emGraph: Efficient Multiparty Secure Graph Computation

Siddharth Kapoor[1], Nishat Koti[2], Varsha Bhat Kukkala[3], Arpita Patra[1], Bhavish Raj Gopal[1]
[1]*Indian Institute of Science, Bangalore*
[2]*Aztec Labs*
[3]*Indian Institute of Technology, Tirupati*

Abstract

Secure graph computation enables computing on graphs while hiding the graph topology as well as the associated node/edge data. This facilitates collaborative analysis among multiple data owners, who may only hold a private partial view of the global graph. Several works address this problem using the technique of secure multiparty computation (MPC) in the presence of 2 or 3 parties. However, when moving to the multiparty setting, as required for collaborative analysis among multiple data owners, these solutions are no longer scalable. This remains true with respect to the state-of-the-art framework of Graphiti (Koti et al., CCS 2024) as well. Specifically, Graphiti incurs a round complexity linear in the number of parties or data owners. This is due to its reliance on secure shuffle protocol, constituting a bottleneck in the multiparty setting. Additionally, Graphiti has a prohibitively expensive initialisation phase due to its reliance on secure sort, with a round complexity dependent on both the graph size and the number of parties.

We propose emGraph, a generic framework for secure graph computation in the multiparty setting that eliminates the need of shuffle and instead, relies on a weaker primitive known as Permute + Share. Further emGraph is designed to have a lightweight initialisation, that eliminates the need for sorting, making its round complexity independent of the graph size and number of parties. Unlike any of the prior works, achieving a round complexity independent of the number of parties is what makes emGraph scalable.

Finally, we implement and benchmark the performance of emGraph for the application of PageRank computation and showcase its efficiency and scalability improvements over Graphiti. Concretely, we witness improvements of up to 80× in runtime in comparison to state-of-the-art framework Graphiti. Further, we observe that emGraph takes under a minute to perform 10 iterations of PageRank computation on a graph of size 10^6 that is distributed among 25 parties/data owners, making it highly practical for secure graph computation in the multiparty setting.

1 Introduction

Graphs are fundamental for representing and analysing complex relationships in diverse real-world applications. However, in many applications these graphs contain sensitive user information and are usually distributed across multiple data owners. In such scenarios, performing computation while hiding the graph topology and the associated sensitive data is a challenge. For example, consider a financial transaction graph where 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 analysing the global transaction graph for various reasons, such as identifying fraudulent transactions, credit risk analysis, etc. However, due to privacy constraints, and regulations and commercial interests, banks cannot simply share this information with each other. Thus, there is a need for designing techniques that allow data owners to collaboratively perform computation on sensitive and distributed graph data while ensuring privacy.

Secure graph computation [1,10,13,16,17,20] has emerged as a powerful approach to address this challenge. It allows realising diverse applications, including social network analysis, recommendation systems [13], contact tracing [1, 10, 15], transaction network analysis [8, 9, 17], graph neural network training and inference [8, 20] etc, while ensuring privacy. These works rely on the cryptographic technique of secure multiparty computation (MPC). This technique allows a set of n parties to compute a function on their private inputs such that an adversary controlling any $t < n$ parties cannot learn any information beyond the output. In this context of secure graph computation via MPC, data owners can enact the role of parties, with their local views of the graph as private inputs, and the graph computation corresponds to the function. Although, MPC can guarantee the privacy of graph data during computation, it introduces overheads in terms of rounds (number of sequential interactions among the computing parties),

communication (volume of the messages exchanged between parties), and computation (the local operations performed by each party). For large-scale real-world graphs, these overheads can significantly impact practicality.

To design efficient solutions, existing generic graph computation frameworks [1, 10, 13] leverage the sparsity of real-world graphs. They 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. Graphiti [10] forms the state-of-the-art MPC framework for securely realising message-passing algorithms. Leveraging the fact that many widely used graph algorithms rely on linear aggregation functions (in step (ii)), Graphiti achieves round complexity independent of the graph size, unlike the prior works, making it highly efficient. Even for applications that require non-linear aggregation, Graphiti outperforms prior MPC solutions [1, 13].

Graphiti is designed primarily for the outsourced computation setting. In this setting, data owners do not directly 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 the 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 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 enact the role of MPC parties, retaining complete control over their data. Specifically, an honest data owner's privacy is guaranteed even if all other data owners collude. Thus, non-outsourced setting may be more desirable in certain scenarios when there are multiple data owners.

