GraSP: Secure Collaborative Graph Processing Made Scalable

Siddharth Kapoor, Nishat Koti, Varsha Bhat Kukkala, Arpita Patra and Bhavish Raj Gopal

***Abstract*—Secure graph processing enables a set of n data owners or parties to jointly compute on a distributed graph while hiding its private topology and associated vertex and edge data. Existing solutions based on the technique of secure multiparty computation (MPC) face a fundamental trade-off between efficiency and security. The generic framework of Koti et al. (CCS 2024) can tolerate collusion up to $n-1$ parties, but its efficiency degrades as the number of parties grows. In contrast, the framework of Zou et al. (CCS 2025) achieves better scalability by adopting the restrictive assumption that no two parties collude.**

In this work, we present GraSP, a secure graph-processing framework that combines scalability with security against collusion among up to $n-1$ parties. At its core, GraSP introduces a novel Decompose-Compute-Combine (DCC) paradigm that partitions the global computation into party-specific subgraph computations while preserving privacy and correctness. Consequently, its secure realisation achieves a round complexity independent of both the graph size and the number of parties, while retaining communication complexity linear in the graph size and the number of parties. We implement GraSP and evaluate ten iterations of end-to-end PageRank computation. Compared with the generic framework of Koti et al., GraSP reduces runtime by up to 88.24× and communication by up to 51.95×. Compared with Zou et al., GraSP improves runtime and communication by up to 3.70× and 2.61×, respectively, despite providing a substantially stronger security guarantee. We further demonstrate the expressiveness of GraSP by securely implementing anti-money-laundering analytics, such as transaction-risk propagation and group-connection detection, for collaborative analysis. For a graph of size 2^{22} distributed across eight parties, GraSP completes ten-hop group-connection detection in 78.40 seconds and ten-iteration risk propagation in 202.63 seconds.

***Index Terms*—Secure graph processing, secure multiparty computation**

I. Introduction

Graphs are fundamental for representing and analysing complex relationships in diverse real-world applications. In many applications, these graphs contain sensitive user information and are usually distributed across multiple data owners. In such scenarios, processing the global graph while hiding its topology and associated sensitive data is challenging. For example, consider a financial transaction graph in which nodes (or vertices) represent bank accounts and edges represent transactions. Each bank (data owner) is privy to only part of the transaction graph, related to their own registered accounts (vertices) and the corresponding transactions (edges). Nonetheless, banks would be interested in processing the global transaction graph for various purposes, such as identifying fraudulent or money-laundering transactions and analysing risk. Particularly in money-laundering/fraudulent transactions, illicit funds could be routed through multiple accounts and banks to obscure their origin, and no individual institution may observe the complete flow. Detecting such activity may require banks to jointly propagate risk scores through the transaction network [1] or determine whether a bounded-length funding path connects a known high-risk source group to a suspicious target group [2]. However, directly sharing account relationships, transaction values, or intermediate risk scores may reveal sensitive customer and business information. Thus, there is a need to design techniques that allow data owners to collaboratively process sensitive, distributed graph data while ensuring individual subgraph data and topologies are not revealed. We refer to this as *secure collaborative graph processing*. With this motivation, the current work aims to design an efficient framework that enables multiple data owners with private subgraphs to collaboratively process the global graph, such that each data owner learns no additional information beyond the final result. We rely on the technique of secure multiparty computation (MPC) to design a solution for this. This technique allows a set of n parties to compute a function on their private inputs such that even if some t parties, with $t < n$, collude with one another, they learn no information beyond the output. In the context of secure collaborative graph processing via MPC, data owners can act as parties, with their local views of the graph as private inputs, and the graph computation corresponds to the function. The efficiency of MPC protocols is commonly evaluated using three parameters: communication, rounds and local computation. Communication complexity measures the total data exchanged among the parties, while round complexity measures the number of sequential communication steps. Local computation captures the work performed by each party. Therefore, an efficient, secure collaborative graph-processing framework should minimise communication, interaction rounds, and computation.

Related Works. To motivate our work and contextualise our contributions, we first outline the existing approaches.

There are two natural ways to design MPC-based frameworks for secure collaborative graph processing. In the first, called the *outsourced setting*, data owners do not directly

Siddharth Kapoor is affiliated with the University of Utah (email: siddharth00kapoor@gmail.com). Nishat Koti is an independent researcher (email: kotinishat@gmail.com). Varsha Bhat Kukkala is affiliated with the Indian Institute of Technology, Tirupati (email: varshabhat15@gmail.com). Arpita Patra and Bhavish Raj Gopal are affiliated with the Indian Institute of Science, Bangalore (email: arpita@iisc.ac.in, bhavishraj@iisc.ac.in).

0000–0000/00$00.00 © 2021 IEEE

participate in the MPC computation. Instead, they delegate these computations to a small set of servers hired on a pay-per-use basis. Privacy is ensured as long as at least one server remains honest and does not collude with other servers or data owners. However, this model may not be suitable for all scenarios, as it requires data owners to relinquish direct control over their data and trust that the non-collusion assumption among external servers holds. Further, the data owners may be distributed across regions with stringent privacy regulations that restrict data movement beyond regional boundaries. In such cases, it may be infeasible to identify and agree on a small set of servers to outsource the computation. In contrast, a non-outsourced multiparty computation setting addresses these issues. Here, data owners themselves serve as MPC parties, retaining complete control over their data. Specifically, the privacy of an honest data owner is guaranteed, even if all other data owners collude. Thus, a non-outsourced setting may be more desirable in scenarios involving multiple data owners with strict privacy requirements, such as a financial transaction network spanning multiple banks discussed earlier.

Several works in the literature design generic graph computation frameworks [3]–[5] (capable of realising a large class of graph algorithms). These are designed to leverage the sparsity of real-world graphs and operate on a list-based graph representation to securely realise any message-passing graph algorithm. The latter are stateful iterative algorithms where each node/edge maintains state information. Each iteration involves the following: (i) propagating information from vertices onto their incident edges; (ii) receiving and aggregating data from incident edges onto vertices; (iii) using the aggregated data to update the current state of each vertex. These operations performed in each iteration of the message-passing algorithm are referred to as Scatter, Gather and Apply respectively. Such generic frameworks are conceptually applicable to both deployment models: in the outsourced setting, data owners *secret-share*[1] their local view of the graph to external servers that execute the MPC protocol, whereas in the non-outsourced setting, the data owners themselves do so. Among these generic graph computation frameworks, Graphiti [5] forms the state-of-the-art.

As an alternative to the above, a few recent works study secure graph computation specifically in the non-outsourced setting. CoGNN [6] proposes a solution for secure training and inference of graph neural networks (GNNs) in a multiparty setting. While their framework can, in principle, support any graph algorithm, it assumes a restrictive threat model in which no two participating parties collude. This weaker assumption directly enables efficiency optimisations in CoGNN [6]. RingSG [2] builds on [6] and obtains communication complexity independent of the number of parties, meaning that the total communication does not grow with the number of participating data owners. However, it still relies on the same restrictive no-collusion assumption.

Motivation. While specialised non-outsourced solutions such as [6] and [2] demonstrate efficiency, their security model is significantly weaker than the standard MPC guarantee considered in our work, in which the privacy of an honest data owner should hold even if some or all other data owners collude. Such restrictive assumptions can be impractical in collaborative settings involving many mutually distrustful organisations such as banks, telecom providers, or cybersecurity firms. For example, in cross-institution anti-money-laundering investigations, participating banks are commercially competing entities, and the privacy of one bank's transaction graph should not depend on every other pair of banks remaining non-colluding. On the other hand, generic secure graph-processing frameworks such as [5] can, in principle, provide stronger security guarantees in the non-outsourced setting by being instantiated with a generic n-party MPC protocol, where the data owners themselves act as the MPC parties. However, this comes at a significant efficiency cost, as they incur a round complexity that depends on the number of data owners and the graph size. Further, they incur a communication cost that is log-linear in the graph size. This inefficiency is especially problematic when parties are geographically distributed over a WAN, where high round-trip time and limited bandwidth amplify the cost of protocols with high round and communication complexity.

This reveals a gap in the existing literature, which the current work aims to bridge by designing a secure graph-processing framework for multiple data owners that supports strong privacy guarantees against collusion among all but one data owner while also ensuring efficiency.

A. Contributions

We design GraSP (Graph Secure Processing), an efficient end-to-end framework for secure graph processing in the distributed setting to securely realise any graph algorithm that can be expressed as a message-passing graph algorithm.

- We first design a novel *data-oblivious*[2] paradigm for efficiently executing message-passing graph algorithms with multiple data owners.
- We then translate this paradigm into a secure collaborative graph-processing framework using generic MPC protocols from the literature, as well as newly designed ones. Our framework achieves low round complexity, independent of both the number of data owners and the graph size, and communication complexity that is linear in these parameters, resulting in the desired scalability.
- We demonstrate the applicability of GraSP beyond conventional graph analytics by implementing and benchmarking collaborative anti-money-laundering workloads such as risk propagation and group-connection detection. We elaborate on all the above points next.

Since efficiency takes centre stage, GraSP leverages the preprocessing paradigm in MPC, where input-independent MPC computations are offloaded to a preprocessing phase

[1]Secret-sharing enables data-owner to split a private value into multiple random shares such that no individual server or a subset of servers learn anything, but all servers together can reconstruct the value.

[2]The access pattern and communication patterns depend only on public parameters, such as the graph size or number of vertices and edges, and not on the private graph topology or the intermediate values computed during the algorithm. This ensures that the access pattern does not leak sensitive information about the underlying graph or its topology.

so that input-dependent computations performed in the online phase can be carried out extremely fast. The distinction between the two phases is crucial in practice, as the online phase determines the protocol's response time, while the preprocessing phase can be performed well in advance without impacting the response time. Given the significance of achieving a fast response time, unless otherwise stated, the complexities stated subsequently refer to the online complexities. Our contributions are detailed below.

Novel design paradigm of GraSP: We design a new and efficient Decompose-Compute-Combine (DCC) paradigm for data-oblivious realisation of any message-passing graph algorithm on the distributed global graph $\mathcal{G} = (\mathsf{V}, \mathsf{E})$. The paradigm involves performing: (i) decomposing $\mathcal{G}$ into n subgraphs $\mathcal{G}_i$ for $i = 1$ to n, where $\mathcal{G}_i$'s topology is known to a distinct party P_i, (ii) carrying out message-passing computations on the individual subgraphs $\mathcal{G}_i$ in parallel, instead of doing the same on the global graph $\mathcal{G}$, (iii) combining results on these subgraphs to obtain results on the global graph $\mathcal{G}$. GraSP leverages the fact that each party (data owner) has knowledge of its own vertices, edges incident on them, and the neighbouring vertices, all of which comprise the subgraph $\mathcal{G}_i$ to be operated on. The DCC paradigm is designed to yield significant efficiency benefits when translated to the MPC setting. In particular, steps (i)-(iii) aid in achieving a round complexity *independent of the number of parties* n, allowing GraSP to scale efficiently for multiparty settings.

Secure realisation of DCC paradigm. Our secure realisation of message passing through DCC substantially improves upon prior secure graph-processing frameworks. A detailed comparison of the round complexity, communication complexity, and security guarantees appears in Table I.

Compared with the state-of-the-art generic graph-computation framework Graphiti [5], GraSP provides the following improvements:

- *Lightweight initialisation.* Similar to prior generic graph computation works [4], [5], our secure realisation consists of a lightweight one-time initialisation phase followed by repeated message-passing computations. Graphiti's initialisation phase requires securely sorting the list-based representation of the global graph. In contrast, GraSP avoids secure sorting altogether. Consequently, the initialisation of GraSP requires only a constant number of online rounds, independent of both the number of parties n and the graph size N,
- *Improved message-passing.* In Graphiti, each iteration requires securely reordering the secret-shared graph list using a secure shuffle protocol. The protocol randomly permutes a secret-shared list without revealing the permutation to any party. In the multiparty setting, such secure shuffles incur $O(n)$ rounds [7], and hence the message-passing round complexity of Graphiti [5] becomes $O(n)$. In contrast, in the secure realisation of DCC, message-passing computations are performed over subgraphs whose topology is already known to the corresponding data owner. Hence, GraSP does not need a full secure shuffle with a permutation hidden from every party. Leveraging this, GraSP's message-passing phase relies on a more efficient Permute and Share primitive, where a secret-shared list is permuted according to a random permutation known to a designated party. This can be realised in $O(1)$ rounds in the multiparty setting.

Ref.	Rounds	Communication	Collusion threshold
Graphiti [5]	$\boldsymbol{O(n + \log \mathsf{N})} + k \cdot O(n)$	$\boldsymbol{O(n\mathsf{N} \log \mathsf{N})} + O(k \cdot n \cdot \mathsf{N})$	$\leq n-1$
RingSG [2]	$k \cdot O(\log \lvert\mathsf{E}\rvert)$	$O(k \cdot \mathsf{N})$	≤ 1
GraSP (Ours)	$\boldsymbol{O(1)} + k \cdot O(1)$	$\boldsymbol{O(n\mathsf{N})} + O(k \cdot n \cdot \mathsf{N})$	$\leq n-1$

For $\mathcal{G} = (\mathsf{V}, \mathsf{E})$, $\mathsf{N} = \lvert\mathsf{V}\rvert + \lvert\mathsf{E}\rvert$.

TABLE I: Comparison of the initialisation (in bold) and k message-passing iterations.

Compared with the specialised RingSG framework [2], GraSP achieves better round complexity, which is independent of the graph size. Although RingSG has asymptotically lower communication, it is worth noting that this is attributed to its weak security model, where no two parties can collude, whereas GraSP provides security against collusion among up to $n-1$ parties. Finally, as corroborated by our benchmarks, RingSG has a higher concrete communication cost, and we observe that GraSP communicates less than or remains on par with RingSG for all practically evaluated values of n.

Design of new MPC protocols: Naively using Permute and Share for secure realisation of DCC results in round complexity that is independent of n, but it introduces a quadratic communication overhead in the decompose step, i.e., $O(n^2 \cdot \mathsf{N})$. This is because our design requires permuting the same input list according to n different permutations, each known to a distinct party. A naive approach requires n parallel invocations of Permute and Share, with each invocation requiring $O(n \cdot \mathsf{N})$ communication. Instead, by leveraging the fact that the input list remains the same across all n invocations, we design Amortised Permute and Share that significantly reduces the online complexity. This protocol has same online communication as that of a single invocation of Permute and Share while maintaining constant round complexity. Elaborately, the Amortised Permute and Share protocol takes as input a secret-shared list and outputs n secret-shared lists, where each list is permuted according to a permutation known to a distinct party. Note that the Amortised Permute and Share protocol can be of independent interest, as it enables multiple permutations to be performed on a secret-shared list efficiently.

Implementation, evaluation, and applications. We implement GraSP and evaluate ten iterations of end-to-end PageRank computation over distributed graphs, comparing against both Graphiti [5] and RingSG [2]. Our experiments demonstrate that GraSP substantially reduces runtime and communication while supporting graph sizes and party counts that the prior generic approach could not handle. Compared with Graphiti, GraSP achieves up to an 88.24× runtime improvement for ten parties and a graph of size 2^{19}, and up to a 51.95× communication reduction for five parties and a graph of size 2^{20}. For larger graph sizes of 2^{21} and 2^{22}, Graphiti runs out of memory during initialisation, while GraSP successfully completes the computation. Compared with RingSG, GraSP achieves improvements of up to 3.70× in runtime and 2.61× in communication for three parties, with each party owning 2^{16} vertices. Beyond PageRank, we demon-

strate the expressiveness of GraSP by securely realising two representative algorithms for cross-institution anti-money-laundering analysis: transaction-weighted risk propagation and group-connection detection; see Appendix C. For a graph of size 2^{22} distributed across eight parties, GraSP performs ten-hop group-connection detection in under 85 seconds and ten-iteration risk propagation in approximately 203 seconds.

II. Preliminaries

System and Threat model: We consider a directed graph $\mathcal{G} = (\mathsf{V}, \mathsf{E})$ that is distributed across n parties (data owners) in $\mathcal{P}$. Each party $P_i \in \mathcal{P}$ *owns* a subset of vertices or nodes, denoted as V_i, such that $\mathsf{V} = \bigcup_{i=1}^{n} \mathsf{V}_i$. Each vertex $\mathsf{v} \in \mathsf{V}$ is associated with a public label, $\mathsf{v.id}$, where the label as well as the information about which party owns this vertex is known to all parties. Thus, V is known to all the parties. Each vertex v may also be associated with a data component denoted as $\mathsf{v.data}$, and is private to the party that owns v. Each directed edge $\mathsf{e} \in \mathsf{E}$ consists of $(\mathsf{e.src}, \mathsf{e.dst})$, representing the source and destination label, respectively. An edge e may also have a data component denoted as $\mathsf{e.data}$. Information about existence of an edge e where both $\mathsf{e.src}, \mathsf{e.dst} \in \mathsf{V}_i$, is known only to P_i. Its associated data $\mathsf{e.data}$ is also private only to P_i. For edges where $\mathsf{e.src} \in \mathsf{V}_i$ and $\mathsf{e.dst} \in \mathsf{V}_j$, the existence of e and its associated data $\mathsf{e.data}$ is known to both P_i and P_j. We let E_i represent the set of all edges $\mathsf{e} \in \mathsf{E}$ where $\mathsf{e.dst} \in \mathsf{V}_i$, and have $\mathsf{E} = \bigcup_{i=1}^{n} \mathsf{E}_i$.

The parties are interested in performing computations on the global graph $\mathcal{G}$, while ensuring their graph data remains private from other parties. We assume that the parties are each connected via pairwise private and authenticated channels in a synchronous network. In this work, we consider the parties to be honest-but-curious or semi-honest, where they follow the protocol honestly while trying to gain more information. Furthermore, we also consider that any subset of up to $n-1$ parties may collude to learn information about the inputs of other parties. We model this using a static, semi-honest, probabilistic, polynomial-time adversary $\mathcal{A}$ capable of corrupting up to $n-1$ parties in $\mathcal{P}$. We prove security using the standard real-world/ideal-world simulation paradigm.

