4.7 GB	1.2+1.2 s	0.1 GB	4.3+2.0 s	12.9 GB	3.4+15.9 s	3.5 GB
4000	4.6+7.0 s	1.5 GB	207.2+204.5 ms	0.0 GB	17.9+31.1 s	3.4 GB	711.2+916.7 ms	0.1 GB	33.9+58.9 s	6.0 GB	1.6+1.6 s	0.2 GB	7.5+3.8 s	21.6 GB	5.5+27.3 s	5.8 GB
5000	5.9+11.0 s	1.7 GB	200.9+279.3 ms	0.0 GB	23.3+40.8 s	4.3 GB	1.1+1.0 s	0.1 GB	45.3+78.0 s	7.7 GB	2.0+1.9 s	0.2 GB	11.5+5.1 s	34.8 GB	8.6+43.4 s	9.1 GB
6000	6.9+11.8 s	1.8 GB	228.7+290.8 ms	0.0 GB	27.8+48.5 s	5.1 GB	1.3+1.2 s	0.2 GB	54.1+93.0 s	9.3 GB	2.5+2.2 s	0.2 GB	16.6+8.3 s	48.8 GB	12.0+61.8 s	14.0 GB
7000	7.9+13.4 s	1.9 GB	338.6+317.6 ms	0.1 GB	32.4+56.0 s	5.7 GB	1.4+1.3 s	0.1 GB	1.0+1.8 min	10.4 GB	2.9+2.6 s	0.3 GB	22.3+11.1 s	67.6 GB	16.5+84.1 s	18.3 GB
8000	8.9+14.4 s	2.1 GB	348.1+340.2 ms	0.0 GB	36.7+63.4 s	6.3 GB	1.6+1.6 s	0.2 GB	1.2+2.0 min	11.5 GB	3.2+3.1 s	0.5 GB	29.0+14.8 s	84.5 GB	0.4+1.8 min	23.4 GB
9000	10.3+16.3 s	2.3 GB	353.1+348.5 ms	0.0 GB	44.2+75.7 s	7.5 GB	1.8+1.7 s	0.2 GB	1.4+2.5 min	14.1 GB	3.8+3.2 s	0.4 GB				
10000	11.4+20.1 s	2.5 GB	493.6+532.5 ms	0.0 GB	48.9+83.8 s	8.1 GB	2.0+1.9 s	0.2 GB	1.6+2.7 min	15.3 GB	4.1+3.7 s	0.4 GB				
20000	22.9+39.4 s	4.2 GB	873.7+854.0 ms	0.1 GB	1.7+2.9 min	16.3 GB	4.1+3.7 s	0.4 GB	3.4+5.7 min	31.4 GB	8.4+7.0 s	0.9 GB				
30000	34.0+58.0 s	6.0 GB	1.3+1.2 s	0.1 GB	2.6+4.3 min	24.0 GB	6.3+5.4 s	0.7 GB	5.2+8.6 min	47.3 GB	12.7+10.8 s	1.4 GB				
40000	47.7+80.5 s	8.0 GB	1.8+1.6 s	0.2 GB	3.7+6.1 min	34.8 GB	8.7+7.3 s	0.9 GB	7.5+12.1 min	68.9 GB	16.7+14.4 s	1.8 GB				
50000	59.3+99.4 s	10.1 GB	2.2+2.0 s	0.3 GB	4.6+7.6 min	43.0 GB	10.5+9.3 s	1.2 GB	9.4+15.2 min	84.9 GB	21.6+17.1 s	2.3 GB				

Table 9: Run times and maximal observed memory allocation for our and prior [6] protocols for π_{K}^{D} with different sparsity, in a LAN setting. Run times are split into preprocessing + online.