To account for the aforementioned scenarios, Graphiti can be instantiated in the non-outsourced setting by using a generic MPC protocol that supports arbitrary number of parties. However, unlike the outsourced setting, which benefits from customised small-party computation protocols, generic MPC protocols incur higher overheads in terms of both rounds and communication, thereby affecting scalability. Specifically, in the non-outsourced multiparty scenario, Graphiti suffers from two main limitations—(i) it requires an expensive initialisation phase that incurs a round complexity that is linearly dependent on both the graph size and the number of parties; additionally, communication complexity is log-linear in graph size, (ii) each iteration of message-passing incurs a round complexity that is linearly dependent on the number of parties. A higher round and communication complexity is a critical bottleneck, as it directly impacts the runtime of the protocol. For instance, parties could be geographically distributed over a WAN. Here, the high round trip time (RTT) and low bandwidth amongst them, coupled with the higher round and communication complexity of the MPC protocols, can result in an impractical run time for securely performing graph computation. This motivates us to ask the question:

"Can we design a framework for secure graph computation in the multiparty setting with round complexity independent of both the graph size and the number of parties while minimising the communication complexity?"

We answer this question affirmatively and propose a novel framework, emGraph. To put our solution in perspective, we note the following. Keeping efficiency at center stage when designing secure graph computation frameworks, the first observation is to operate on a list-based representation of the graph that allows evaluating graph algorithms as message-passing algorithms [13] rather than operating on matrices. However, since the list is required to be reordered multiple times in each message-passing iteration, this requires relying on expensive secure sort operation. When performed across several iterations, this forms a bottleneck and significantly impacts overall efficiency. To overcome this, [1] observed that the need for secure sort across iterations can be replaced by that of a secure shuffle protocol that is known to be more efficient. However, [1] still requires one invocation to a secure sort protocol to be performed as part of a one-time initialisation, which continues to constitute a major efficiency bottleneck. Despite improving over [1], the work of [10] continues to suffer from the same efficiency bottleneck. In this regard, emGraph completely eliminates the need for the secure sort protocol. Moreover, the need for secure shuffle is also circumvented and instead replaced by a much more efficient primitive of Permute + Share. Eliminating the secure sort and shuffle operations allows emGraph to witness tremendous efficiency improvements and significantly improve its scalability. We outline our key contributions next.

1.1 Contributions

We design a generic and efficient end-to-end framework, emGraph, for secure graph computation in the multiparty setting. emGraph enables the secure realisation of any message-passing graph algorithm when the graph (and its associated data) is distributively held by multiple data owners. emGraph operates in the preprocessing paradigm which allows to offload input-independent MPC computations to a preprocessing phase to pave the way for a fast input-dependent online phase. 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. Below, we highlight our contributions:

- **Novel design paradigm of** emGraph**:** emGraph leverages the fact that data owners themselves participate in the protocol and have direct access to their input data. Specifically, each party or data owner has in its local view, a subgraph of the global graph $\mathcal{G}$, which includes the vertices it owns, their neighbouring vertices, and the edges between these. emGraph leverages this local information to design a new and efficient Decompose-Compute-Combine (DCC) paradigm for secure message-passing on the global graph $\mathcal{G}$ by: (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) computing multiple iterations of the message-passing algorithm on the subgraphs $\mathcal{G}_i$ in parallel, instead of doing the same on the global graph $\mathcal{G}$, (iii) combining the results on these subgraphs to realise message-passing on the global graph $\mathcal{G}$. Steps (i)-(iii) are designed to leverage knowledge of $\mathcal{G}_i$'s topology by P_i such that the round complexity is made independent of the number of parties[1]. In this way, we design emGraph to scale efficiently in the multiparty setting. Similar to Graphiti, these computations are performed in two phases: a one-time initialisation phase, and an iterative message-passing phase. To improve the efficiency of these phases, we introduce novel techniques. In doing so, we observe that emGraph outperforms Graphiti even in the small-party setting, as discussed next, with the improvements reported in Table 1.
- **Lightweight initialisation phase:** Computations that remain consistent across iterations of message-passing are moved into a one-time initialisation phase. This initialisation phase is prohibitively expensive in Graphiti, as it requires secure sorting operations. Specifically, it incurs a round complexity dependent on the number of parties and the graph size, i.e., $O(n\log(\mathsf{N}))$, where N denotes the total number of vertices and edges in $\mathcal{G}$. Further, it has a communication complexity that is log-linear in the graph size, i.e. $O(n\mathsf{N}\log(\mathsf{N}))$. In contrast, leveraging the fact that the topology of each subgraph is known to a distinct party, we design a lightweight initialisation phase for emGraph that completely eliminates the need for secure sorting. Consequently, our initialisation phase achieves constant round complexity, independent of the graph size and number of parties. Further, it achieves an improved communication complexity of $O(n|\mathsf{E}|)$, linear in both the number of parties and the graph size.
- **Improved message-passing phase:** The goal of this phase is to iteratively update the state of the vertices in $\mathcal{G}$. In Graphiti, each iteration incurs a round complexity that is dependent on the number of parties, n. This dependence on n arises from the reliance on a secure shuffle protocol, which allows secret shares of a list (in this case, the list representation of $\mathcal{G}$) to be reordered based on a random permutation not known to any party. This protocol incurs $O(n)$ rounds in the multiparty setting [7]. In contrast, in our approach, message-passing happens in the subgraphs whose topology is known to one of the parties. Thus, leveraging this information, we modify and enhance each iteration of message-passing to only rely on a weaker primitive known as Permute + Share [3, 7], instead of shuffle. Permute + Share allows secret shares of a list to be reordered based on a permutation known to one party and can be realised in $O(1)$ rounds, even in multiparty setting.