Secret-Sharing and Multiparty Computation: Firstly, we assume that the data owners map the graph data to elements on a ring algebraic structure, with $\mathbb{Z}_{2^\ell}$ denoting the ring of ℓ-bit integers (see Appendix A-C). We say a value $\mathsf{x} \in \mathbb{Z}_{2^\ell}$ is $\langle\cdot\rangle$-shared (or additively shared) over $\mathbb{Z}_{2^\ell}$ if each party $P_i \in \mathcal{P}$ holds a random share $\langle \mathsf{x} \rangle_i \in \mathbb{Z}_{2^\ell}$ such that $\mathsf{x} = \langle \mathsf{x} \rangle_1 + \cdots + \langle \mathsf{x} \rangle_n$. This secret sharing scheme is linear since, given shares of $\mathsf{x}, \mathsf{y} \in \mathbb{Z}_{2^\ell}$ and public constants $\mathsf{c}_1, \mathsf{c}_2 \in \mathbb{Z}_{2^\ell}$, parties can non-interactively generate shares of $\mathsf{c}_1\mathsf{x} + \mathsf{c}_2\mathsf{y}$. However, non-linear operations such as multiplication or comparison require communication among the parties. We rely on a one-time key setup [8]–[12] that establishes common random keys for a pseudo-random function (PRF) among the parties. Using these keys, subsets of parties can non-interactively sample a common random ℓ-bit string $\mathsf{v} \in \mathbb{Z}_{2^\ell}$. This enables a party to non-interactively generate $\langle\cdot\rangle$-shares of a value x as described in Appendix A. However, the reconstruction of a $\langle\cdot\rangle$-shared value towards P_i requires interaction where every party communicates its share to P_i.

To enable secure computation over shares, we rely on a secret-sharing-based MPC protocol [13] which allows the parties to perform computation on secret-shared inputs to generate secret-shared outputs. Specifically, we rely on the following primitive operations, which are abstracted via ideal functionalities. (i) Multiplication: This primitive takes as input two $\langle\cdot\rangle$-shared values $\mathsf{x}, \mathsf{y} \in \mathbb{Z}_{2^\ell}$ and outputs the $\langle\cdot\rangle$-shares of $\mathsf{z} = \mathsf{x} \cdot \mathsf{y}$. $\mathcal{F}_{\mathsf{mult}}$ denotes the ideal functionality for the same. (ii) Permute and share: This primitive takes as input a $\langle\cdot\rangle$-shared vector $\mathsf{T} \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$ of size N, where each element in T, i.e., $\mathsf{T}[i], \forall i \in \{1, \ldots, \mathsf{N}\}$, is $\langle\cdot\rangle$-shared and a permutation π from one party. It outputs $\langle\cdot\rangle$-shares of a vector $\mathsf{T_O} \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$ such that $\mathsf{T_O} = \pi(\mathsf{T})$. $\mathcal{F}_{\mathsf{PS}}$ denotes the ideal functionality for it. (iii)The underlying MPC: Note that our protocols may require various other MPC primitives such as comparison, equality etc. depending on the considered graph algorithm. We abstract these out via the $\mathcal{F}_{\mathsf{MPC}}$ functionality, which takes secret-shared inputs from the parties and returns a secret-shared output of the function computed on the inputs. We instantiate $\mathcal{F}_{\mathsf{MPC}}$ via the generic MPC protocol of [13] in the semihonest setting.

Secure Shuffle, Permute and Share. Let $\mathsf{T} \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$ be a list of N entries, where each entry is from $\mathbb{Z}_{2^\ell}$. The list T is additively secret-shared *entry-wise* among all parties in $\mathcal{P}$. Let $\langle \mathsf{T} \rangle_i \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$ denote the share of party $P_i \in \mathcal{P}$. Secure shuffle's aim is to take $\langle\cdot\rangle$-shares of T as input and generate $\langle\cdot\rangle$-shares of a shuffled list $\mathsf{T_O} \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$, where $\mathsf{T_O} = \pi(\mathsf{T})$, i.e., T shuffled using a random secret permutation π (not known to any party) where $\mathsf{T_O}[i] = \mathsf{T}[\pi(i)]$.

Secure shuffle can be constructed using Permute and Share, as described in [7], [14]. The Permute and Share protocol allows shuffling a secret-shared list using a permutation known to one party. Specifically, the Permute and Share protocol takes as input a secret-shared list T and a permutation π_i known to a designated party P_i, and outputs $\langle \mathsf{T_O} \rangle$ such that $\mathsf{T_O} = \pi_i(\mathsf{T})$. The Permute and Share protocol described in [14] consists of two phases: preprocessing and online phase. During the preprocessing phase, the parties generate additive secret shares of a random list R (not known to any party) and its permuted counterpart $\pi_i(\mathsf{R})$. These shares can be generated using pairwise invocation (between P_i and every other party) of a 2-party Share Translation protocol, as detailed in [7], [14]. Each invocation of Share Translation protocol incurs a communication of $O(\mathsf{N})$, making the total preprocessing communication cost for n invocations $O(n\mathsf{N})$. Given, $\langle \mathsf{R} \rangle, \langle \pi_i(\mathsf{R}) \rangle$ generated in the preprocessing, the parties reconstruct $\mathsf{T} + \mathsf{R}$ towards the designated party P_i. All parties except P_i then set their output shares of $\mathsf{T_O}$ as their respective shares of $\pi_i(\mathsf{R})$ i.e party P_j for $j \neq i$ sets $\langle \mathsf{T_O} \rangle_j = -\langle \pi_i(\mathsf{R}) \rangle_j$. Party P_i computes its output share as $\pi_i(\mathsf{T} + \mathsf{R}) - \langle \pi_i(\mathsf{R}) \rangle_i$. This ensures that the output list $\mathsf{T_O}$ satisfies $\mathsf{T_O} = \pi_i(\mathsf{T})$. This requires one round and $O(n\mathsf{N})$ communication.

Secure shuffle can be realised by performing n sequential invocations of Permute and Share, where each party contributes a random permutation ensuring that the final permutation applied is a composition of all permutations that

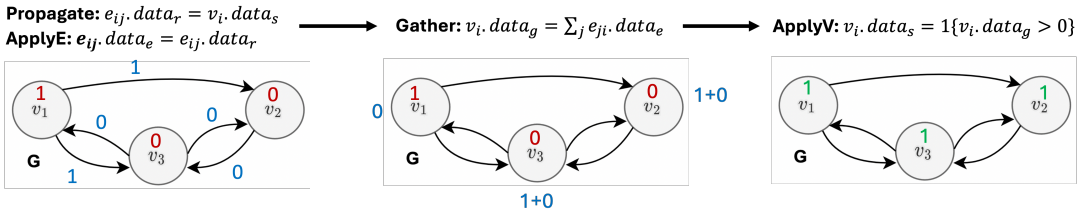


Fig. 1: Example of one message-passing iteration for identifying the one-hop neighbours of source vertex v_1. Repeating the computation for k iterations identifies vertices reachable from v_1 within at most k hops.

is not known to any party. This approach results in $O(n)$ rounds of communication and $O(n^2\mathsf{N})$ total communication cost during the online phase. The preprocessing cost similarly involves $O(n^2)$ invocations of Share Translation, resulting in a communication cost of $O(n^2\mathsf{N})$. The work of [7] improves this by optimising the online phase communication to $O(n\mathsf{N})$ while maintaining the same rounds and preprocessing costs. We refer interested readers to [7] for further details.

Secure computation of graph algorithms: A broad class of graph algorithms, including BFS [4], [5], DFS, PageRank, centrality and clustering algorithms [15], and graph convolutional networks [16], can be naturally expressed as iterative message-passing computations. We next describe the list-based representation and data-oblivious message-passing approach used by generic graph-computation frameworks [3]–[5].

*Data-augmented list representation (*DAG-list*).* The input graph $\mathcal{G} = (\mathsf{V}, \mathsf{E})$ is represented as a data-augmented graph list, or DAG-list, denoted by G. The list contains one entry for every vertex and edge in $\mathcal{G}$. Each entry is a tuple $(\mathsf{src}, \mathsf{dst}, \mathsf{isV}, \mathsf{data})$, where src and dst denote the source and destination identifiers, isV distinguishes vertex entries from edge entries, and data stores the state associated with the corresponding vertex or edge. Specifically, a vertex u is represented as $(\mathsf{u}, \mathsf{u}, 1, \mathsf{data})$, while a directed edge $\mathsf{e} = (\mathsf{u}, \mathsf{v})$ is represented as $(\mathsf{u}, \mathsf{v}, 0, \mathsf{data})$. The data field may contain several sub-components, some of which are used only for vertex entries and others only for edge entries. To avoid revealing whether an entry is a vertex or an edge through its representation, every entry contains all sub-components, with irrelevant components set to zero or dummy values. An undirected edge is represented using two directed-edge entries, one in each direction.

Data-oblivious computations on DAG-list G. A key requirement for the secure realisation of a graph algorithm is that its execution reveal no information about the underlying graph topology beyond what is explicitly permitted by the security model. A data-oblivious algorithm ensures that the sequence of memory accesses, the control flow, and the order in which entries of the DAG-list are processed depend only on public parameters, such as $|\mathsf{V}|$ and $|\mathsf{E}|$, and not on the actual graph structure or intermediate values. Thus, securely realising a graph algorithm essentially involves two steps: first, translating the underlying algorithm into an equivalent data-oblivious algorithm, and second, securely evaluating the operations of this data-oblivious algorithm using MPC.

Generic secure graph-processing frameworks realise a message-passing iteration using the following primitives: (i) Propagate, which transfers a value stored at each vertex to its outgoing edges; (ii) ApplyE, which applies a user-defined function f_{AE} to the state of each edge and the value propagated from its source vertex; (iii) Gather, which aggregates the updated messages on the incoming edges of each vertex; and (iv) ApplyV, which applies a user-defined function f_{AV} to the current vertex state and the gathered aggregate[3].

Algorithm 1 presents the case where incoming messages are aggregated using addition. To realise these primitives efficiently, the framework of [5] uses three orderings of the DAG-list. In the *vertex* order, all vertex entries in the list appear first, followed by all edge entries. In the *source* order, every vertex entry in the list appears immediately before entries of all its outgoing edges. Similarly, in the *destination* order, entries of all the incoming edges of a vertex appear immediately before the entry of the corresponding vertex. An iteration of message-passing begins with G in vertex order. Propagate first difference-encodes the values stored in $\mathsf{data_s}$ at the vertex entries and stores the result in the temporary component $\mathsf{data_r}$. After reordering the list to source order, a prefix sum over $\mathsf{data_r}$ recovers the source vertex's propagated value at every outgoing edge. The subtraction of $\mathsf{data_s}$ ensures that the propagated value is zero at vertex entries. Next, ApplyE computes the updated edge message $\mathsf{data_e}$ from the previous edge state and the propagated source value. Multiplication by $1 - \mathsf{isV}$ ensures that $\mathsf{data_e}$ is zero at vertex entries. The list is then reordered to destination order, where Gather computes prefix sums of the updated edge messages $\mathsf{data_e}$. After returning to vertex order, differences between consecutive cumulative values recover the aggregate of incoming messages for each vertex and store it in $\mathsf{data_g}$. Finally, ApplyV updates each vertex state $\mathsf{data_v}$ using its current state and the corresponding aggregate $\mathsf{data_g}$. Taken together, these operations provide a data-oblivious realisation of one message-passing iteration. A pictorial example appears in Fig. 1. We refer the reader to Appendix A-D and [5] for further intuition and correctness argument.

Secure realisation of message-passing computations. Secure realisation of Algorithm 1 using MPC on a secret-shared DAG-list G enables the secure realisation of one iteration of message-passing computations and subsequently the graph algorithm. In the secure realisation of Propagate and Gather, the computations performed on the DAG-list consist of linear operations and hence do not require interactions between

[3]The operations Propagate and ApplyE are together commonly referred to as Scatter. Note that the framework can analogously be adapted to propagate and aggregate information in the reverse edge direction.

Algorithm 1: One iteration of message-passing

Input: DAG-list G in vertex order. Each entry contains data components: $\mathsf{data_s}, \mathsf{data_g}, \mathsf{data_v}, \mathsf{data_e}, \mathsf{data_r}$. Here, $\mathsf{data_s}$ denotes data to be propagated from vertices to edges, $\mathsf{data_g}$ aggregates incoming messages, and $\mathsf{data_v}, \mathsf{data_e}$ denote vertex and edge data. $\mathsf{data_r}$ is (temporary) to hold intermediate computations. Aggregation operation is addition.

Output: G in vertex order with updated states.

/* --- **Propagate operation** --- */

1 $\mathsf{G}[1].\mathsf{data_r} \leftarrow \mathsf{G}[1].\mathsf{data_s}$;

2 **for** $i \leftarrow 2$ **to** $|\mathsf{V}|$ **do**

3 $\quad \mathsf{G}[i].\mathsf{data_r} \leftarrow \mathsf{G}[i].\mathsf{data_s} - \mathsf{G}[i-1].\mathsf{data_s}$;

4 **for** $i \leftarrow |\mathsf{V}|+1$ **to** N **do**

5 $\quad \mathsf{G}[i].\mathsf{data_r} \leftarrow 0$;

6 Reorder G to source order;

7 $\rho \leftarrow 0$;

8 **for** $i \leftarrow 1$ **to** N **do**

9 $\quad \rho \leftarrow \rho + \mathsf{G}[i].\mathsf{data_r}$;

10 $\quad \mathsf{G}[i].\mathsf{data_r} \leftarrow \rho - \mathsf{G}[i].\mathsf{data_s}$;

/* --- **ApplyE operation** --- */

11 **for** $i \leftarrow 1$ **to** N **do**

12 $\quad \mathsf{G}[i].\mathsf{data_e} \leftarrow f_{\mathsf{AE}}(\mathsf{G}[i].\mathsf{data_e}, \mathsf{G}[i].\mathsf{data_r}) \cdot (1 - \mathsf{G}[i].\mathsf{isV})$;

/* --- **Gather operation** --- */

13 Reorder G to destination order;

14 $\gamma \leftarrow 0$;

15 **for** $i \leftarrow 1$ **to** N **do**

16 $\quad \gamma \leftarrow \gamma + \mathsf{G}[i].\mathsf{data_e}$;

17 $\quad \mathsf{G}[i].\mathsf{data_r} \leftarrow \gamma$;

18 Reorder G to vertex order;

19 $\mathsf{G}[1].\mathsf{data_g} \leftarrow \mathsf{G}[1].\mathsf{data_r}$;

20 **for** $i \leftarrow 2$ **to** $|\mathsf{V}|$ **do**

21 $\quad \mathsf{G}[i].\mathsf{data_g} \leftarrow \mathsf{G}[i].\mathsf{data_r} - \mathsf{G}[i-1].\mathsf{data_r}$;

/* --- **ApplyV operation** --- */

22 **for** $i \leftarrow 1$ **to** $|\mathsf{V}|$ **do**

23 $\quad \mathsf{G}[i].\mathsf{data_v} \leftarrow f_{\mathsf{AV}}(\mathsf{G}[i].\mathsf{data_v}, \mathsf{G}[i].\mathsf{data_g})$;

parties when realised via MPC, i.e., given secret-shares of the DAG-list, the computations involved in steps 1-5, 7-10, 14-17 and 19 can be performed non-interactively by leveraging the linearity of secret-sharing. Thus, the only interactive operations involved are reordering the DAG-list between different orderings (in steps 6, 13, and 18) and updating the edge and vertex data (in steps 12 and 23). The computations in ApplyE (step 12) and ApplyV (step 23) can be realised in parallel for each entry of the DAG-list G and are dependent on the functions f_{AE} and f_{AV}, respectively. A straightforward approach to reorder the DAG-list is to use a secure sorting protocol [3]. However, subsequent work has proposed optimisations to avoid repeated sorting. The key observation in [4] is that the sequence of required reordering (vertex→ source→ destination→ vertex) remains fixed across all message-passing iterations. Therefore, these permutations can be securely precomputed once during an initialisation phase using a secure sort. Thereafter, reordering can be performed by applying the permutation using secure shuffle (see Appendix A-C for further details).

Notations. For clarity of presentation, we summarise the key symbols and terminology used throughout in Table II.

III. DCC APPROACH FOR MESSAGE PASSING

Consider one iteration of message-passing computation on a graph $\mathcal{G}$ that is distributed across n-data owners. Recall that this involves performing— (i) Propagate, (ii) ApplyE (iii) Gather, and (iv) ApplyV as described in §II. Observe that the state update of a vertex for this iteration depends only on the current data in its immediate neighbours and the corresponding incoming edges. Therefore, for all vertices V_i owned by a party P_i, their state update depends on the neighbours of V_i and the edges with destination in V_i. Consequently, to update the state of all vertices in V, it suffices to independently update the state of vertices in V_i for each $i \in [n]$, and then collect these updates to get the updated state of V. To facilitate this, we define subgraphs $\mathcal{G}_i = (\bar{\mathsf{V}}_i, \mathsf{E}_i)$ for $i \in [n]$, where $\bar{\mathsf{V}}_i$ denotes the set of vertices in V_i along with their immediate neighbours and E_i denote the set of edges with destination in V_i. Note that each $\mathcal{G}_i$ is a subgraph of $\mathcal{G}$ and is typically smaller than the global graph. Despite containing fewer entries than $\mathcal{G}$, assuming that the state of all the vertices in $\mathcal{G}_i$ is up to date, $\mathcal{G}_i$ contains sufficient information such that performing one iteration of message-passing allows the state of the vertices in V_i to be correctly updated. Thus, realising one iteration of message-passing computations on $\mathcal{G}$ via the DCC approach entails the following: (1) **Decompose:** Generating subgraphs $\mathcal{G}_i$ for $i \in [n]$ from the global graph $\mathcal{G}$. (2) **Compute:** Performing one iteration of message passing computations on each $\mathcal{G}_i$ for $i \in [n]$ to update the states of V_i. (3) **Combine:** Extracting the updated state of V_i from $\mathcal{G}_i$ for $i \in [n]$ and combining them to get the updated V.

A pictorial illustration of DCC approach for message-passing computations in cleartext appears in Fig. 2.

A. Data Oblivious Realisation of DCC

To securely realise the DCC paradigm on a secret-shared graph, we begin with a data-oblivious algorithm—one whose access patterns and instruction flow depend *only* on public parameters (e.g., $|\mathsf{V}_i|$, $|\mathsf{E}_i|$, n), and not on the actual topology of the graph.