	ours ($\lvert\mathcal{E}\rvert = 10\lvert V\rvert$)				**ours** ($\lvert\mathcal{E}\rvert = 50\lvert V\rvert$)				**ours** ($\lvert\mathcal{E}\rvert = 100\lvert V\rvert$)				prior [6] (arbitrary $\lvert\mathcal{E}\rvert$)			
	one-time cost		cost per iteration		one-time cost		cost per iteration		one-time cost		cost per iteration		one-time cost		cost per iteration	
$\lvert V\rvert$	run time	memory	run time	memory	run time	memory	run time	memory	run time	memory	run time	memory	run time	memory	run time	memory
100	30.3+67.7 ms	0.9 GB	12.8+4.8 ms	0.0 GB	99.4+109.2 ms	0.9 GB	3.1+3.8 ms	0.0 GB	245.9+170.7 ms	1.0 GB	4.8+1.4 ms	0.0 GB	18.5+8.1 ms	0.9 GB	9.3+1.8 ms	0.0 GB
200	86.9+87.6 ms	0.9 GB	11.4+4.4 ms	0.0 GB	283.0+186.6 ms	1.0 GB	0.0+0.7 ms	0.0 GB	457.0+315.7 ms	1.1 GB	0.0+7.2 ms	0.1 GB	53.9+28.5 ms	0.8 GB	27.0+11.7 ms	0.2 GB
300	114.1+116.7 ms	0.8 GB	0.0+0.0 ms	0.1 GB	388.3+269.0 ms	1.0 GB	0.2+8.5 ms	0.0 GB	780.6+524.6 ms	1.3 GB	19.7+6.4 ms	0.0 GB	153.2+71.3 ms	1.0 GB	40.5+8.5 ms	0.1 GB
400	153.7+128.8 ms	1.0 GB	2.3+2.1 ms	0.0 GB	550.3+346.2 ms	1.0 GB	0.0+3.6 ms	0.1 GB	951.1+683.3 ms	1.5 GB	71.5+11.0 ms	0.0 GB	185.3+108.6 ms	0.9 GB	120.5+47.9 ms	0.4 GB
500	174.4+132.8 ms	1.0 GB	0.0+7.1 ms	0.0 GB	643.9+441.8 ms	1.2 GB	37.1+3.4 ms	0.0 GB	1,289.7+913.4 ms	1.7 GB	0.0+10.1 ms	0.0 GB	353.5+166.8 ms	1.4 GB	0.0+32.0 ms	0.1 GB
600	212.1+167.4 ms	1.0 GB	3.5+4.5 ms	0.0 GB	816.5+574.7 ms	1.4 GB	39.8+0.3 ms	0.0 GB	1,551.3+1,149.1 ms	1.9 GB	53.9+25.4 ms	0.0 GB	356.0+231.7 ms	1.4 GB	61.1+54.2 ms	0.3 GB
700	251.7+184.5 ms	1.0 GB	6.4+0.9 ms	0.0 GB	914.1+657.4 ms	1.4 GB	65.2+8.3 ms	0.1 GB	1.8+1.4 s	2.1 GB	49.9+13.7 ms	0.0 GB	452.4+314.3 ms	1.8 GB	69.1+77.8 ms	0.2 GB
800	225.1+199.7 ms	1.0 GB	24.5+5.7 ms	0.0 GB	1,078.6+735.9 ms	1.6 GB	49.2+18.7 ms	0.0 GB	2.1+1.6 s	2.2 GB	60.1+22.2 ms	0.0 GB	527.2+410.4 ms	2.1 GB	113.0+80.5 ms	0.2 GB
900	304.3+207.3 ms	1.0 GB	12.6+6.8 ms	0.0 GB	1,174.9+850.1 ms	1.6 GB	75.2+4.1 ms	0.0 GB	2.3+1.7 s	2.1 GB	61.0+33.0 ms	0.1 GB	709.6+512.9 ms	2.5 GB	113.0+119.8 ms	0.2 GB