To give a more detailed characterisation of the round complexity, note that both, Graphiti and emGraph realise each iteration of message-passing through the primitive operations of Propagate, ApplyE, Gather and ApplyV. Informally, Propagate transfers data from the source vertex to its outgoing edges. ApplyE updates the data on the edges based on a user-defined function f_{AE}. Gather aggregates the data received on the incoming edges of a vertex, and ApplyV updates the vertex's state based on the aggregated data from Gather as per a user-defined function f_{AV}. The secure realisation of these primitives enables the secure realisation of any message-passing graph algorithm. In Graphiti, this incurs a round complexity of $O(n + \mathsf{r}_{\mathsf{AE}} + \mathsf{r}_{\mathsf{AV}})$, where r_{AE} and r_{AV} denote the round complexity of securely computing f_{AE} and f_{AV}. On the other hand, emGraph incurs round complexity independent of number of parties, i.e., $O(\mathsf{r}_{\mathsf{AE}} + \mathsf{r}_{\mathsf{AV}})$ while maintaining the same communication as that of Graphiti.

Phase	Ref	Rounds	Communication
Initialisation	Graphiti [10]	$O(n \cdot \log(\mathsf{N}))$	$O(n \cdot \mathsf{N} \cdot \log(\mathsf{N}))$
	emGraph	$O(1)$	$O(n \cdot \lvert\mathcal{E}\rvert)$
Message-passing	Graphiti [10]	$O(n + \mathsf{r}_{\mathsf{AE}} + \mathsf{r}_{\mathsf{AV}})$	$O(n \cdot \mathsf{N} + (\mathsf{c}_{\mathsf{AE}} + \mathsf{c}_{\mathsf{AV}}) \cdot \mathsf{N})$
	emGraph	$O(\mathsf{r}_{\mathsf{AE}} + \mathsf{r}_{\mathsf{AV}})$	$O(n \cdot \mathsf{N} + (\mathsf{c}_{\mathsf{AE}} + \mathsf{c}_{\mathsf{AV}}) \cdot \lvert\mathcal{E}\rvert)$

Given graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, $|\mathcal{V}|$ denotes number of vertices, $|\mathcal{E}|$ denotes number of edges and $\mathsf{N} = |\mathcal{V}| + |\mathcal{E}|$.
r_x and c_x denote the round and communication complexity of function f_x when securely realised via MPC. Here, AE, AV denote the functions to be applied as part of ApplyE, ApplyV, respectively.

Table 1: Comparison of the initialisation and one iteration of message-passing.

- **Amortised Permute + Share:** Naively using the Permute + Share primitive in our framework results in round complexity that is independent of n, but it introduces a significant communication overhead that is quadratic in n, i.e., $O(n^2(\mathsf{N}))$. This is because the naive approach requires permuting the same input list according to n different permutations, each known to a distinct party. This requires n parallel invocations of Permute + Share, where each invocation has a communication complexity $O(n(\mathsf{N}))$. Instead, by leveraging the fact that the input list remains the same across all n invocations, we design an Amortised Permute + Share protocol that significantly reduces the online complexity. This protocol achieves the same online communication complexity as

[1] While emGraph is designed in preprocessing model, its round complexity remains independent of the number of parties, even when operating in an all-online setting.

that of a single invocation of the Permute + Share protocol while maintaining constant round complexity. Elaborately, the Amortised Permute + 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 + Share protocol can be of independent interest, as it enables multiple permutations to be performed on a secret-shared list efficiently.

- **Benchmarks:** We implement and benchmark emGraph to demonstrate its practicality and improvements over Graphiti. Specifically, we compare the initialisation phase of emGraph with that of Graphiti while varying graph size ($N = |\mathcal{V}| + |\mathcal{E}|$) and number of parties. We observe improvements of 2200× in runtime and 270× in communication for a graph of size 10^5 distributed among 15 parties. Similarly, we compare the message-passing phase of emGraph and Graphiti across varying numbers of parties and observe a runtime improvement of 11× for $n = 25$ parties on a graph of size 10^5. When considering the total runtime for 10 iterations of PageRank computation, emGraph achieves an 80× improvement in runtime and a 106× improvement in communication compared to Graphiti for a graph of size 10^5 and 15 parties. Moreover, we note that emGraph is highly scalable and takes under a minute to complete PageRank computation on a graph of size 10^6 distributed among 25 parties.