Input representation: The data augmented input graph $\mathcal{G} = (\mathsf{V}, \mathsf{E})$ (see §II) is represented as a set of lists $\mathsf{G} = (\mathsf{V}, \{\mathsf{E}_i\}_{i\in[n]})$. Further, V is ordered as $\mathsf{V}_1||\mathsf{V}_2||\ldots||\mathsf{V}_n$, where V_i denotes all the vertices belonging to P_i and $||$ denotes concatenation. E_i for $i \in [n]$, represents the subset of edges in E that have destination in V_i (vertices belonging to P_i). Similar to prior works [5], each entry in V and $\{\mathsf{E}_i\}_{i\in[n]}$ consist of a tuple $(\mathsf{src}, \mathsf{dst}, \mathsf{isV}, \mathsf{data})$ of four components: source (src), destination (dst), is_vertex (isV), and data. In addition to the public vertex labels and ownership information, we consider n and $\{|\mathsf{V}_i|, |\mathsf{E}_i|\}_{i\in[n]}$ to be public, similar to [2], [6]. For a fixed set of public parameters, this input representation is data-oblivious and independent of the graph topology.

At a high level, given an input graph G, the data-oblivious algorithm for an iteration of DCC involves (i) generating the

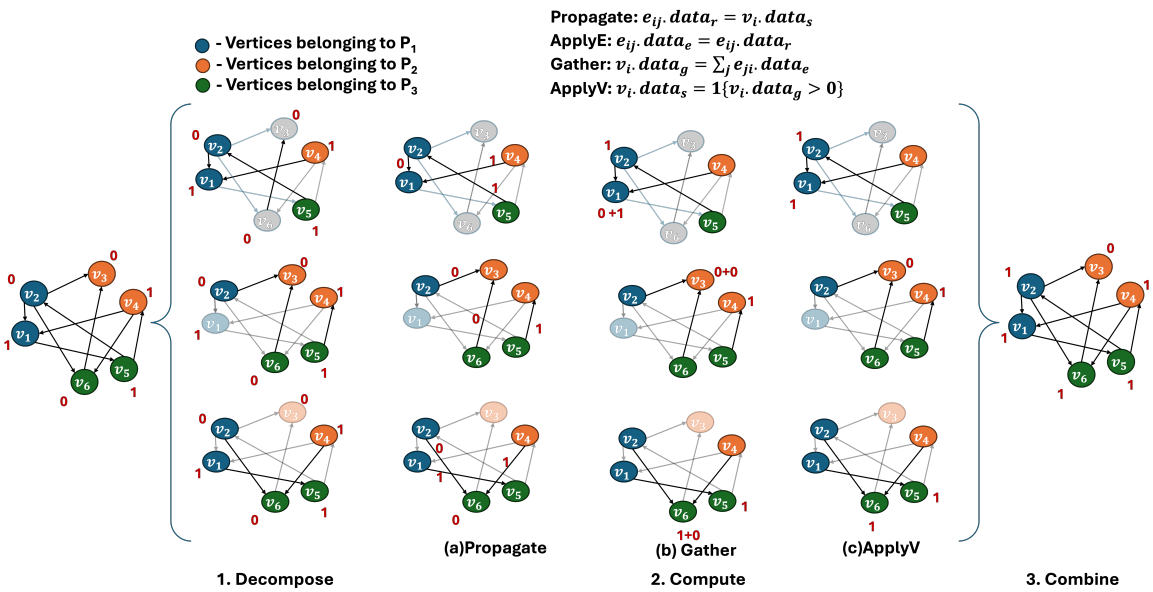


Fig. 2: Example of one iteration DCC approach for message-passing computations in cleartext.

DAG-list representation (G_i) for subgraphs $\mathcal{G}_i$, for $i \in [n]$, in the decompose step, (ii) performing message-passing computation on the DAG-lists G_i in the compute step, (iii) extracting the updated vertices V_i from G_i and updating the V for G in the combine step. Computing the same on a secret-shared input graph G via MPC enables secure graph computation.

Decompose: The decompose step takes as input the list representation of the global graph $\mathsf{G} = (\mathsf{V}, \{\mathsf{E}_i\}_{i\in[n]})$ and outputs a collection of DAG-lists $\{\mathsf{G}_i\}_{i\in[n]}$, where each G_i is a DAG-list (see II) representing the subgraph $\mathcal{G}_i = (\bar{\mathsf{V}}_i, \mathsf{E}_i)$. Here, E_i denotes all edges in G whose destination lies in V_i, and $\bar{\mathsf{V}}_i$ includes the vertices in V_i along with their immediate neighbours. Considering $\mathcal{G}_i$ to include only vertices in $\bar{\mathsf{V}}_i$ is not data-oblivious as it reveals information about the topology of G—in particular, the size $|\bar{\mathsf{V}}_i|$ in the view of only party P_i. Specifically, for any two subgraphs $\mathcal{G}_i$ of the same size, $|\bar{V}_i|$ need not be the same and hence directly using $\bar{V}_i$ is not data-oblivious. To avoid this, one naive approach is to include the entire vertex set V in each subgraph. Thus, the DAG-list G_i is simply set as V concatenated with E_i. This way, for any two subgraphs $\mathcal{G}_i$ of the same size, the generated DAG-list G_i will have the same size making it data-oblivious. Algorithm 2 gives the formal details of the data-oblivious algorithm for Decompose. Note, however, that this approach increases the computation and communication as additional vertices are processed in each subgraph. Looking ahead, message-passing computations on each secret-shared subgraph G_i would incur a communication cost of $O(n|\mathsf{G}_i|) = O(n(|\mathsf{V}|+|\mathsf{E}_i|))$, resulting in a total cost of $\sum_{i\in[n]} O(n(|\mathsf{V}|+|\mathsf{E}_i|)) = O(n^2|\mathsf{V}| + n|\mathsf{E}|)$, which scales quadratically with the number of parties and is thus prohibitive. We discuss in §IV-B how this can be improved.

Compute: This step takes as input DAG-lists of subgraph $\{\mathsf{G}_i\}_{i\in[n]}$ and outputs updated DAG-lists after one iteration of message-passing computations. At a high level, each subgraph G_i can be processed in parallel. Further, the message-passing computations on subgraphs DAG-list G_i can be realised data-obliviously by relying on the algorithm of [5] (see Algorithm 1) with G_i as input.

Algorithm 2: DCC-Decompose

Input: $\mathsf{G} = (\mathsf{V}, \{\mathsf{E}_i\})$ where each entry contains a tuple $(\mathsf{src}, \mathsf{dst}, \mathsf{isV}, \mathsf{data})$.
Output: DAG-lists $\{\mathsf{G}_i\}_{i\in[n]}$ in vertex order.

```
for i = 1 to n do in parallel
    G_i = V||E_i
```

Combine: This step takes as input DAG-lists G_i for $i \in [n]$ and outputs the updated graph representation $\mathsf{G} = (\mathsf{V}, \{\mathsf{E}_i\}_{i\in[n]})$, where $\mathsf{V} = \mathsf{V}_1||...||\mathsf{V}_n$, with V_i and E_i updated as per message-passing computations in G_i. Recall that, at the end of one iteration of message-passing computations in compute, the DAG-list G_i is already sorted in vertex order i.e $\mathsf{G}_i = \mathsf{V}||\mathsf{E}_i$. Since the labels of vertices in V as well as V_i are public information, they can be extracted from G_i. Specifically, we have, $\mathsf{V} = \mathsf{G}_1[1 : |\mathsf{V}_1|]|| \ldots ||\mathsf{G}_n[\sum_{i=1}^{n-1} |\mathsf{V}_i| + 1 : |\mathsf{V}_n|]$ and $\mathsf{E}_i = \mathsf{G}_i[|\mathsf{V}| + 1 :]$. Here, $\mathsf{G}[l : m]$ denotes the sublist of G containing entries from $\mathsf{G}[l]$ to $\mathsf{G}[m]$ and $\mathsf{G}[l :]$ denotes the sublist of G containing entries from $\mathsf{G}[l]$ to the last entry of G. Since all accesses in this phase depend only on public values ($|\mathsf{V}_i|$ and $|\mathsf{E}_i|$), the combine step is naturally data-oblivious.

Algorithm 3: DCC-Combine

Input: DAG-lists $\{\mathsf{G}_i = \mathsf{V}||\mathsf{E}_i\}_{i\in[n]}$ in vertex order
Output: $\mathsf{G} = (\mathsf{V}, \{\mathsf{E}_i\})$

```
V = []
for i = 1 to n do
    ind_i = 1 + Σ_{j=1}^{i-1} |V_j|,
    V_i = G_i[ind_i : ind_i + |V_i| − 1]
    E_i = G_i [|V| + 1 :]
G = (V_1| . . . |V_n, {E_i}_{i∈[n]})
```

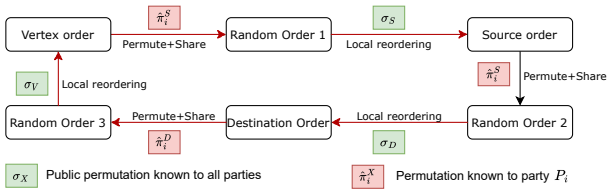


Fig. 3: Transitions involved in the 'compute' step for G_i.

IV. Secure Realisation of DCC paradigm

We begin by describing in §IV-A the secure realisation of the data-oblivious DCC algorithm described in §III-A. We then showcase how its communication complexity can be improved further in §IV-B.

A. First attempt

At the start of the computation, during an input sharing phase (described in §V), the parties generate secret shares of DAG-list $\mathsf{G} = (\mathsf{V}, \{\mathsf{E}_i\}_{i\in[n]})$, where each entry in these lists is additively secret-shared. Each iteration of message-passing computation begins with the secret-shared graph $\langle \mathsf{G} \rangle = (\langle \mathsf{V} \rangle, \{\langle \mathsf{E}_i \rangle\}_{i\in[n]})$ and outputs updated secret shares $\langle \mathsf{G} \rangle$, where the data components of the vertices and edges are updated. The steps involved in the secure realisation of a single iteration of our DCC approach are described next.

Decompose: This step takes as input the secret shares $\langle \mathsf{G} \rangle = (\langle \mathsf{V} \rangle, \{\langle \mathsf{E}_i \rangle\}_{i\in[n]})$, and outputs the secret-shares of n DAG-lists $\langle \mathsf{G}_i \rangle = (\langle \mathsf{V} \rangle, \langle \mathsf{E}_i \rangle)$, where each G_i is sorted in vertex order. Observe that since the input G is already secret shared as $(\langle \mathsf{V} \rangle, \{\langle \mathsf{E}_i \rangle\}_{i\in[n]})$, generating the DAG-lists G_i is non-interactive. Specifically, parties set $\langle \mathsf{G}_i \rangle$ as $\langle \mathsf{G}_i \rangle = \langle \mathsf{V} \rangle \,||\, \langle \mathsf{E}_i \rangle$, where $||$ denotes concatenation.

Compute: This step takes as input the secret-shared DAG-lists $\{\langle \mathsf{G}_i \rangle\}_{i\in[n]}$ in vertex order and performs one message-passing iteration on all of them in parallel. This involves secure realisation of Algorithm 1 on each DAG-list $\{\langle \mathsf{G}_i \rangle\}_{i\in[n]}$. For each G_i, the linear computations in secure realisation of Propagate and Gather are evaluated non-interactively, while ApplyE and ApplyV securely evaluate functions f_{AE} and f_{AV} respectively, entry-wise. The remaining operations in Algorithm 1 are the three transitions: from vertex to source, from source to destination, and from destination back to vertex. Realising each transition using a generic secure shuffle as in prior works [5] would incur $O(n)$ rounds per iteration. We instead exploit that party P_i knows the topology of G_i and can determine the required permutations for reordering. Specifically, consider the transition in G_i from vertex order to source order. Party P_i samples a uniformly random permutation $\hat{\pi^S}_i$ and sends $\sigma_i^S = \pi_i^S \circ (\hat{\pi_i^S})^{-1}$ to all parties during initialisation. Then to realise the transition from vertex order to source order, the parties first apply $\hat{\pi^S}_i$ to $\langle \mathsf{G}_i \rangle$ using Permute and Share (see §II) and then locally apply the public permutation σ_i^S. Since $\hat{\pi^S}_i$ is uniform, σ_i^S is uniformly distributed and reveals no information about the topology-dependent permutation π_i^S. Consequently, all three transitions are realised in $O(1)$ rounds. This is unlike prior generic frameworks, in which message-passing computations occur on the global graph whose topology is not known to any party. Thus, each transition there requires a one-time sort in the initialisation phase (to generate sharing of permutations that realise this transition), followed by a secure shuffle ($O(n)$ rounds) to apply the secret-shared permutation. An illustration of the transitions appears in Fig. 3.

Combine: This step takes as input secret shares of DAG-lists G_i for $i \in [n]$ and outputs secret-shares of V and $\{\mathsf{E}_i\}_{i\in[n]}$, where $\mathsf{V} = \mathsf{V}_1||...||\mathsf{V}_n$, with V_i and E_i updated as per message-passing computations in G_i. Recall that, at the end of one iteration of message-passing in compute, the DAG-list G_i is already sorted in vertex order. Note that labels of vertices in V as well as V_i is public information. Thus, the parties can non-interactively extract the shares of vertex entries $\mathsf{v} \in \mathsf{V}_i$ from shares of V in G_i. Following this, parties can non-interactively set $\langle \mathsf{V} \rangle = \langle \mathsf{V}_1 \rangle ||...|| \langle \mathsf{V}_n \rangle$. Similarly, the parties can extract shares of E_i from G_i non-interactively since these are the last $|\mathsf{E}|_i$ entries in G_i, where $|\mathsf{E}|_i$ is public information. Thus, this step does not require any additional rounds. Finally, the secret-shares of updated V and E_i together constitute the shares of G that is updated through one iteration of message-passing that serves as the input for the next iteration of DCC.

Formal details of the protocol described above appear in Fig. 12 (Appendix B).

Complexity of the solution. Each message-passing iteration consists of Decompose, Compute, and Combine. Decompose and Combine involve only local extraction and concatenation operations and therefore incur no communication or rounds. During Compute, the message-passing operations are performed on all G_is in parallel. For a DAG-list G_i of size $O(|\mathsf{V}|+|\mathsf{E}_i|)$, the three ordering transitions required by Propagate and Gather incur $O(1)$ rounds and $O(n(|\mathsf{V}|+|\mathsf{E}_i|))$ communication. ApplyE incurs $O(\mathsf{r}_{\mathsf{AE}})$ rounds and $O((|\mathsf{V}|+|\mathsf{E}_i|)\mathsf{c}_{\mathsf{AE}})$ communication, while ApplyV incurs $O(\mathsf{r}_{\mathsf{AV}})$ rounds and $O(|\mathsf{V}|\mathsf{c}_{\mathsf{AV}})$ communication. Here, r_{AE} and r_{AV} denote the round complexities of securely evaluating f_{AE} and f_{AV}, respectively, while c_{AE} and c_{AV} denote their per-entry communication costs. Hence, Compute on a single G_i incurs $O(1+\mathsf{r}_{\mathsf{AE}}+\mathsf{r}_{\mathsf{AV}})$ rounds and $O(n(|\mathsf{V}|+|\mathsf{E}_i|)+(|\mathsf{V}|+|\mathsf{E}_i|)\mathsf{c}_{\mathsf{AE}}+|\mathsf{V}|\mathsf{c}_{\mathsf{AV}})$ communication. Since all G_is are processed in parallel, the round complexity remains $O(1+\mathsf{r}_{\mathsf{AE}}+\mathsf{r}_{\mathsf{AV}})$, whereas summing the communication over all $i \in [n]$ gives $O(n^2|\mathsf{V}| + n|\mathsf{E}| + \mathsf{c}_{\mathsf{AE}}(n|\mathsf{V}| + |\mathsf{E}|) + \mathsf{c}_{\mathsf{AV}}n|\mathsf{V}|)$. Therefore, one DCC iteration incurs $O(1+\mathsf{r}_{\mathsf{AE}}+\mathsf{r}_{\mathsf{AV}})$ rounds and $O(n^2|\mathsf{V}| + n|\mathsf{E}| + \mathsf{c}_{\mathsf{AE}}(n|\mathsf{V}| + |\mathsf{E}|) + \mathsf{c}_{\mathsf{AV}}n|\mathsf{V}|)$ communication. Thus, the round complexity is independent of n, while the communication complexity is quadratic in n due to replicating the complete vertex set in every G_i.

B. Achieving linear communication

We now describe how to reduce the communication complexity to be linear in the number of parties n. The main bottleneck in the first attempt arises from the transitions required during Propagate and Gather. For each DAG-list $\mathsf{G}_i = \mathsf{V}||\mathsf{E}_i$, these transitions incur $O(n(|\mathsf{V}|+|\mathsf{E}_i|))$ communication. Since there are n such DAG-lists, the total communication is $O(n^2|\mathsf{V}|+n|\mathsf{E}|)$, which is quadratic in n due to the replication of the entire vertex set in every G_i.

Recall from §III that, to update the vertices in V_i, it suffices for G_i to contain all vertices in V_i together with the source

Algorithm 4: DCC-Decompose (Improved)

Input: $\mathsf{G} = (\mathsf{V}, \{\mathsf{E}_i\}_{i\in[n]})$, where each entry contains $(\mathsf{src}, \mathsf{dst}, \mathsf{isV}, \mathsf{data})$.

Output: DAG-lists $\{\mathsf{G}_i\}_{i\in[n]}$ in vertex order.