1000	340.1+232.7 ms	1.0 GB	4.9+4.9 ms	0.0 GB	1,324.7+1,011.6 ms	1.5 GB	62.7+10.6 ms	0.1 GB	2.6+1.9 s	2.6 GB	32.4+52.7 ms	0.0 GB	841.2+624.6 ms	2.8 GB	157.3+147.8 ms	0.3 GB
2000	643.1+442.4 ms	1.3 GB	6.1+8.2 ms	0.0 GB	2.7+2.1 s	2.7 GB	107.4+39.3 ms	0.0 GB	5.3+4.2 s	4.5 GB	90.2+96.8 ms	0.0 GB	3.3+2.6 s	8.2 GB	646.9+617.2 ms	1.4 GB
3000	1,001.7+704.1 ms	1.5 GB	15.4+13.5 ms	0.0 GB	4.3+3.4 s	3.8 GB	100.6+58.3 ms	0.1 GB	8.3+6.8 s	6.7 GB	242.6+126.0 ms	0.1 GB	7.2+5.8 s	18.2 GB	1.5+1.4 s	3.4 GB
4000	1,353.5+937.3 ms	1.7 GB	19.3+7.3 ms	0.0 GB	5.6+4.5 s	4.7 GB	139.2+72.9 ms	0.1 GB	11.1+9.0 s	8.4 GB	254.4+161.4 ms	0.2 GB	12.6+9.9 s	30.4 GB	2.8+2.8 s	5.7 GB
5000	1.7+1.3 s	1.9 GB	57.2+13.8 ms	0.1 GB	7.4+6.0 s	5.9 GB	139.6+116.9 ms	0.1 GB	14.7+12.0 s	10.9 GB	357.0+266.7 ms	0.2 GB	19.6+15.6 s	48.7 GB	4.0+4.0 s	9.1 GB
6000	2.0+1.5 s	2.2 GB	68.1+27.1 ms	0.1 GB	8.9+7.3 s	7.1 GB	181.2+93.5 ms	0.2 GB	17.9+14.6 s	13.6 GB	422.8+186.5 ms	0.2 GB	28.1+22.5 s	71.5 GB	5.5+5.9 s	13.6 GB
7000	2.4+1.8 s	2.5 GB	21.5+14.5 ms	0.0 GB	10.3+8.5 s	8.1 GB	276.1+141.2 ms	0.1 GB	21.0+16.8 s	15.2 GB	525.3+251.6 ms	0.3 GB	37.9+30.4 s	95.2 GB	7.6+8.5 s	17.5 GB
8000	2.7+2.0 s	2.6 GB	29.3+41.1 ms	0.1 GB	11.9+9.7 s	8.9 GB	251.5+169.0 ms	0.2 GB	24.3+19.1 s	17.2 GB	458.9+424.1 ms	0.3 GB				
9000	3.1+2.3 s	3.1 GB	61.8+35.2 ms	−0.0 GB	14.0+11.6 s	10.5 GB	338.8+173.6 ms	0.2 GB	28.7+22.8 s	20.4 GB	711.7+386.9 ms	0.4 GB				
10000	3.5+2.6 s	3.2 GB	73.5+46.3 ms	−0.0 GB	15.6+12.8 s	11.6 GB	371.5+224.3 ms	0.2 GB	32.4+25.5 s	22.6 GB	639.4+443.0 ms	0.5 GB				
20000	7.1+5.8 s	5.8 GB	160.0+59.0 ms	0.1 GB	34.7+27.0 s	24.4 GB	930.8+397.0 ms	0.4 GB	74.3+54.9 s	48.4 GB	1.4+0.9 s	0.9 GB				
30000	10.6+8.8 s	8.3 GB	212.1+97.9 ms	0.1 GB	53.9+41.1 s	35.5 GB	1.1+0.4 s	0.8 GB	2.0+1.4 min	70.0 GB	2.9+1.5 s	1.6 GB				
40000	14.9+12.4 s	11.2 GB	328.6+152.4 ms	0.2 GB	80.7+58.6 s	50.9 GB	1.8+1.0 s	0.9 GB	3.0+2.0 min	99.9 GB	3.5+2.0 s	1.8 GB				

Table 10: Run times and maximal observed memory allocation for our and prior [6] protocols for π_{R}^{D} with different sparsity, in a LAN setting. Run times are split into preprocessing + online.