1 **for** $i = 1$ **to** n **do in parallel**

2 $\quad B_i \leftarrow \min(|\mathsf{V}|, |\mathsf{V}_i| + |\mathsf{E}_i|)$;

3 $\quad \mathsf{V}'_i \leftarrow \pi_i^G(\mathsf{V})$, where π_i^G places the vertices in V_i first, followed by those in $\bar{\mathsf{V}}_i \setminus \mathsf{V}_i$ and the remaining vertices;

4 $\quad \widetilde{\mathsf{V}}_i \leftarrow \mathsf{V}'_i[1 : B_i]$;

5 $\quad \mathsf{G}_i \leftarrow \widetilde{\mathsf{V}}_i \| \mathsf{E}_i$;

endpoints of the edges in E_i. We denote this vertex set by $\bar{\mathsf{V}}_i = \mathsf{V}_i \cup \{u \in \mathsf{V} : (u, v) \in \mathsf{E}_i \text{ for some } v \in \mathsf{V}_i\}$. The exact value of $|\bar{\mathsf{V}}_i|$ depends on the private topology of G_i and must therefore remain hidden. Towards this, we observe that the size of $\bar{\mathsf{V}}_i$ is publicly bounded. Specifically, since every edge in E_i has its destination in V_i and can introduce at most one additional source vertex outside V_i, we have $|\bar{\mathsf{V}}_i| \le \min(|\mathsf{V}|, |\mathsf{V}_i| + |\mathsf{E}_i|)$. We define the public padding bound as $B_i = \min(|\mathsf{V}|, |\mathsf{V}_i| + |\mathsf{E}_i|)$. The value B_i depends only on the public parameters $|\mathsf{V}|$, $|\mathsf{V}_i|$, and $|\mathsf{E}_i|$. We pad $\bar{\mathsf{V}}_i$, if necessary, with vertices from $\mathsf{V} \setminus \bar{\mathsf{V}}_i$ to obtain a padded vertex list $\widetilde{\mathsf{V}}_i$ containing exactly B_i entries. The resulting padded DAG-list is $\mathsf{G}_i = \widetilde{\mathsf{V}}_i \| \mathsf{E}_i$ and has size $|\mathsf{G}_i| = B_i + |\mathsf{E}_i| \le |\mathsf{V}_i| + 2|\mathsf{E}_i|$. Consequently, the aggregate size of all the padded subgraphs satisfies $\sum_{i=1}^n |\mathsf{G}_i| \le \sum_{i=1}^n (|\mathsf{V}_i| + 2|\mathsf{E}_i|) = |\mathsf{V}| + 2|\mathsf{E}| = O(|\mathsf{V}| + |\mathsf{E}|)$. Thus, although the graph is decomposed into n subgraphs, their aggregate size remains linear in the size of the global graph.

However, this optimisation introduces a new challenge: generating each padded vertex list $\widetilde{\mathsf{V}}_i$ data-obliviously during the 'Decompose' step. In particular, the parties must collect all vertices in V_i and their neighbours without revealing which vertices are actual neighbours and which vertices are used only for padding. To address this, we propose a new data-oblivious approach to realise this in Algorithm 4. Specifically, party P_i defines a permutation π_i^G that reorders V such that the vertices in V_i appear first, followed by the remaining vertices in $\bar{\mathsf{V}}_i \setminus \mathsf{V}_i$ and finally the remaining vertices. The first B_i entries of the permuted list then form $\widetilde{\mathsf{V}}_i$. Placing V_i first is necessary so that the updated states of the vertices owned by P_i can be extracted directly during the 'Combine' step.

In the secure realisation of Algorithm 4, the permutation π_i^G is known only to party P_i. Given additive shares $\langle \mathsf{V} \rangle$, the parties can obtain shares of $\pi_i^G(\mathsf{V})$ using one invocation of $\mathcal{F}_{\mathsf{PS}}$. However, independently generating all n permuted lists would require n parallel invocations of $\mathcal{F}_{\mathsf{PS}}$. Since one invocation on a list of size $|\mathsf{V}|$ incurs $O(n|\mathsf{V}|)$ communication, the naive approach incurs $O(n^2|\mathsf{V}|)$ communication.

To avoid this quadratic overhead, we introduce the functionality $\mathcal{F}_{\mathsf{APS}}$, which generates multiple permuted versions of the same secret-shared input using the communication cost of a single $\mathcal{F}_{\mathsf{PS}}$ invocation. Specifically, $\mathcal{F}_{\mathsf{APS}}$ takes as input a secret-shared list $\langle \mathsf{T} \rangle$ and a permutation π_i from every party P_i, and outputs secret shares of n lists $\{\mathsf{T}_i\}_{i\in[n]}$, where $\mathsf{T}_i = \pi_i(\mathsf{T})$. We design a novel MPC protocol Π_{APS} for securely realising $\mathcal{F}_{\mathsf{APS}}$. The formal details of this protocol appears ahead in §VI. Assuming $\mathcal{F}_{\mathsf{APS}}$, we next detail the secure realisation of the modified decompose step.

DAG-list generation during the Decompose step. Each party P_i samples a uniformly random permutation $\hat{\pi}_i^G$. It then computes the public permutation $\sigma_i^G = \pi_i^G \circ (\hat{\pi}_i^G)^{-1}$ and sends σ_i^G to the other parties. The permutation σ_i^G maps the randomly ordered list $\hat{\pi}_i^G(\mathsf{V})$ to the ordering defined by π_i^G.

During the 'Decompose' step, the parties invoke Amortised Permute and Share on $\langle \mathsf{V} \rangle$ using the random permutations $\{\hat{\pi}_i^G\}_{i\in[n]}$. This produces secret shares of the randomly permuted lists $\{\hat{\pi}_i^G(\mathsf{V})\}_{i\in[n]}$. The parties then locally apply the corresponding public permutation σ_i^G to obtain $\langle \mathsf{V}'_i \rangle = \langle \pi_i^G(\mathsf{V}) \rangle$. They extract the first B_i entries to obtain $\langle \widetilde{\mathsf{V}}_i \rangle = \langle \mathsf{V}'_i[1 : B_i] \rangle$ and set $\langle \mathsf{G}_i \rangle = \langle \widetilde{\mathsf{V}}_i \rangle \| \langle \mathsf{E}_i \rangle$.

The mappings π_i^G depend only on the fixed graph topology and remain unchanged across message-passing iterations. Hence, the same random permutations $\hat{\pi}_i^G$ and public permutations σ_i^G can be reused across all iterations. The public permutations are therefore computed and communicated once during the initialisation phase.

V. GraSP Framework

In this section, we formally describe the end-to-end protocol of GraSP for securely realising a message-passing graph algorithm. The ideal functionality $\mathcal{F}_{\mathsf{GraSP}}$ appears in Fig. 4, and the corresponding protocol Π_{GraSP} appears in Fig. 5. Party P_i inputs its local vertices V_i and the edges E_i whose destinations belong to V_i. All parties agree on the message-passing algorithm alg, including the number of iterations k, the propagated and aggregated data, and the apply functions f_{AE} and f_{AV}. For simplicity, we assume that party P_i receives the updated state of its vertices V_i however, the reconstruction phase can be adapted to deliver the outputs to any designated set of recipients. The protocol consists of three phases: (1) input sharing and initialisation, (2) message-passing computation via DCC, and (3) reconstruction.

Functionality $\mathcal{F}_{\mathsf{GraSP}}$

Let $\mathcal{C} \subset \mathcal{P}$ denote the set of parties corrupted by the ideal-world adversary $\mathcal{S}$. The functionality $\mathcal{F}_{\mathsf{GraSP}}$ interacts with the parties in $\mathcal{P}$ and $\mathcal{S}$.

It receives $(\mathsf{V}_i, \mathsf{E}_i)$ from party P_i for every $i \in [n]$. All parties additionally provide the same description of the graph algorithm alg, which specifies the number of iterations k, the data propagated and aggregated, the aggregation operation, and the functions f_{AE} and f_{AV}. The functionality proceeds as follows.

- Construct $\mathsf{G} = (\mathsf{V}, \mathsf{E})$, where $\mathsf{V} = \mathsf{V}_1 \| \cdots \| \mathsf{V}_n$ and $\mathsf{E} = \mathsf{E}_1 \| \cdots \| \mathsf{E}_n$.
- Perform k iterations of the message-passing computation specified by alg on G.
- Decompose the updated vertex list as $\mathsf{V} = \mathsf{V}_1 \| \cdots \| \mathsf{V}_n$.
- Send $(\mathsf{Output}, \mathsf{V}_i, \{|\mathsf{V}_j|, |\mathsf{E}_j|\}_{j\in[n]})$ to P_i for every $i \in [n]$.

Fig. 4: Ideal functionality for secure graph processing.

Input sharing. Each party P_i generates additive shares of V_i and E_i using the shared-key setup described in §II. The parties then set $\langle \mathsf{V} \rangle = \langle \mathsf{V}_1 \rangle \| \cdots \| \langle \mathsf{V}_n \rangle$ and $\langle \mathsf{G} \rangle = (\langle \mathsf{V} \rangle, \{\langle \mathsf{E}_i \rangle\}_{i\in[n]})$. Given the shared-key setup, the shares of a party's input can be

generated without online interaction; hence, this phase incurs no online communication or rounds.

Initialisation. The parties generate the public permutations used repeatedly during the 'decompose' and 'compute' steps. For each $i \in [n]$, define the public padding bound $B_i = \min(|\mathsf{V}|, |\mathsf{V}_i| + |\mathsf{E}_i|)$.

- Party P_i defines a permutation π_i^G of V that places the vertices in V_i first, followed by the vertices in $\bar{\mathsf{V}}_i \setminus \mathsf{V}_i$ and finally the remaining vertices. Thus, the first B_i entries of $\pi_i^G(\mathsf{V})$ form the padded vertex list required for G_i, while its first $|\mathsf{V}_i|$ entries are exactly the vertices owned by P_i. Party P_i samples a uniformly random permutation $\hat{\pi}_i^G$ over V and computes the public permutation $\sigma_i^G = \pi_i^G \circ (\hat{\pi}_i^G)^{-1}$. Consequently, securely applying $\hat{\pi}_i^G$ followed by locally applying σ_i^G produces the ordering specified by π_i^G without revealing π_i^G.
- Let $\widetilde{\mathsf{V}}_i$ denote the padded vertex list of size B_i, and let $\mathsf{G}_i = \widetilde{\mathsf{V}}_i \| \mathsf{E}_i$. Using its knowledge of the topology of G_i, party P_i defines permutations π_i^S, π_i^D, and π_i^V that map vertex order to source order, source order to destination order, and destination order back to vertex order, respectively.

Party P_i samples independent random permutations $\hat{\pi}_i^S$, $\hat{\pi}_i^D$, and $\hat{\pi}_i^V$ and publishes $\sigma_i^S = \pi_i^S \circ (\hat{\pi}_i^S)^{-1}$, $\sigma_i^D = \pi_i^D \circ (\hat{\pi}_i^D)^{-1}$, and $\sigma_i^V = \pi_i^V \circ (\hat{\pi}_i^V)^{-1}$.

For simplicity, the protocol is described using the complete permutation σ_i^G over $|\mathsf{V}|$ entries. In the concrete implementation, however, only its first B_i output positions are required during Decompose; hence, only the corresponding B_i entries of σ_i^G are communicated. The permutations σ_i^S, σ_i^D, and σ_i^V are each communicated in full and have size $L_i = B_i + |\mathsf{E}_i|$. Therefore, the total number of permutation entries communicated by all parties is $\sum_{i\in[n]}(B_i + 3L_i) = O(N)$, since $\sum_i B_i \le |\mathsf{V}| + |\mathsf{E}|$ and $\sum_i L_i \le |\mathsf{V}| + 2|\mathsf{E}|$. As every entry is sent to all n parties, the initialisation communication is $O(nN)$.

Message-passing computations via DCC. The parties perform k DCC iterations. Each iteration securely realises 'decompose' from Algorithm 4, 'compute' from Algorithm 1, and 'combine' from Algorithm 3. During 'decompose', the parties invoke $\mathcal{F}_{\mathsf{APS}}$ once on $\langle \mathsf{V} \rangle$ using $\{\hat{\pi}_i^G\}_{i\in[n]}$. For every i, they locally apply σ_i^G, retain the first B_i entries to obtain $\left\langle \widetilde{\mathsf{V}}_i \right\rangle$, and set $\langle \mathsf{G}_i \rangle = \left\langle \widetilde{\mathsf{V}}_i \right\rangle \| \langle \mathsf{E}_i \rangle$. During 'compute', message-passing computations on all subgraphs G_i are performed in parallel. Elaborately, steps corresponding to linear operations (Steps 1–5, 7–10, 14–17, and 19–21) are realised non-interactively by leveraging the linearity of $\langle\cdot\rangle$-sharing. Each of the three ordering transitions invokes $\mathcal{F}_{\mathsf{PS}}$ using the random permutation held by P_i, followed by local application of the corresponding public permutation. The entry-wise computations in ApplyE and ApplyV are securely evaluated using $\mathcal{F}_{\mathsf{MPC}}$ and $\mathcal{F}_{\mathsf{mult}}$. During 'combine', the parties extract the first $|\mathsf{V}_i|$ entries of G_i as the updated V_i. They also retain the final $|\mathsf{E}_i|$ entries as E_i. These operations use only public indices and are therefore non-interactive. The parties then set $\langle \mathsf{V} \rangle = \langle \mathsf{V}_1 \rangle \| \cdots \| \langle \mathsf{V}_n \rangle$ and $\langle \mathsf{G} \rangle = (\langle \mathsf{V} \rangle, \{\langle \mathsf{E}_i \rangle\}_{i\in[n]})$ for the next iteration.

Reconstruction. After completing the required iterations, the parties reconstruct $\langle \mathsf{V}_i \rangle$ towards P_i for every $i \in [n]$. Depending on the application, the outputs may instead be reconstructed towards a designated subset of parties or a trusted aggregator.

Protocol Π_{GraSP}

Inputs: Party P_i holds $(\mathsf{V}_i, \mathsf{E}_i)$ for $i \in [n]$. All parties agree on alg, including the number of iterations k, the propagated and aggregated data, the aggregation operation, and the functions f_{AE} and f_{AV}.
Outputs: Party P_i receives V_i with its data components updated according to alg.
Setup: The parties invoke $\mathcal{F}_{\mathsf{setup}}$ for key distribution.

Input Sharing and Initialisation: Each party P_i performs the following: Generate additive shares of V_i and E_i using the shared-key setup. Set $B_i = \min(|\mathsf{V}|, |\mathsf{V}_i| + |\mathsf{E}_i|)$. Define π_i^G so that $\pi_i^G(\mathsf{V})$ consists first of the vertices in V_i, then the vertices in $\bar{\mathsf{V}}_i \setminus \mathsf{V}_i$, then sufficient padding vertices, and finally the remaining vertices. Define π_i^S, π_i^D, and π_i^V for the padded list G_i, mapping vertex order to source order, source order to destination order, and destination order to vertex order. Sample random permutations $\hat{\pi}_i^G$, $\hat{\pi}_i^S$, $\hat{\pi}_i^D$, and $\hat{\pi}_i^V$, and send $\sigma_i^G = \pi_i^G \circ (\hat{\pi}_i^G)^{-1}$, $\sigma_i^S = \pi_i^S \circ (\hat{\pi}_i^S)^{-1}$, $\sigma_i^D = \pi_i^D \circ (\hat{\pi}_i^D)^{-1}$, and $\sigma_i^V = \pi_i^V \circ (\hat{\pi}_i^V)^{-1}$ to other parties.
The parties set $\langle \mathsf{V} \rangle = \langle \mathsf{V}_1 \rangle \| \cdots \| \langle \mathsf{V}_n \rangle$ and $\langle \mathsf{G} \rangle = (\langle \mathsf{V} \rangle, \{\langle \mathsf{E}_i \rangle\}_{i\in[n]})$.
DCC message-passing computation: For $r = 1, \ldots, k$, perform the following.

- **Decompose.** Compute $\{\langle \mathsf{V}_i' \rangle\}_{i\in[n]} \leftarrow \mathcal{F}_{\mathsf{APS}}(\langle \mathsf{V} \rangle, \{\hat{\pi}_i^G : P_i\}_{i\in[n]})$ and locally apply σ_i^G to $\langle \mathsf{V}_i' \rangle$ for $i \in [n]$.
  Set $\left\langle \widetilde{\mathsf{V}}_i \right\rangle = \langle \mathsf{V}_i' \rangle [1 : B_i]$ and $\langle \mathsf{G}_i \rangle = \left\langle \widetilde{\mathsf{V}}_i \right\rangle \| \langle \mathsf{E}_i \rangle$.

Compute. For every $i \in [n]$ in parallel, perform the following.
– **Propagate in vertex order:** Set $\langle \mathsf{G}_i[1].\mathsf{data_r} \rangle = \langle \mathsf{G}_i[1].\mathsf{data_s} \rangle$. For $q = 2, \ldots, B_i$, set $\langle \mathsf{G}_i[q].\mathsf{data_r} \rangle = \langle \mathsf{G}_i[q].\mathsf{data_s} \rangle - \langle \mathsf{G}_i[q-1].\mathsf{data_s} \rangle$. For $q = B_i + 1, \ldots, |\mathsf{G}_i|$, set $\langle \mathsf{G}_i[q].\mathsf{data_r} \rangle = 0$.
– Invoke $\mathcal{F}_{\mathsf{PS}}(\langle \mathsf{G}_i \rangle, \{P_i : \hat{\pi}_i^S\})$ and locally apply σ_i^S to obtain G_i in source order.
– For $q = 1, \ldots, |\mathsf{G}_i|$, set $\langle \mathsf{G}_i[q].\mathsf{data_r} \rangle = \sum_{h=1}^{q} \langle \mathsf{G}_i[h].\mathsf{data_r} \rangle - \langle \mathsf{G}_i[q].\mathsf{data_s} \rangle$.
– **ApplyE:** for every $q = 1, \ldots, |\mathsf{G}_i|$, compute $\langle z_q \rangle \leftarrow \mathcal{F}_{\mathsf{MPC}}(\langle \mathsf{G}_i[q].\mathsf{data_e} \rangle, \langle \mathsf{G}_i[q].\mathsf{data_r} \rangle, f_{\mathsf{AE}})$ and then $\langle \mathsf{G}_i[q].\mathsf{data_e} \rangle \leftarrow \mathcal{F}_{\mathsf{mult}}(\langle z_q \rangle, 1 - \langle \mathsf{G}_i[q].\mathsf{isV} \rangle)$.
– Invoke $\mathcal{F}_{\mathsf{PS}}(\langle \mathsf{G}_i \rangle, \{P_i : \hat{\pi}_i^D\})$ and locally apply σ_i^D to obtain destination order.
– **Gather in destination order:** for $q = 1, \ldots, |\mathsf{G}_i|$, set $\langle \mathsf{G}_i[q].\mathsf{data_g}' \rangle = \sum_{h=1}^{q} \langle \mathsf{G}_i[h].\mathsf{data_e} \rangle$.
– Invoke $\mathcal{F}_{\mathsf{PS}}(\langle \mathsf{G}_i \rangle, \{P_i : \hat{\pi}_i^V\})$ and locally apply σ_i^V to obtain vertex order.
– Set $\langle \mathsf{G}_i[1].\mathsf{data_g} \rangle = \langle \mathsf{G}_i[1].\mathsf{data_g}' \rangle$ and, for $q = 2, \ldots, B_i$, set $\langle \mathsf{G}_i[q].\mathsf{data_g} \rangle = \langle \mathsf{G}_i[q].\mathsf{data_g}' \rangle - \langle \mathsf{G}_i[q-1].\mathsf{data_g}' \rangle$.
– **ApplyV:** for $q = 1, \ldots, |\mathsf{V}_i|$, compute $\langle \mathsf{G}_i[q].\mathsf{data_v} \rangle \leftarrow \mathcal{F}_{\mathsf{MPC}}(\langle \mathsf{G}_i[q].\mathsf{data_v} \rangle, \langle \mathsf{G}_i[q].\mathsf{data_g} \rangle, f_{\mathsf{AV}})$.

Combine. For every $i \in [n]$, set $\langle \mathsf{V}_i \rangle = \langle \mathsf{G}_i \rangle [1 : |\mathsf{V}_i|]$ and $\langle \mathsf{E}_i \rangle = \langle \mathsf{G}_i \rangle [B_i + 1 : B_i + |\mathsf{E}_i|]$. Then set $\langle \mathsf{V} \rangle = \langle \mathsf{V}_1 \rangle \| \cdots \| \langle \mathsf{V}_n \rangle$ and $\langle \mathsf{G} \rangle = (\langle \mathsf{V} \rangle, \{\langle \mathsf{E}_i \rangle\}_{i\in[n]})$.
Reconstruction: For every $i \in [n]$, the parties reconstruct $\langle \mathsf{V}_i \rangle$ towards P_i.

Fig. 5: GraSP protocol for secure collaborative graph processing.

Complexity of GraSP. The input-sharing phase incurs no online rounds or communication. Initialisation requires one round and $O(n\mathsf{N})$ communication for distributing the public permutations. In each DCC iteration, 'decompose' invokes

$\mathcal{F}_{\mathsf{APS}}$ once on a list of size $|\mathsf{V}|$, incurring two rounds and $O(n|\mathsf{V}|)$ communication. For subgraph G_i, the 'compute' step invokes $\mathcal{F}_{\mathsf{PS}}$ three times on a list of size $B_i + |\mathsf{E}_i|$. Since $\sum_{i\in[n]}(B_i + |\mathsf{E}_i|) = O(\mathsf{N})$, the three transitions incur three rounds and $O(n\mathsf{N})$ total communication. Let r_{AE} and r_{AV} denote the round complexities of securely evaluating f_{AE} and f_{AV}, and let c_{AE} and c_{AV} denote their per-entry communication costs. Since $\sum_i |\mathsf{G}_i| = O(\mathsf{N})$ and $\sum_i |\mathsf{V}_i| = |\mathsf{V}|$, the application functions incur $O(\mathsf{r}_{\mathsf{AE}} + \mathsf{r}_{\mathsf{AV}})$ rounds and $O((\mathsf{c}_{\mathsf{AE}} + \mathsf{c}_{\mathsf{AV}})\mathsf{N})$ communication per iteration. Thus, one DCC iteration incurs $5 + O(\mathsf{r}_{\mathsf{AE}} + \mathsf{r}_{\mathsf{AV}})$ rounds and $O((n + \mathsf{c}_{\mathsf{AE}} + \mathsf{c}_{\mathsf{AV}})\mathsf{N})$ communication. Reconstruction requires one round and $O(n|\mathsf{V}|)$ communication. Therefore, the complete online protocol incurs $2+k(5+O(\mathsf{r}_{\mathsf{AE}}+\mathsf{r}_{\mathsf{AV}}))$ rounds and $O(n\mathsf{N}+k(n+\mathsf{c}_{\mathsf{AE}}+\mathsf{c}_{\mathsf{AV}})\mathsf{N}+n|\mathsf{V}|)$ communication. Equivalently, suppressing additive one-time costs, the complexities are $O(k(1 + \mathsf{r}_{\mathsf{AE}} + \mathsf{r}_{\mathsf{AV}}))$ rounds and $O(k(n + \mathsf{c}_{\mathsf{AE}} + \mathsf{c}_{\mathsf{AV}})\mathsf{N})$ communication. In particular, the round complexity is independent of both the graph size and the number of parties.

Security of GraSP. We prove security using the standard real-world/ideal-world simulation paradigm [17]. Let $\mathcal{A}$ denote the real-world adversary corrupting at most $n-1$ parties and $\mathcal{S}$ denote the ideal-world adversary. We analyse Π_{GraSP} in the $(\mathcal{F}_{\mathsf{setup}}, \mathcal{F}_{\mathsf{MPC}}, \mathcal{F}_{\mathsf{mult}}, \mathcal{F}_{\mathsf{APS}}, \mathcal{F}_{\mathsf{PS}})$-hybrid model, where the sub-protocol Π_X is replaced by a call to its ideal functionality $\mathcal{F}_X$ [18]. Since secure protocols for these appear in the literature, following standard practice, we replace them with a call to their ideal functionalities, where the latter are known to be secure by definition [4].

Lemma V.1: Protocol Π_{GraSP} securely realises $\mathcal{F}_{\mathsf{GraSP}}$ against a static, semi-honest adversary corrupting at most $n-1$ parties in the $(\mathcal{F}_{\mathsf{setup}}, \mathcal{F}_{\mathsf{MPC}}, \mathcal{F}_{\mathsf{mult}}, \mathcal{F}_{\mathsf{APS}}, \mathcal{F}_{\mathsf{PS}})$-hybrid model.

Proof. Without loss of generality, assume the adversary $\mathcal{A}$ corrupts parties $P_1, \ldots, P_t$ for some $t < n$. We now construct a simulator $\mathcal{S}_{\Pi_{\mathsf{GraSP}}}$ that generates a view for $\mathcal{A}$ that is indistinguishable from its real-world view, given only the inputs and outputs of the corrupted parties. The full description of $\mathcal{S}_{\Pi_{\mathsf{GraSP}}}$ is given in Fig. 15. At a high level, $\mathcal{S}_{\Pi_{\mathsf{GraSP}}}$ proceeds as follows. The simulator first emulates $\mathcal{F}_{\mathsf{setup}}$ and obtains the setup information delivered to the corrupted parties. For every honest party, it samples the public permutations $\sigma_i^G, \sigma_i^S, \sigma_i^D, \sigma_i^V$ according to their real-world distribution. Since each is the composition of a fixed permutation with the inverse of an independently uniform permutation, every such public permutation is itself uniformly distributed over the relevant permutation domain. During the DCC iterations, the simulator forwards the corrupted parties' inputs to the corresponding ideal functionalities and simulates the honest parties' by emulating $\mathcal{F}_{\mathsf{APS}}$, $\mathcal{F}_{\mathsf{PS}}$, $\mathcal{F}_{\mathsf{MPC}}$, and $\mathcal{F}_{\mathsf{mult}}$ on random shares. Finally, during reconstruction towards a corrupted party P_i, the simulator chooses random shares on behalf of all but one honest party and fixes the remaining honest share so that the shares reconstruct to the output V_i supplied by $\mathcal{F}_{\mathsf{GraSP}}$. Reconstruction towards honest parties reveals no message to the adversary beyond the corrupted parties' outgoing shares. The simulated view is therefore computationally indistinguishable from the adversary's view in the real protocol. □

[4]the security of Amortised Permute and Share appears in Appendix F.

VI. Amortised Permute+Share

This protocol enables the generation of multiple permuted lists from a single secret-shared input list. It takes as input a secret-shared list $\langle \mathsf{T} \rangle$ and a permutation π_i from each party P_i, and outputs n secret-shared lists $\{\langle \mathsf{T}_j \rangle\}_{j\in[n]}$ such that $\mathsf{T}_j = \pi_j(\mathsf{T})$ for every $j \in [n]$.

Our construction builds on the Permute and Share protocol of [7], [14], which securely applies a permutation known to one designated party. Specifically, Permute and Share takes as input a secret-shared list $\langle \mathsf{T} \rangle$ and a permutation π_i known to party P_i, and outputs shares $\langle \mathsf{T}_\mathsf{O} \rangle$ such that $\mathsf{T}_\mathsf{O} = \pi_i(\mathsf{T})$. Its preprocessing phase generates a shuffle correlation consisting of shares $\langle \mathsf{R} \rangle$ and $\langle \pi_i(\mathsf{R}) \rangle$, where R is a uniformly random mask. In the online phase, the parties reconstruct the masked list $\mathsf{T} + \mathsf{R}$ towards P_i. Party P_i then sets its output share to $\langle \mathsf{T}_\mathsf{O} \rangle_i = \pi_i(\mathsf{T} + \mathsf{R}) - \langle \pi_i(\mathsf{R}) \rangle_i$, while every other party P_j, for $j \neq i$, sets $\langle \mathsf{T}_\mathsf{O} \rangle_j = -\langle \pi_i(\mathsf{R}) \rangle_j$. Consequently, the output shares reconstruct to $\pi_i(\mathsf{T} + \mathsf{R}) - \pi_i(\mathsf{R}) = \pi_i(\mathsf{T})$.

A straightforward realisation of Amortised Permute and Share would execute n parallel instances of Permute and Share, with party P_i supplying permutation π_i in the i-th instance. This requires n independent shuffle correlations $\{(\langle \mathsf{R}_i \rangle, \langle \pi_i(\mathsf{R}_i) \rangle)\}_{i\in[n]}$ during preprocessing and the reconstruction of n masked lists $\{\mathsf{T} + \mathsf{R}_i\}_{i\in[n]}$ during the online phase. Since each reconstruction incurs $O(n\mathsf{N})$ communication, the total online communication becomes $O(n^2\mathsf{N})$. To reduce this cost, Amortised Permute and Share uses a single uniformly random mask R, unknown to every individual party, across all n output permutations within one protocol invocation. During preprocessing, the parties generate shares $\langle \mathsf{R} \rangle$ and, for every $j \in [n]$, shares $\langle \pi_j(\mathsf{R}) \rangle$[5]. In the online phase, they reconstruct the single masked list $\mathsf{T} + \mathsf{R}$ to all parties. This reconstruction is realised using the P_{king} approach [19] where every party sends its share of $\mathsf{T} + \mathsf{R}$ to a designated king, which reconstructs the masked list and broadcasts it to the other parties. This requires two rounds and $O(n\mathsf{N})$ communication.

After receiving $\mathsf{T} + \mathsf{R}$, the parties locally generate shares of every output list $\mathsf{T}_j = \pi_j(\mathsf{T})$. For each $j \in [n]$, party P_j, which knows π_j, sets its distinguished output share as $\langle \mathsf{T}_j \rangle_j = \pi_j(\mathsf{T} + \mathsf{R}) - \langle \pi_j(\mathsf{R}) \rangle_j$. Every other party P_i, for $i \neq j$, sets $\langle \mathsf{T}_j \rangle_i = -\langle \pi_j(\mathsf{R}) \rangle_i$. The resulting shares satisfy $\sum_{i=1}^n \langle \mathsf{T}_j \rangle_i = \pi_j(\mathsf{T} + \mathsf{R}) - \sum_{i=1}^n \langle \pi_j(\mathsf{R}) \rangle_i = \pi_j(\mathsf{T})$ for every $j \in [n]$. Thus, Amortised Permute and Share generates all n permuted outputs using a single masked-list reconstruction, achieving $O(1)$ rounds and $O(n\mathsf{N})$ online communication. The formal protocol (Fig. 14) and its security proof appears in Appendix F.

VII. Benchmarks

Implementation. We implement GraSP in C++ by extending the codebase of [5], with all protocols instantiated using the

[5]Note that A fresh random mask R and fresh Share Translation correlations are generated independently for every invocation of Π_{APS}.

generic semi-honest MPC protocol of [20]. Specifically, our implementation relies on additive secret sharing together with Beaver-triple-based secure multiplication [21]. We instantiate the Permute and Share using the protocol of [14], while equality testing is implemented using the protocol of [22]. To demonstrate the efficiency of GraSP, we compare the cost of securely computing PageRank against two state-of-the-art systems. First, we compare with Graphiti [5], instantiated in the non-outsourced setting using the same underlying MPC protocols as GraSP, thereby isolating the algorithmic improvements introduced by our framework. Second, we compare against RingSG using its publicly available implementation[6] under the same benchmarking environment to ensure a like-for-like comparison. Our implementation is intended solely for benchmarking and is publicly available.[7]

Benchmark Environment. Experiments are conducted on Ubuntu servers equipped with AMD Ryzen Threadripper PRO 5965WX processors and 512 GB RAM. Each party is executed as a separate process and utilises up to four worker threads. We evaluate PageRank on randomly generated scale-free graphs, where vertices are distributed uniformly across the participating parties; that is, each party owns approximately $|\mathsf{V}|/n$ vertices together with their incident edges. Since the performance of both GraSP and prior secure graph-processing frameworks depends primarily on graph size rather than topology, all experiments use randomly generated graphs. Unless stated otherwise, computations are performed over the ring $\mathbb{Z}_{2^{64}}$. We focus on the online performance throughout the paper, while preprocessing benchmarks are deferred to Appendix D.

PageRank Computation. We benchmark the PageRank (PR) algorithm, implemented as an iterative message-passing computation. Each vertex stores its current PR value together with the precomputed reciprocal of its out-degree. In every iteration, Propagate sends the current PR of each vertex to its outgoing edges, Gather accumulates incoming messages at each vertex, and ApplyV updates the PR according to $PR(\mathsf{u}) = \frac{1-\alpha}{|\mathsf{V}|} + \alpha \sum_{\mathsf{e}(\mathsf{v},\mathsf{u})\in\mathsf{E}} \frac{PR(\mathsf{v})}{\deg(\mathsf{v})}$, where $\alpha = 0.85$ is the damping factor. Since the reciprocal out-degree is precomputed and stored as a vertex attribute, the computations in ApplyV involve only secure additions and multiplications. Unless stated otherwise, all benchmarks report the execution time and communication for 10 iterations.

A. Comparison with Graphiti

Message-passing computation. We first isolate the improvement achieved by the DCC message-passing approach by comparing only the message-passing phase of GraSP and Graphiti. Excluding the application-specific rounds required to evaluate f_{AE} and f_{AV}, the secure realisation of DCC relies on Permute and Share and Amortised Permute and Share and incurs only five rounds per iteration, independent of the number of parties. In contrast, the corresponding message-passing operations in Graphiti require $3n$ rounds. We therefore fix the graph size to $|\mathsf{V}|+|\mathsf{E}| = 2^{19}$, vary the number of parties from 2 to 10, and report the results in Fig. 6 and Table IV (Appendix D). As expected, the runtime of Graphiti increases approximately linearly with n, from 33.57 s for two parties to 259.88 s for ten parties. In contrast, the runtime of GraSP remains nearly constant at approximately 22 s. Consequently, the runtime improvement increases from 1.61× for two parties to 11.98× for ten parties. The communication of both protocols also increases approximately linearly with n.

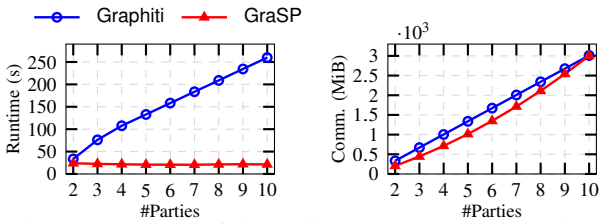


Fig. 6: Comparison of the online message-passing phase of GraSP and Graphiti while varying the number of parties, for graph size $|\mathsf{V}| + |\mathsf{E}| = 2^{19}$.

End-to-end PageRank computation. We next compare the end-to-end online cost, comprising initialisation and ten PageRank message-passing iterations. The results, while varying the number of parties, are presented in Fig. 7(c–d) and Table IV (Appendix D). Among the configurations for which Graphiti completes its initialisation, at $n = 10$, GraSP achieves runtime and communication improvements of 88.24× and 37.07×, respectively. These improvements primarily arise from the lightweight initialisation of GraSP, in which the parties exchange permutations in a single round. In contrast, the initialisation of Graphiti performs secure sorting whose cost grows with both the graph size and the number of parties.[8] Finally, we fix $n = 5$, vary the graph size from 2^{16} to 2^{22}, and report the results in Fig. 7(a–b) and Table III (Appendix D). At the smallest evaluated graph size, 2^{16}, GraSP improves the end-to-end runtime and communication by 26× and 41.32×, respectively. The gains increase with the graph size: at 2^{20}, the largest configuration completed by both systems, GraSP achieves a 48.11× runtime improvement and a 51.95× communication reduction. For graph sizes 2^{21} and 2^{22}, Graphiti runs out of memory during initialisation, while GraSP successfully completes the computation.

B. Comparison with RingSG

We next compare GraSP with RingSG by executing its publicly available implementation in our benchmark environment. To ensure a like-for-like comparison, we use RingSG's graph-generation procedure and its WAN configuration of 10 ms latency and 200 Mbit/s bandwidth. For this comparison, we also follow the same procedure as [2] for generating random graph. The vertices are distributed evenly among the parties, the average incoming and outgoing degree is set to three. If each party owns 2^x vertices, then the resulting global graph satisfies $|\mathsf{V}| + |\mathsf{E}| = 4n \cdot 2^x$. In our experiments, we vary the relevant graph and system parameters as described below.

We first fix $n = 5$ and vary the number of vertices owned by each party from 2^{12} to 2^{16}, corresponding to global graph

[6] https://github.com/CBackyx/RingSG/blob/dev-graph
[7] https://anonymous.4open.science/r/GraSP_CCS

[8] The secure sorting operations in the initialisation phase of Graphiti are micro-benchmarked using the secure quicksort variant described in [4].