	ours ($\lvert\mathcal{E}\rvert = 10\lvert V\rvert$)				**ours** ($\lvert\mathcal{E}\rvert = 50\lvert V\rvert$)				**ours** ($\lvert\mathcal{E}\rvert = 100\lvert V\rvert$)				prior [6] (arbitrary $\lvert\mathcal{E}\rvert$)			
	one-time cost		cost per iteration		one-time cost		cost per iteration		one-time cost		cost per iteration		one-time cost		cost per iteration	
$\lvert V\rvert$	run time	memory	run time	memory	run time	memory	run time	memory	run time	memory	run time	memory	run time	memory	run time	memory
100	105.3+72.9 ms	1.0 GB	49.2+34.0 ms	0.0 GB	175.2+159.5 ms	0.8 GB	218.9+149.4 ms	0.3 GB	361.8+266.7 ms	1.2 GB	453.3+295.3 ms	0.5 GB	26.7+14.4 ms	0.9 GB	239.1+170.2 ms	0.4 GB
200	232.3+158.0 ms	1.1 GB	200.3+123.6 ms	0.1 GB	586.8+473.8 ms	1.8 GB	843.9+585.0 ms	0.9 GB	1.1+0.9 s	2.4 GB	1.6+1.2 s	1.9 GB	166.3+63.5 ms	1.0 GB	1.2+1.3 s	2.8 GB
300	341.1+327.6 ms	1.3 GB	499.0+290.2 ms	0.5 GB	1.2+1.0 s	2.7 GB	1.9+1.3 s	1.9 GB	2.1+1.8 s	4.5 GB	3.6+2.7 s	4.0 GB	278.9+132.4 ms	0.9 GB	4.4+4.3 s	10.4 GB
400	593.4+553.8 ms	1.8 GB	813.7+485.7 ms	0.8 GB	1.9+1.7 s	4.0 GB	3.2+2.3 s	3.7 GB	3.4+3.1 s	7.3 GB	6.5+4.7 s	7.2 GB	426.2+234.8 ms	1.2 GB	10.3+10.1 s	24.3 GB
500	906.9+783.3 ms	2.2 GB	1.2+0.8 s	1.2 GB	2.7+2.7 s	5.8 GB	5.1+3.7 s	5.7 GB	5.0+5.4 s	10.7 GB	10.2+7.3 s	11.5 GB	572.5+324.9 ms	1.6 GB	19.8+19.7 s	47.8 GB
600	1.3+1.2 s	2.8 GB	1.7+1.1 s	1.8 GB	3.9+3.9 s	8.2 GB	7.3+5.3 s	8.2 GB	7.2+7.7 s	15.2 GB	14.7+10.9 s	16.3 GB	769.0+471.4 ms	1.6 GB	34.1+33.8 s	84.5 GB
700	1.7+1.5 s	3.5 GB	2.2+1.4 s	2.3 GB	5.1+4.7 s	10.9 GB	1,0.0+7.1 s	11.2 GB	9.7+10.9 s	20.4 GB	20.1+14.6 s	22.6 GB				
800	2.2+1.9 s	4.4 GB	2.8+1.9 s	3.1 GB	6.7+6.6 s	13.5 GB	13.2+9.2 s	14.6 GB	12.5+13.8 s	25.5 GB	28.3+19.5 s	29.1 GB				
900	2.7+2.5 s	5.1 GB	3.6+2.4 s	4.0 GB	8.4+8.6 s	17.2 GB	16.6+11.6 s	18.8 GB	15.7+18.0 s	32.6 GB	35.5+24.8 s	37.4 GB				