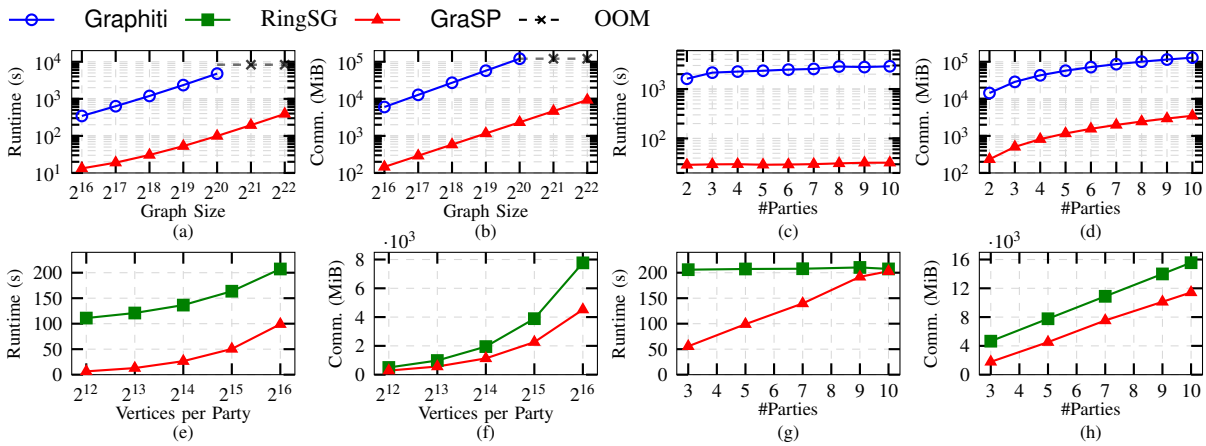


Fig. 7: End-to-end online PageRank comparison of GraSP with Graphiti [5] and RingSG [2], including initialisation and 10 message-passing iterations. (a–d) Comparison with Graphiti in the WAN setting (100 ms latency and 100 Mbps bandwidth): runtime and communication while varying the graph size with $n = 5$ in (a–b), and while varying the number of parties with $|V| + |E| = 2^{19}$ in (c–d). Dashed lines with × markers indicate configurations for which Graphiti's initialisation does not complete. (e–h) Comparison with RingSG in the WAN setting (10 ms latency and 200 Mbps bandwidth). Panels (e–f) vary the vertices per party with $n = 5$, while panels (g–h) vary the number of parties with 2^{16} vertices per party.

sizes from $20 \cdot 2^{12}$ to $20 \cdot 2^{16}$. The results are reported in Fig. 7(e–f) and Table VII (Appendix D). Across all evaluated graph sizes, GraSP outperforms RingSG in both runtime and communication. In particular, GraSP achieves runtime improvements ranging from 17.19× at 2^{12} local vertices to 2.09× at 2^{16}, while reducing communication by 1.72–1.76×. The runtime improvement primarily follows from the constant-round message-passing protocol of GraSP, whereas RingSG requires $O(\log |E|)$ rounds. Although RingSG has an asymptotically better communication complexity that is independent of the number of parties, its larger concrete constants dominate for $n = 5$, resulting in higher communication than GraSP throughout the evaluated range.

Following [2], we next fix the local graph size at 2^{16} vertices per party and vary n from 3 to 10. Thus, the global graph size grows proportionally with n, from $12 \cdot 2^{16}$ to $40 \cdot 2^{16}$. The results are shown in Fig. 7(g–h) and Table VIII (Appendix D). At $n = 3$, GraSP improves runtime and communication by 3.70× and 2.61×, respectively. These improvements decrease as n grows i.e., at $n = 10$, the two systems are nearly matched in runtime, with GraSP providing a 1.02× improvement, while GraSP still reduces communication by 1.36×. The observed runtime trends suggest that RingSG eventually becomes faster for sufficiently large values of n. This trend has two causes. First, RingSG incurs a relatively large but n-independent constant multiplying $|V| + |E|$, whereas the communication of GraSP additionally depends on n. Hence, as n increases, the lower concrete cost of GraSP is gradually offset by its dependence on the number of parties. Second, RingSG assumes that no two parties collude, allowing many intermediate values to be shared and processed among only a constant number of parties. Its per-party computation is therefore proportional primarily to the size of the corresponding local graph. In contrast, GraSP operates in the general n-party setting and tolerates collusion among up to $n-1$ parties. Consequently, all parties participate in computations over globally shared values, and the local computation performed by each party is proportional to the global graph size. Nevertheless, for all evaluated and practically relevant party counts, GraSP outperforms or remains on par with RingSG in runtime, communicates less, and provides a substantially stronger security guarantee.

C. Applications

To showcase the practicality of GraSP, we implement and benchmark two representative anti-money-laundering workloads: risk propagation and group-connection detection. In *risk propagation*, every account's risk score is iteratively updated using the transaction-weighted scores of accounts from which it receives funds, allowing suspicion to propagate through multi-hop flows. In *group-connection detection*, the parties determine whether any account in a suspicious source group is connected to an account in a target group within a bounded number of transfers. The details of how these applications can be formulated as message-passing algorithms appear in Appendix C. Table IX (Appendix D) reports the benchmarks for these applications. For a graph of size 2^{22} distributed across eight parties, GraSP completes 10 hop group-connection detection in 78.40 seconds and 10 iteration risk propagation in 202.63 seconds.

VIII. Conclusion

We design GraSP, a generic secure framework for efficiently evaluating message-passing graph algorithms. For this, we introduce a novel Decompose-Compute-Combine (DCC) approach that leverages the knowledge of the topology of the subgraph held by a data owner (or party). To achieve the claimed efficiency and scalability, GraSP completely eliminates the need for expensive operations, such as secure sort and secure shuffle, which constituted the primary bottleneck in prior works. GraSP instead relies on a much simpler and efficient Permute and Share primitive and further improves it. All this enables GraSP to achieve improvements of up to 78× compared to prior works. We further demonstrate the applicability of GraSP to collaborative anti-money-laundering by

formulating transaction-risk propagation and bounded group-connection detection as secure message-passing computations.

REFERENCES

[1] M. B. van Egmond, V. Dunning, S. van den Berg, T. Rooijakkers, A. Sangers, T. Poppe, and J. Veldsink, "Privacy-preserving anti-money laundering using secure multi-party computation," *Cryptology ePrint Archive*, 2024.

[2] Z. Zou, Z. Liu, J. Shan, Q. Li, K. Xu, and M. Xu, "Ringsg: Optimal secure vertex-centric computation for collaborative graph processing," in *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, 2025.

[3] K. Nayak, X. S. Wang, S. Ioannidis, U. Weinsberg, N. Taft, and E. Shi, "Graphsc: Parallel secure computation made easy," in *IEEE S&P*, 2015.

[4] T. Araki, J. Furukawa, K. Ohara, B. Pinkas, H. Rosemarin, and H. Tsuchida, "Secure graph analysis at scale," in *ACM CCS*, 2021.

[5] N. Koti, V. B. Kukkala, A. Patra, and B. Raj Gopal, "Graphiti: Secure graph computation made more scalable," in *ACM CCS 2024*, 2024.

[6] Z. Zou, Z. Liu, J. Shan, Q. Li, K. Xu, and M. Xu, "Cognn: Towards secure and efficient collaborative graph learning," in *ACM CCS 2024*, 2024.

[7] S. Eskandarian and D. Boneh, "Clarion: Anonymous communication from multiparty shuffling protocols," in *NDSS*, 2022.

[8] P. Mohassel and P. Rindal, "ABY^3: A mixed protocol framework for machine learning," in *ACM CCS*, 2018.

[9] H. Chaudhari, A. Choudhury, A. Patra, and A. Suresh, "ASTRA: High Throughput 3PC over Rings with Application to Secure Prediction," in *ACM CCSW@CCS*, 2019.

[10] A. Patra and A. Suresh, "BLAZE: Blazing Fast Privacy-Preserving Machine Learning," in *NDSS*, 2020.

[11] E. Boyle, N. Gilboa, Y. Ishai, and A. Nof, "Practical fully secure three-party computation via sublinear distributed zero-knowledge proofs," in *ACM CCS*, 2019.

[12] N. Koti, M. Pancholi, A. Patra, and A. Suresh, "SWIFT: Super-fast and robust privacy-preserving machine learning," in *USENIX Security*, 2021.

[13] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart, "Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits," in *ESORICS*, 2013, pp. 1–18.

[14] M. Chase, E. Ghosh, and O. Poburinnaya, "Secret-shared shuffle," in *ASIACRYPT*, 2020.

[15] N. Koti, V. B. Kukkala, A. Patra, B. R. Gopal *et al.*, "Find thy neighbourhood: Privacy-preserving local clustering," *PoPETs*, 2023.

[16] N. Koti, V. B. Kukkala, A. Patra, and B. R. Gopal, "Entrada to secure graph convolutional networks," *Cryptology ePrint Archive*, 2023.

[17] Y. Lindell, "How to simulate it–a tutorial on the simulation proof technique," *Tutorials on the Foundations of Cryptography: Dedicated to Oded Goldreich*, pp. 277–346, 2017.

[18] R. Canetti, "Security and composition of multiparty cryptographic protocols," *Journal of CRYPTOLOGY*, vol. 13, no. 1, pp. 143–202, 2000.

[19] I. Damgård and J. B. Nielsen, "Scalable and unconditionally secure multiparty computation," in *CRYPTO*, 2007.

[20] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias, "Multiparty computation from somewhat homomorphic encryption," in *CRYPTO*, 2012, pp. 643–662.

[21] D. Beaver, "Efficient multiparty protocols using circuit randomization," in *CRYPTO*, 1992.

[22] T. Lu, X. Kang, B. Zhang, Z. Ma, X. Zhang, Y. Liu, K. Ren, and C. Chen, "Efficient {2PC} for constant round secure equality testing and comparison," in *34th USENIX Security Symposium (USENIX Security 25)*, 2025, pp. 1985–2004.

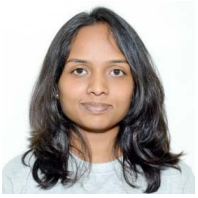
IX. BIOGRAPHY SECTION



Siddharth Kapoor completed his B.E. in Computer Science from BITS Pilani in 2022 and worked as a Software Engineer at Amazon for 2 years. He is a Ph.D. student at University of Utah. His research interests lie in applied cryptography focusing on secure multiparty computation and privacy-preserving machine learning.



Nishat Koti is an independent researcher who obtained her PhD from Indian Insitute of Science, Bangalore, India. She then undertook a post-doctoral research position at TU Darmstadt, Germany and worked as a cryptography researcher at Aztec Labs.



Varsha Bhat Kukkala is an Assistant Professor at Indian Institute of Technology, Tirupati, working in the area of secure multiparty computation. Prior to this, she was a postdoctoral fellow at Indian Institute of Science and completed her PhD from Indian Institute of Technology, Ropar.



Arpita Patra is a Professor at the Indian Institute of Science, Bangalore, India. She received her MS (by Research) and PhD from the Indian Institute of Technology (IIT), Madras and held post-doctoral positions at the University of Bristol, ETH Zurich, and Aarhus University. She specializes in cryptography with the core focus on the theoretical foundations of cryptography.



Bhavish Raj Gopal received his BS and MS degree in Mathematics from the IISER, Mohali. He is a Ph.D. student at Indian Institute of Science, Bangalore. His research interests lie in applied cryptography focusing on secure multiparty computation and privacy-preserving machine learning.

APPENDIX A
PRELIMS

A. Notations

A list of common notations used in the paper, appears in Table II.

Symbol	Description
n	Number of parties
$\mathcal{G} = (\mathsf{V}, \mathsf{E})$	Global graph
$\|\mathsf{V}\|, \|\mathsf{E}\|, \mathsf{N}$	size of vertexset, edgeset, $\mathsf{N} = \|\mathsf{V}\| + \|\mathsf{E}\|$
V, E	Vertex set; edge set
V_i	Vertices owned by party P_i
E_i	Edges with destination in V_i
$\tilde{\mathsf{V}}_i$	V_i and its neighbours (local subgraph vertices)
$\widetilde{\mathsf{V}_i}$	V_i and its immediate neighbours padded up to $\min(\|\mathsf{V}\|, \|\mathsf{V}_i\| + \|\mathsf{E}_i\|)$
G	$(\mathsf{V}, \{\mathsf{E}_i\})$ List representation of global graph $\mathcal{G}$
DAG-list	Data augmented list representation
G_i	DAG-list representation for subgraph $\mathcal{G}_i$
$X[i:j]$	List containing entries of X from $X[i]$ to $X[j]$

TABLE II: Summary of Notations

B. Shared key setup

$\mathcal{F}_{\mathsf{setup}}$ enables the establishment of common random keys for a pseudo-random function (PRF) F among parties. This aids in non-interactively generating correlated randomness. Here $\mathsf{F} : \{0,1\}^\kappa \to \mathbb{Z}_{2^\ell}\mathsf{F}$ is a secure PRF. The shared key setup enables all parties to non-interactively generate $\langle\cdot\rangle$-shares of a value r. This is done by each party locally sampling a random value for its share $\langle \mathsf{r}\rangle_i$ using the PRF key k_i and setting $\mathsf{r} = \sum_{i\in[n]} \langle \mathsf{r}\rangle$. The shared key setup also enables non-interactive generation of $\langle\cdot\rangle$-shares of an input value x held by a party $P_i \in \mathcal{P}$. For this all parties $P_j \in \mathcal{P}$ sample a common random value jointly with P_i as their share of x using the key k_{ij} and party P_i sets its share as $\langle \mathsf{x}\rangle_i = \mathsf{x} - \sum_{j\in[n], j\neq i} \langle \mathsf{x}\rangle_j$.

Functionality $\mathcal{F}_{\mathsf{setup}}$

$\mathcal{F}_{\mathsf{setup}}$ proceeds as follows.

- $\mathcal{F}_{\mathsf{setup}}$ picks random keys $\{\{k_i\}_{i\in[n]}, \{k_{ij}\}_{i,j\in[n]}, k_{all}\}$.
- Send $(\mathsf{Output}, k_i, \{k_{ij}\}_{j\in[n]}, k_{all})$ to party P_i for $i \in [n]$.

Fig. 8: Ideal functionality for setup.

C. Fixed point arithmetic

All real-valued quantities are represented over $\mathbb{Z}_{2^\ell}$ using signed fixed-point encoding, with $\ell = 64$ in our implementation. For a real value $\tilde{x}$, its fixed-point representation with f fractional bits is $x = \lfloor 2^f \tilde{x} \rceil \bmod 2^\ell$. Values in $[0, 2^{\ell-1})$ represent non-negative numbers, while values in $[2^{\ell-1}, 2^\ell)$ represent negative numbers interpreted using the usual two's-complement convention. Public real-valued constants, including the PageRank damping factor and the risk-propagation coefficients, are encoded using the same scale. Reciprocals, such as inverse out-degrees and inverse incoming transaction totals, are computed in the clear by the party holding the corresponding private value and subsequently secret-shared in fixed-point form. When the denominator is zero, its encoded reciprocal is defined to be zero, which makes the corresponding incoming-contribution term zero. The fixed-point precision and input ranges are selected so that all intermediate values remain within the signed ℓ bit range and no modular wraparound occurs in the evaluated benchmarks.

D. Secure computation of graph algorithms

Intuition for Propagate and Gather algorithm of [5]. At a high level, Propagate is realised by reducing "copy the source vertex's value to all its outgoing edges" to a cumulative-sum computation. The key observation is that in source order, each vertex is immediately followed by its outgoing edges, so if one ensures that only vertex entries contribute a non-zero value (and every edge contributes 0), then a linear scan that computes the cumulative sum over all preceding entries will write a constant value across the entire block consisting of a node followed by its outgoing edges. However, a naive cumulative sum would also accumulate contributions from earlier nodes. To prevent this, the work first switches to the vertex order, where all vertices appear contiguously (in the same relative order as they would appear in source order). In this ordering, vertex values are modified into increments rather than absolute values, i.e. the first vertex keeps its value, while every subsequent vertex stores the difference between its value and the previous vertex's value. This transformation is crucial because, when the list is reordered to source order, the cumulative sum correctly propagates the source vertex value to subsequent edges. Gather follows the dual idea for aggregation. In destination order, all incoming edges of a vertex appear contiguously before the vertex entry, so a linear scan that computes cumulative sums of the edge messages naturally accumulates the total incoming sum. But again, the cumulative sum at a vertex also contains a carry-over from everything that appeared before this node's incoming-edge block. However, the carry-over at a vertex is exactly the cumulative value already held by the previous vertex. Thus, to remove this carry-over, the list is reordered to the vertex order, where vertices are contiguous, and each vertex subtracts the previous vertex's temporary cumulative value to recover precisely the aggregation of its own incoming edges. A pictorial illustration of Propagate and Gather for message passing computation involved in identifying one-hop neighbours (see Fig. 1) appears in Fig. 9 and Fig. 10.

Initialisation phase of Graphiti. In a naive approach, transitioning between different orderings of the DAG-list would require secure sorting, which incurs significant overhead. However, [4] observed that the required transitions remain consistent across multiple message-passing rounds. Hence, the secret shares of mappings that allow transitioning between different orderings of the DAG-list can be generated in a one-time initialisation phase. The initialisation phase begins with the DAG-list in vertex order. Using a secure shuffle protocol,

(1) Vertex order $\mathsf{data_r}[i] \leftarrow \mathsf{data_s}[i] - \mathsf{data_s}[i-1],\ \forall i \le |V|$

	v_1	v_2	v_3	e_{12}	e_{13}	e_{23}	e_{31}	e_{32}
$\mathsf{data_s}$	1	0	0	0	0	0	0	0
$\mathsf{data_r}$	1	−1	0	0	0	0	0	0
$\mathsf{data_e}$	0	0	0	0	0	0	0	0

(2) Transition

(3) Source order $\mathsf{data_e}[i] \leftarrow \sum_{j \le i} \mathsf{data_r}[j] - \mathsf{data_s},\ \forall i \le |N|$

	v_1	e_{12}	e_{13}	v_2	e_{23}	v_3	e_{31}	e_{32}
$\mathsf{data_s}$	1	0	0	0	0	0	0	0
$\mathsf{data_r}$	1	0	0	−1	0	0	0	0
$\mathsf{data_e}$	0	1	1	0	0	0	0	0

Fig. 9: Steps in Propagate for the example in Fig. 1.

(1) Destination order $\mathsf{data'_g}[i] \leftarrow \sum_{j \le i} \mathsf{data_r}[j],\ \forall i \le |N|$

	e_{31}	v_1	e_{12}	e_{32}	v_2	e_{13}	e_{23}	v_3
data_r	0	0	1	0	0	1	0	0
$\mathsf{data'}_g$	0	0	1	1	1	2	2	2

(2) Transition

(3) Vertex order $\mathsf{data_g}[i] \leftarrow \mathsf{data'_g}[i] - \mathsf{data'_g}[i-1],\ \forall i \le |V|$

	v_1	v_2	v_3	e_{12}	e_{13}	e_{23}	e_{31}	e_{32}
$\mathsf{data'}_g$	0	1	2	1	2	2	0	1
$\mathsf{data_g}$	0	1	1	0	0	0	0	0

Fig. 10: Gather for the example in Fig. 1.

the parties apply a random permutation π^A to produce a randomised ordering, denoted as Shuffle-A. Next, another random permutation π^B is applied to Shuffle-A, resulting in a new randomised ordering, Shuffle-B. The parties then compute mappings from these randomised orderings to the source and destination orders. Specifically, the mapping from Shuffle-A to the source order is computed using a secure sort protocol. Since this mapping only links a random ordering to a sorted list, it does not reveal any sensitive information and can be made public, denoted as π^S. Similarly, the public mapping π^D from Shuffle-B to the destination order is computed. These precomputed secret-shared permutations (π^A, π^B) and public mappings (π^S, π^D) facilitate efficient transitions during the message-passing phase. Using these precomputed mappings, transitions between DAG-list orderings can be efficiently performed. For transitioning from vertex order to source order, the secret permutation π^A is applied via a secure shuffle to reorder the DAG-list into Shuffle-A, followed by locally applying the public permutation π^S to obtain the source order. Transitioning from source order to destination order involves reversing π^S via a secure shuffle to return to Shuffle-A, applying π^B to reorder into Shuffle-B, and finally applying π^D locally to reach the destination order. Transitioning from destination order to source order involves reversing π^S via a secure shuffle to return to Shuffle-A, applying π^B to reorder into Shuffle-B, and finally applying π^D locally to reach the destination order. This approach ensures that transitions during message-passing are secure and efficient, with round complexity independent of the graph size. An illustration of the initialisation phase appears in Fig. 11.

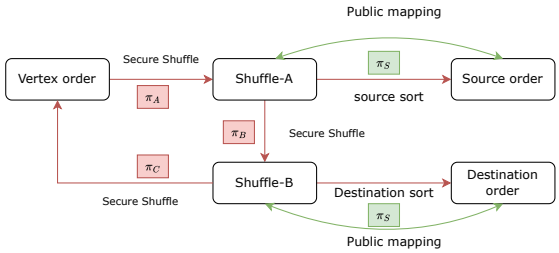


Fig. 11: Initialisation in Graphiti.

Complexity of [5]: When instantiated in n-party MPC protocol, initialisation phase requires $O(n + \log \mathsf{N})$ rounds of interaction and $O(n\mathsf{N} \log \mathsf{N})$ communication. Here, n denotes the number of parties, $\mathsf{N} = |\mathsf{V}| + |\mathsf{E}|$ denotes the size of G. Each iteration of message-passing computations, incurs a cost of $O(n + \mathsf{r_{AE}} + \mathsf{r_{AV}})$ rounds and $O(n\mathsf{N} + \mathsf{c_{AE}} \cdot \mathsf{N} + \mathsf{c_{AV}} \cdot |\mathsf{V}|)$ communication. Here, r_x and c_x denote the round and communication complexity of computing x securely. In case of applications that require non-linear aggregation, Graphiti incurs $O(n \log |\mathsf{V}| + \mathsf{r_{AE}} + \mathsf{r_{AV}})$ rounds and $O(n\mathsf{N} \log |\mathsf{V}| + \mathsf{c_{AE}} \cdot \mathsf{N} + \mathsf{c_{AV}} \cdot |\mathsf{V}|)$ communication. See [4], [5] for further details. Finally, in the preprocessing phase, [5] incurs communication of $O(n\mathsf{N} \log \mathsf{N})$ for initialisation and $O(n^2\mathsf{N} \log \mathsf{N})$ for one iteration of message-passing computations.

Appendix B
First Attempt

Fig. 12 gives the formal details of the secure realisation of the DCC algorithm described in §III-A.

Protocol Π_{DCC} (First attempt)

Inputs: The parties hold $\langle\cdot\rangle$-shares of $\mathsf{G} = (\mathsf{V}, \{\mathsf{E}_i\}_{i\in[n]})$ and agree on the graph algorithm alg, including the number of iterations k, the propagated and aggregated data, the aggregation operation, and the functions f_{AE} and f_{AV}.
Outputs: The parties hold $\langle\cdot\rangle$-shares of $\mathsf{G} = (\mathsf{V}, \{\mathsf{E}_i\}_{i\in[n]})$ with its vertex and edge states updated according to alg.

Initialisation: For every $i \in [n]$, let $\mathsf{G}_i = \mathsf{V} \| \mathsf{E}_i$ and $N_i = |\mathsf{V}| + |\mathsf{E}_i|$. Using its knowledge of E_i, party P_i defines permutations $\pi_i^S, \pi_i^D, \pi_i^V$ over $[N_i]$ that map vertex order to source order, source order to destination order, and destination order back to vertex order, respectively. Party P_i samples independent uniformly random permutations $\hat{\pi}_i^S, \hat{\pi}_i^D, \hat{\pi}_i^V$ over $[N_i]$ and publishes $\sigma_i^S = \pi_i^S \circ (\hat{\pi}_i^S)^{-1}$, $\sigma_i^D = \pi_i^D \circ (\hat{\pi}_i^D)^{-1}$, and $\sigma_i^V = \pi_i^V \circ (\hat{\pi}_i^V)^{-1}$.

DCC message-passing computation: For $r = 1, \ldots, k$, perform the following.

- **Decompose.** For every $i \in [n]$ in parallel, the parties locally set $\langle \mathsf{G}_i \rangle = \langle \mathsf{V} \rangle \| \langle \mathsf{E}_i \rangle$. Thus, G_i is in vertex order and has size N_i.
- **Compute.** For every $i \in [n]$ in parallel, perform the following.
  - **Propagate in vertex order.** Set $\langle \mathsf{G}_i[1].\mathsf{data_r} \rangle = \langle \mathsf{G}_i[1].\mathsf{data_s} \rangle$. For $q = 2, \ldots, |\mathsf{V}|$, set $\langle \mathsf{G}_i[q].\mathsf{data_r} \rangle = \langle \mathsf{G}_i[q].\mathsf{data_s} \rangle - \langle \mathsf{G}_i[q-1].\mathsf{data_s} \rangle$. For $q = |\mathsf{V}| + 1, \ldots, N_i$, set $\langle \mathsf{G}_i[q].\mathsf{data_r} \rangle = 0$.
  - Compute $\langle \mathsf{G}_i \rangle \leftarrow \mathcal{F}_{\mathsf{PS}}(\langle \mathsf{G}_i \rangle, \{P_i : \hat{\pi}_i^S\})$ and locally apply σ_i^S to obtain source order.
  - For $q = 1, \ldots, N_i$, set $\langle \mathsf{G}_i[q].\mathsf{data_r} \rangle = \sum_{h=1}^{q} \langle \mathsf{G}_i[h].\mathsf{data_r} \rangle - \langle \mathsf{G}_i[q].\mathsf{data_s} \rangle$.
  - **ApplyE.** For $q = 1, \ldots, N_i$, compute $\langle z_q \rangle \leftarrow \mathcal{F}_{\mathsf{MPC}}(\langle \mathsf{G}_i[q].\mathsf{data_e} \rangle, \langle \mathsf{G}_i[q].\mathsf{data_r} \rangle, f_{\mathsf{AE}})$ and

$\langle \mathsf{G}_i[q].\mathsf{data_e}\rangle \leftarrow \mathcal{F}_{\mathsf{mult}}(\langle z_q\rangle, 1 - \langle \mathsf{G}_i[q].\mathsf{isV}\rangle)$.
- Compute $\langle \mathsf{G}_i\rangle \leftarrow \mathcal{F}_{\mathsf{PS}}(\langle \mathsf{G}_i\rangle, \{P_i : \hat{\pi}_i^D\})$ and locally apply σ_i^D to obtain destination order.
- **Gather in destination order.** For $q = 1, \ldots, N_i$, set $\langle \mathsf{G}_i[q].\mathsf{data_r}\rangle = \sum_{h=1}^{q} \langle \mathsf{G}_i[h].\mathsf{data_e}\rangle$.
- Compute $\langle \mathsf{G}_i\rangle \leftarrow \mathcal{F}_{\mathsf{PS}}(\langle \mathsf{G}_i\rangle, \{P_i : \hat{\pi}_i^V\})$ and locally apply σ_i^V to obtain vertex order.
- Set $\langle \mathsf{G}_i[1].\mathsf{data_g}\rangle = \langle \mathsf{G}_i[1].\mathsf{data_r}\rangle$. For $q = 2, \ldots, |\mathsf{V}|$, set $\langle \mathsf{G}_i[q].\mathsf{data_g}\rangle = \langle \mathsf{G}_i[q].\mathsf{data_r}\rangle - \langle \mathsf{G}_i[q-1].\mathsf{data_r}\rangle$.
- **ApplyV.** Let $b_i = 1 + \sum_{h=1}^{i-1} |\mathsf{V}_h|$ and $e_i = b_i + |\mathsf{V}_i| - 1$. For $q = b_i, \ldots, e_i$, compute $\langle \mathsf{G}_i[q].\mathsf{data_v}\rangle \leftarrow \mathcal{F}_{\mathsf{MPC}}(\langle \mathsf{G}_i[q].\mathsf{data_v}\rangle, \langle \mathsf{G}_i[q].\mathsf{data_g}\rangle, f_{\mathsf{AV}})$.

- **Combine.** For every $i \in [n]$, let $b_i = 1 + \sum_{h=1}^{i-1} |\mathsf{V}_h|$ and $e_i = b_i + |\mathsf{V}_i| - 1$, and locally set $\langle \mathsf{V}_i\rangle = \langle \mathsf{G}_i[b_i : e_i]\rangle$ and $\langle \mathsf{E}_i\rangle = \langle \mathsf{G}_i[|\mathsf{V}| + 1 : |\mathsf{V}| + |\mathsf{E}_i|]\rangle$. The parties then set $\langle \mathsf{V}\rangle = \langle \mathsf{V}_1\rangle \| \cdots \| \langle \mathsf{V}_n\rangle$ and $\langle \mathsf{G}\rangle = (\langle \mathsf{V}\rangle, \{\langle \mathsf{E}_i\rangle\}_{i\in[n]})$.

Fig. 12: First attempt at secure realisation of DCC for secure collaborative graph processing.

APPENDIX C
ANTI-MONEY-LAUNDERING APPLICATIONS

Financial institutions model inter-account transactions as a directed, weighted graph, where vertices represent bank accounts and edges represent fund transfers. Such graphs are useful for detecting the layering stage of money laundering, in which illicit funds are routed through multiple accounts and institutions to obscure their origin. We consider the secure computation of two representative anti-money-laundering analysis metrics. First, *risk propagation* iteratively updates each account's risk score using the transaction-weighted risk scores of accounts from which it receives funds, thereby propagating suspicion along multi-hop money flows. Second, *group-connection detection* determines whether any account in a designated suspicious source group A is connected, within a bounded number of transfers, to an account in a suspicious target group B. Both computations are naturally expressed as message-passing algorithms: account states are propagated along transaction edges, aggregated at receiving accounts, and updated over multiple iterations. They therefore serve as representative applications of GraSP for privacy-preserving collaborative anti-money-laundering analysis across multiple financial institutions.

A. Risk-Propagation Benchmark.

The risk-propagation algorithm updates the risk score of each account using a weighted average of the scores of accounts from which it receives money. Let $A_{i,j}$ denote the total amount transferred from account i to account j, and let $T_j = \sum_{i\in S_{\text{in}}(j)} A_{i,j}$ be the total incoming amount of j. At iteration k, the risk score of j is updated as

$$r_j^k = (1-\delta_k) r_j^{k-1} + \frac{\delta_k}{T_j} \sum_{i\in S_{\text{in}}(j)} A_{i,j} r_i^{k-1}, \qquad \delta_k = \frac{a}{k^b}.$$

Thus, δ_k controls the influence of neighbouring accounts, while repeated iterations propagate risk across multiple transaction hops.

This computation is naturally expressed as message passing. Each vertex i stores its current risk score r_i and the precomputed value T_i^{-1}, while each directed edge (i, j) stores its transaction weight $A_{i,j}$. During Scatter, vertex i sends its current score to every outgoing edge, and ApplyE computes the message $m_{i,j} = A_{i,j} r_i$. The Gather operation sums the messages on the incoming edges of each vertex, $M_j = \sum_{i\in S_{\text{in}}(j)} m_{i,j}$, after which ApplyV updates

$$r_j \leftarrow (1-\delta_k) r_j + \delta_k T_j^{-1} M_j.$$

Repeating these operations for K iterations propagates risk through paths of increasing length in the transaction graph.

B. Group-Connection Detection.

The algorithm determines whether there exists a directed path of at most K transfers from any vertex in group A to any vertex in group B. Each vertex v stores two public or secret-shared indicator attributes $v.\mathsf{isA}, v.\mathsf{isB} \in \{0, 1\}$, denoting membership in A and B, respectively, together with a propagation value x_v. Initially, $x_v^0 = v.\mathsf{isA}$.

The computation proceeds for K message-passing iterations. In iteration k, Scatter propagates x_u^{k-1} along every outgoing edge (u, v), while Gather sums all values received by v. The vertex update is

$$x_v^k = x_v^{k-1} + \sum_{(u,v)\in \mathsf{E}} x_u^{k-1}.$$

Thus, after K iterations, x_v^K is nonzero whenever at least one directed path of length at most K connects a vertex in A to v.

A final ApplyV operation restricts the result to group B by computing

$$y_v = v.\mathsf{isB} \cdot x_v^K.$$

The parties then securely sum these values,

$$z = \sum_{v\in \mathsf{V}} y_v,$$

and perform an equality test with zero. The output bit is

$$c = 1 - \mathsf{EQ}(z, 0),$$

which equals one exactly when some vertex in B is reachable from a vertex in A within at most K transfers.

APPENDIX D
BENCHMARKS

Comparison with Graphiti. The online runtime and communication while varying the graph size and the number of parties are reported in Table III and Table IV, respectively. The preprocessing runtime and communication while varying the graph size and the number of parties are reported in Table V and Table VI, respectively.

Comparison with RingSG. The online runtime and communication while varying the graph size and the number of parties are reported in Table VII and Table VIII, respectively.

Applications: The online runtime and communication of the group-connection and risk-propagation applications while varying the number of parties are reported in Table IX.

Graph Size	Protocol	Runtime (s)		Communication (MiB)	
		Init.	MPA	Init.	MPA
2^{16}	Graphiti	312.26	32.54	5849.01	167.20
	GraSP	2.75	10.51	18.80	126.80
2^{17}	Graphiti	570.42	61.53	12472.01	334.40
	GraSP	4.56	14.43	37.60	253.60
2^{18}	Graphiti	1105.65	89.83	26492.01	668.80
	GraSP	8.51	22.04	75.20	507.20
2^{19}	Graphiti	2218.20	129.85	56080.01	1337.60
	GraSP	15.78	37.18	150.40	1014.39
2^{20}	Graphiti	4561.98	220.42	118352.01	2675.20
	GraSP	30.08	69.32	300.80	2028.80
2^{21}	Graphiti	OOM	413.47	OOM	5350.40
	GraSP	56.69	138.29	601.60	4057.60
2^{22}	Graphiti	OOM	751.20	OOM	10700.80
	GraSP	107.24	284.34	1203.20	8115.20

TABLE III: Online comparison of GraSP and Graphiti while varying the graph size ($n = 5$).

#Parties	Protocol	Runtime (s)		Communication (MiB)	
		Init.	MPA	Init.	MPA
2	Graphiti	1581.96	33.57	14020.00	334.40
	GraSP	5.30	20.87	28.00	205.60
3	Graphiti	2071.23	76.19	28040.00	668.80
	GraSP	7.35	22.51	62.40	443.20
4	Graphiti	2149.75	107.60	42060.00	1003.20
	GraSP	8.41	21.64	103.20	712.80
5	Graphiti	2214.83	132.81	56080.01	1337.60
	GraSP	8.46	20.86	150.40	1014.39
6	Graphiti	2316.39	158.26	70100.01	1672.00
	GraSP	9.11	20.78	204.00	1347.99
7	Graphiti	2371.21	183.54	84120.01	2006.40
	GraSP	9.48	20.84	264.00	1713.59
8	Graphiti	2649.31	208.84	98140.01	2340.80
	GraSP	10.01	21.00	330.40	2111.18
9	Graphiti	2550.25	234.39	112160.01	2675.20
	GraSP	10.55	22.12	403.20	2540.78
10	Graphiti	2631.84	259.88	126180.01	3009.60
	GraSP	11.08	21.69	482.40	3002.38

TABLE IV: Online comparison of GraSP and Graphiti while varying the number of parties, for graph size $|\mathsf{V}| + |\mathsf{E}| = 2^{19}$.

APPENDIX E
CORRECTNESS OF DCC

Lemma E.1 (Correctness of DCC): Let $\mathcal{G} = (\mathsf{V}, \mathsf{E})$ be a directed graph such that $\mathsf{V} = \dot{\bigcup}_{i\in[n]} \mathsf{V}_i$ and $\mathsf{E} = \dot{\bigcup}_{i\in[n]} \mathsf{E}_i$, where $\mathsf{E}_i = \{(\mathsf{u}, \mathsf{v}) \in \mathsf{E} : \mathsf{v} \in \mathsf{V}_i\}$.

For every $i \in [n]$, let $\bar{\mathsf{V}}_i = \mathsf{V}_i \cup \{\mathsf{u} \in \mathsf{V} : (\mathsf{u}, \mathsf{v}) \in \mathsf{E}_i \text{ for some } \mathsf{v} \in \mathsf{V}_i\}$. Let $\widetilde{\mathsf{V}}_i$ be any padded superset of $\bar{\mathsf{V}}_i$, and define $\mathcal{G}_i = (\widetilde{\mathsf{V}}_i, \mathsf{E}_i)$. Assume that, at the beginning of an iteration, every occurrence of a vertex u in each subgraph containing u stores the same state st_u as in the global graph. Consider a message-passing iteration in which, for every $(\mathsf{u}, \mathsf{v}) \in \mathsf{E}$, the updated edge state and message are defined as $\mathsf{st}'_{(\mathsf{u},\mathsf{v})} = \mathsf{msg}_{\mathsf{u},\mathsf{v}} = f_{\mathsf{AE}}(\mathsf{st}_{(\mathsf{u},\mathsf{v})}, \mathsf{st}_\mathsf{u})$, the aggregate at vertex v is $\mathsf{agg}_\mathsf{v} = \sum_{\mathsf{u}:(\mathsf{u},\mathsf{v})\in\mathsf{E}} \mathsf{msg}_{\mathsf{u},\mathsf{v}}$, and the updated vertex state is $\mathsf{st}'_\mathsf{v} = f_{\mathsf{AV}}(\mathsf{st}_\mathsf{v}, \mathsf{agg}_\mathsf{v})$.

Then, executing one message-passing iteration independently on every $\mathcal{G}_i$ and combining the updated states of V_i and E_i for all $i \in [n]$ produces exactly the same updated graph state as executing the iteration directly on $\mathcal{G}$.

Proof. Fix an arbitrary $i \in [n]$. Consider an edge $(\mathsf{u}, \mathsf{v}) \in \mathsf{E}_i$. By the definition of E_i, its destination satisfies $\mathsf{v} \in \mathsf{V}_i$. Moreover, $\mathsf{u} \in \bar{\mathsf{V}}_i \subseteq \widetilde{\mathsf{V}}_i$. Hence, both endpoints required to process (u, v) are present in $\mathcal{G}_i$. By assumption, the copy of u in $\mathcal{G}_i$ stores the same state as u in the global graph. Therefore, the edge state and message computed locally satisfy $\mathsf{st}'^{(i)}_{(\mathsf{u},\mathsf{v})} = \mathsf{msg}^{(i)}_{\mathsf{u},\mathsf{v}} = f_{\mathsf{AE}}(\mathsf{st}_{(\mathsf{u},\mathsf{v})}, \mathsf{st}_\mathsf{u}) = \mathsf{msg}_{\mathsf{u},\mathsf{v}} = \mathsf{st}'_{(\mathsf{u},\mathsf{v})}$. Thus, every edge in E_i is updated exactly as in the global execution.

Next, fix a vertex $\mathsf{v} \in \mathsf{V}_i$. Since E_i contains exactly

Graph Size	Protocol	Runtime (s)	Comm. (MiB)
2^{16}	Graphiti	74.22	528.95
	GraSP	4.56	33.20
2^{17}	Graphiti	156.61	1124.90
	GraSP	8.78	66.40
2^{18}	Graphiti	331.48	2383.80
	GraSP	17.10	132.80
2^{19}	Graphiti	696.88	5035.60
	GraSP	33.75	265.60
2^{20}	Graphiti	1341.07	10607.20
	GraSP	67.66	531.20
2^{21}	Graphiti	–	–
	GraSP	133.56	1062.40
2^{22}	Graphiti	OOM	OOM
	GraSP	263.80	2124.80

TABLE V: Preprocessing cost of GraSP and Graphiti while varying the graph size with $n = 5$.

#Parties	Protocol	Runtime (s)	Comm. (MiB)
2	Graphiti	561.98	5451.60
	GraSP	15.38	194.80
3	Graphiti	598.08	5035.60
	GraSP	15.23	218.40
4	Graphiti	651.73	5035.60
	GraSP	16.91	242.00
5	Graphiti	702.88	5035.60
	GraSP	18.30	265.60
6	Graphiti	783.67	5035.60
	GraSP	20.53	289.20
7	Graphiti	815.91	5035.60
	GraSP	23.17	312.80
8	Graphiti	1070.82	5035.60
	GraSP	26.22	336.40
9	Graphiti	947.37	5035.60
	GraSP	30.08	360.00
10	Graphiti	1006.62	5035.60
	GraSP	33.78	383.60

TABLE VI: Preprocessing cost of GraSP and Graphiti while varying the number of parties, for graph size $|\mathsf{V}| + |\mathsf{E}| = 2^{19}$.

Graph Size	Protocol	Online	
		Runtime (s)	Comm. (MiB)
2^{12}	RingSG	111.019	498.44
	GraSP	6.46	282.50
2^{13}	RingSG	120.945	981.58
	GraSP	12.93	565.00
2^{14}	RingSG	136.489	1949.90
	GraSP	26.50	1130.00
2^{15}	RingSG	163.679	3889.96
	GraSP	50.48	2260.00
2^{16}	RingSG	207.320	7768.59
	GraSP	99.05	4520.00

TABLE VII: Online comparison of GraSP and RingSG while varying the graph size.

#Parties	Protocol	Online	
		Runtime (s)	Comm. (MiB)
3	RingSG	205.934	4652.20
	GraSP	55.65	1780.00
5	RingSG	207.320	7768.59
	GraSP	99.05	4520.00
7	RingSG	207.800	10886.57
	GraSP	139.66	7530.00
9	RingSG	210.329	13996.74
	GraSP	191.97	10120.00
10	RingSG	207.435	15557.03
	GraSP	202.85	11430.00

TABLE VIII: Online comparison of GraSP and RingSG while varying the number of parties.

the edges whose destination belongs to V_i, it contains every incoming edge of v and no additional edge incident towards v. Consequently, $\mathsf{agg}_\mathsf{v}^{(i)} = \sum_{\mathsf{u}:(\mathsf{u},\mathsf{v})\in\mathsf{E}_i} \mathsf{msg}_{\mathsf{u},\mathsf{v}}^{(i)} = \sum_{\mathsf{u}:(\mathsf{u},\mathsf{v})\in\mathsf{E}} \mathsf{msg}_{\mathsf{u},\mathsf{v}} = \mathsf{agg}_\mathsf{v}$. The locally updated vertex state therefore satisfies $\mathsf{st}_\mathsf{v}'^{(i)} = f_{\mathsf{AV}}(\mathsf{st}_\mathsf{v}, \mathsf{agg}_\mathsf{v}^{(i)}) = f_{\mathsf{AV}}(\mathsf{st}_\mathsf{v}, \mathsf{agg}_\mathsf{v}) = \mathsf{st}'_\mathsf{v}$. Vertices included only for padding do not affect the result: no additional edges are added to E_i, and the states of padding vertices are discarded during Combine.

Finally, $\{\mathsf{V}_i\}_{i\in[n]}$ partitions V, and $\{\mathsf{E}_i\}_{i\in[n]}$ partitions E. Hence, combining the updated vertex states from V_i and the updated edge states from E_i, for all $i \in [n]$, yields exactly the updated graph state obtained by executing the iteration directly on $\mathcal{G}$. □

Corollary E.2: Starting from identical initial states, applying Decompose, Compute, and Combine for k iterations produces the same graph state as performing k iterations of the message-passing algorithm directly on the global graph $\mathcal{G}$.

Proof. The claim follows by induction on the iteration number. The states coincide before the first iteration by assumption. Suppose they coincide at the beginning of some iteration. Lemma E.1 shows that the local DCC computations produce exactly the same updated vertex and edge states as the corresponding global iteration. Combine recovers these global states, and the subsequent Decompose step copies the updated state of each vertex into every subgraph in which it occurs. Thus, the required state-consistency invariant holds at the beginning of the next iteration. The result follows for all k iterations. □

#Parties	Group Connection		Risk Propagation	
	Runtime (s)	Comm. (MiB)	Runtime (s)	Comm. (MiB)
2	95.58	860.80	183.19	1795.20
3	85.25	1875.20	179.05	3904.00
4	79.29	3043.20	178.82	6326.40
5	74.62	4364.80	182.31	9062.40
6	75.07	5840.00	187.78	12112.00
7	75.98	7468.80	194.50	15475.19
8	78.40	9251.20	202.63	19151.99

TABLE IX: Online cost of the anti-money-laundering applications while varying the number of parties.

APPENDIX F
SECURITY PROOFS

We rely on the standard real-world/ideal-world simulation paradigm to prove the security of our protocols. Let $\mathcal{A}$ denote the real-world adversary corrupting at most $n-1$ parties and $\mathcal{S}$ denote the ideal-world adversary. We prove the security of the protocol in the $\mathcal{F}_{\mathsf{setup}}$-hybrid model where there exists an ideal functionality $\mathcal{F}_{\mathsf{setup}}$ to establish common PRF keys among parties in $\mathcal{P}$. This allows the parties to sample common random values among themselves non-interactively. Note that the simulation begins with the simulator $\mathcal{S}$ emulating $\mathcal{F}_{\mathsf{setup}}$ to establish the common keys with the adversary. Since $\mathcal{S}$ has access to the inputs and randomness of $\mathcal{A}$, it can simulate the steps in the real protocol. Without loss of generality, we assume that $P_1, ..., P_t$ for any $t < n$ are the parties corrupted by the adversary. In what follows, we first prove the security of our Amortised Permute and Share protocol, followed by the security of GraSP.

Fig. 13 presents the ideal functionality for Amortised Permute and Share and the secure protocol that realises it, Π_{APS}, appears in Fig. 14.

Functionality $\mathcal{F}_{\mathsf{APS}}$

Let $\mathcal{C} \subset \mathcal{P}$ denote the set of parties corrupted by the ideal-world adversary $\mathcal{S}$. The functionality $\mathcal{F}_{\mathsf{APS}}$ interacts with the parties in $\mathcal{P}$ and $\mathcal{S}$.

For every $i \in [n]$, $\mathcal{F}_{\mathsf{APS}}$ receives from party P_i its additive share $\langle\mathsf{T}\rangle_i$ of the input list T and a permutation π_i.

The functionality proceeds as follows.

- Reconstruct the input list as $\mathsf{T} = \sum_{i=1}^{n} \langle\mathsf{T}\rangle_i$.
- For every $j \in [n]$, compute $\mathsf{T}_j = \pi_j(\mathsf{T})$.
- For every $j \in [n]$, generate fresh random additive shares $\{\langle\mathsf{T}_j\rangle_i\}_{i\in[n]}$ such that $\sum_{i=1}^{n} \langle\mathsf{T}_j\rangle_i = \mathsf{T}_j$.
- Send $(\mathsf{Output}, \{\langle\mathsf{T}_j\rangle_i\}_{j\in[n]})$ to party P_i for every $i \in [n]$.

Fig. 13: Ideal functionality for Amortised Permute and Share.

Protocol Π_{APS}

Inputs: Party P_i inputs its $\langle\cdot\rangle$-shares of the list T of size N and a permutation π_i.

Outputs: Additive shares of n lists $\{\langle \mathsf{T}_j \rangle\}_{j\in[n]}$, where $\mathsf{T}_j = \pi_j(\mathsf{T})$.

Preprocessing: The parties generate additive shares of a uniformly random list R of size N and, for every $j \in [n]$, additive shares of $\pi_j(\mathsf{R})$ using $\mathcal{F}_{\mathsf{Share\ Translation}}$.

Online phase:

- Parties reconstruct $\mathsf{T}+\mathsf{R}$ to all parties using the king-based reconstruction protocol.
- For every $j \in [n]$:
  - Party P_j sets $\langle \mathsf{T}_j \rangle_j = \pi_j(\mathsf{T}+\mathsf{R}) - \langle \pi_j(\mathsf{R}) \rangle_j$.
  - Every party P_i, for $i \neq j$, sets $\langle \mathsf{T}_j \rangle_i = -\langle \pi_j(\mathsf{R}) \rangle_i$.

Fig. 14: Amortised Permute and Share protocol.

Lemma F.1: The protocol Π_{APS} in Fig. 14 securely realises the functionality $\mathcal{F}_{\mathsf{APS}}$ in Fig. 13 against a static, semi-honest adversary corrupting at most $n-1$ parties, in the $(\mathcal{F}_{\mathsf{setup}}, \mathcal{F}_{\mathsf{Share\ Translation}})$-hybrid model.

Proof. We first establish correctness. Let $\mathsf{T}_j = \pi_j(\mathsf{T})$ denote the j-th output. For every $j \in [n]$, the output shares generated by Π_{APS} satisfy $\sum_{i=1}^n \langle \mathsf{T}_j \rangle_i = \pi_j(\mathsf{T}+\mathsf{R}) - \sum_{i=1}^n \langle \pi_j(\mathsf{R}) \rangle_i = \pi_j(\mathsf{T}+\mathsf{R}) - \pi_j(\mathsf{R}) = \pi_j(\mathsf{T})$. Thus, the parties obtain additive shares of the outputs prescribed by $\mathcal{F}_{\mathsf{APS}}$.

For privacy, let $C \subset \mathcal{P}$ denote the corrupted parties, with $|C| \leq n-1$. The simulator receives the corrupted parties' input shares and permutations, together with their output shares from $\mathcal{F}_{\mathsf{APS}}$. The simulator first samples a uniformly random list $\mathsf{T}^\star$, representing the reconstructed value $\mathsf{T}+\mathsf{R}$. If $P_{\mathsf{king}} \notin C$, it sends $\mathsf{T}^\star$ to the corrupted parties as the value broadcast by the honest king. If $P_{\mathsf{king}} \in C$, the simulator samples the shares of $\mathsf{T}+\mathsf{R}$ sent by all but one honest party uniformly at random and fixes the remaining honest party's share so that all shares reconstruct to $\mathsf{T}^\star$. The simulator then emulates $\mathcal{F}_{\mathsf{Share\ Translation}}$. For every $j \in [n]$ and every corrupted party $P_i \in C$, it sets $\langle \pi_j(\mathsf{R}) \rangle_i = -\langle \mathsf{T}_j \rangle_i$ if $i \neq j$. If P_j is corrupted, it instead sets $\langle \pi_j(\mathsf{R}) \rangle_j = \pi_j(\mathsf{T}^\star) - \langle \mathsf{T}_j \rangle_j$. Since R is uniformly random and independent of T, the reconstructed value $\mathsf{T}^\star = \mathsf{T}+\mathsf{R}$ is uniformly distributed and independent of T. Moreover, the programmed shares have the same distribution as the shares delivered to the corrupted parties by $\mathcal{F}_{\mathsf{Share\ Translation}}$. Hence, the simulated view is identically distributed to the real hybrid-world view. □

Simulator $\mathcal{S}_{\Pi_{\mathsf{GraSP}}}$

Assume without loss of generality that $C = \{P_1, \ldots, P_t\}$ for some $t < n$, and fix an honest party $P_{h^\star} \in \mathcal{P} \setminus C$. The simulator $\mathcal{S}_{\Pi_{\mathsf{GraSP}}}$ is given the inputs and prescribed outputs of the corrupted parties by $\mathcal{F}_{\mathsf{GraSP}}$. It maintains, throughout the simulation, the shares held by the corrupted parties for all intermediate secret-shared values.

– **Setup.** The simulator emulates $\mathcal{F}_{\mathsf{setup}}$ on behalf of the honest parties and delivers to $\mathcal{A}$ the setup information prescribed for the corrupted parties. Using the keys shared with the corrupted parties, it generates all common randomness visible to $\mathcal{A}$ according to its real-world distribution.

– **Input sharing.** For every honest party $P_i \notin C$, the simulator samples the shares of V_i and E_i held by the corrupted parties uniformly at random, according to the additive-sharing distribution.

– **Initialisation.** For every $i \in [n]$, let $B_i = \min(|\mathsf{V}|, |\mathsf{V}_i| + |\mathsf{E}_i|)$ and $L_i = B_i + |\mathsf{E}_i|$. On behalf of every honest party $P_i \notin C$, the simulator samples σ_i^G as a uniformly random permutation over $[|\mathsf{V}|]$. It additionally samples σ_i^S, σ_i^D, and σ_i^V as independent uniformly random permutations over $[L_i]$, where $L_i = B_i + |\mathsf{E}_i|$.

Message-passing computations: For $r = 1, \ldots, k$, the simulator performs the following.

– **Decompose.** The simulator emulates $\mathcal{F}_{\mathsf{APS}}$ using the current shares of V.

– **Compute.** For every $i \in [n]$, the simulator processes G_i as follows.

∘ For all linear operations in Propagate and Gather the simulator applies the prescribed local linear operations to the maintained shares of the corrupted parties. Since these operations involve no communication, no additional transcript needs to be simulated.

∘ For the transitions from vertex order to source order, source order to destination order, and destination order to vertex order, the simulator emulates $\mathcal{F}_{\mathsf{PS}}$ on the current simulated shares. It uses the corrupted parties' prescribed input shares and samples valid random inputs on the honest parties' behalf.

∘ For ApplyE and ApplyV, the simulator emulates $\mathcal{F}_{\mathsf{MPC}}$ and $\mathcal{F}_{\mathsf{mult}}$ using the current simulated shares for the corrupt party and valid random inputs on the honest parties' behalf.

– **Combine.** The simulator locally applies the prescribed extraction and concatenation operations to the maintained corrupted-party shares; since no messages are exchanged, no additional transcript needs to be simulated.

Reconstruction:

– For every corrupted recipient $P_i \in C$, let $\mathsf{V}_i^{\mathsf{out}}$ denote the output supplied by $\mathcal{F}_{\mathsf{GraSP}}$. The simulator already holds the final shares $\{\langle \mathsf{V}_i \rangle_j\}_{P_j \in C}$ of the corrupted parties.
For every honest party $P_h \notin C \cup \{P_{h^\star}\}$, the simulator samples $\langle \mathsf{V}_i \rangle_h$ uniformly at random. It then sets the share sent on behalf of $P_{h^\star}$ to $\langle \mathsf{V}_i \rangle_{h^\star} = \mathsf{V}_i^{\mathsf{out}} - \sum_{P_j \in C} \langle \mathsf{V}_i \rangle_j - \sum_{P_h \notin C \cup \{P_{h^\star}\}} \langle \mathsf{V}_i \rangle_h$.
The resulting shares reconstruct to $\mathsf{V}_i^{\mathsf{out}}$ and have the same distribution as the honest parties' reconstruction messages in the real execution.

– For reconstruction towards an honest party, the adversary receives no messages from honest parties. The corrupted parties' outgoing shares are already determined by the maintained simulated state, and therefore no additional simulation is required.

Finally, $\mathcal{S}_{\Pi_{\mathsf{GraSP}}}$ outputs the simulated view of $\mathcal{A}$.

Fig. 15: Simulator for the GraSP protocol.