Organisation: §2 discusses the related works that design generic secure graph computation frameworks. Preliminaries are discussed in §3. Our Decompose-Compute-Combine (DCC) approach for realising message-passing graph algorithms is discussed in §4. This is followed by §5 where the end-to-end framework of emGraph is detailed. Finally, benchmark results are provided in §6, followed by the conclusion in §7. Additional details such as the framework of Graphiti, additional experimental results, security proofs, etc. appear in Appendices §A-§D.

2 Related works

Secure graph computation has gained significant attention due to its applicability in privacy-sensitive scenarios. Here, we discuss some of the relevant works that look at designing generic secure graph computation frameworks. The work of [13] was the first to propose a generic framework for secure graph computation. Leveraging the sparsity in real-world networks, [13] introduced the GraphSC framework, that operates on a list representation of the graph rather than its adjacency matrix. This design choice reduces computational complexity from $O(|\mathcal{V}|^2)$ to $O(|\mathcal{V}| + |\mathcal{E}|)$, where $|\mathcal{V}|$ and $|\mathcal{E}|$ denote the number of vertices and edges, respectively. At a high level, GraphSC realises message passing using the primitives of Scatter and Gather. Informally, Scatter involves propagating data from vertices to their outgoing edges, while Gather aggregates data from incoming edges to update the state of vertices. GraphSC relies on garbled circuits (GC) [19] for the secure realisation of Scatter and Gather. While this approach ensures constant round complexity, it incurs prohibitively high communication and computation costs.

Towards this, [1] proposed a secret-sharing-based (SS-MPC) realisation of GraphSC. SS-MPC protocols are known for their reduced communication costs compared to GC but suffer from round complexity proportional to the circuit's multiplicative depth. For a naive realisation of Scatter and Gather primitives, this depth scales linearly with the graph size ($|\mathcal{V}| + |\mathcal{E}|$). [1] introduces a round-optimised (RO) solution to improve this and achieves a round complexity of $O(\log(|\mathcal{V}| + |\mathcal{E}|))$ for message passing at the expense of increased communication overhead. Building upon [1], the work of [10] designs Graphiti by introducing novel algorithms for realising Scatter and Gather with round complexity independent of the graph size, significantly improving efficiency. Specifically, it leverages the fact that several widely used real-world graph algorithms rely on linear aggregation operations during Gather to design a solution to realise message passing in rounds independent of graph size. However, Graphiti is tailored primarily for the outsourced computation setting that leverages customised small-party MPC protocols. When instantiated with generic MPC protocols, it incurs a round complexity that scales linearly with the number of parties. Further, it suffers from an expensive initialisation phase that scales linearly with a number of parties and graph size, thereby hindering scalability in the multiparty setting.

To address secure graph computation in generic multiparty settings, [20] proposed alternative approach. [20] considers secure training and inference of graph neural networks (GNNs) in a multi-party setting. While their framework can theoretically support any message-passing graph algorithm, it assumes a highly restrictive security model where no two parties participating in the protocol are allowed to collude. By adopting this weaker security model, [20] aims to optimise efficiency but still suffers from a round complexity dependent on graph size. Moreover, it leaks information about the number of edges between parties, which may reveal sensitive details about the graph topology.

3 Preliminaries

Threat model: For the non-outsourced scenario we consider, we rely on a general-purpose multiparty computation protocol in the semi-honest setting. Let $\mathcal{P} = \{P_1, \ldots, P_n\}$ denote the set of n parties, each connected via pairwise private and authenticated channels in a synchronous network. We assume 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. Our framework operates over a ring algebraic structure, with $\mathbb{Z}_{2^\ell}$ denoting the ring of ℓ-bit elements.

We rely on additive secret sharing, where a value $x \in \mathbb{Z}_{2^\ell}$

is said to be $\langle\cdot\rangle$-shared (or additively shared) over $\mathbb{Z}_{2^\ell}$ if each party $P_i \in \mathcal{P}$ holds a 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, i.e., 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 $\mathsf{c}_1\mathsf{x} + \mathsf{c}_2\mathsf{y}$. To enable secure computation over shares, we rely on a one-time key setup [2, 4, 11, 12, 14] 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 v as described in §A. However, the reconstruction of a $\langle\cdot\rangle$-shared value towards a party $P_i \in \mathcal{P}$, requires interaction where every party to communicates its share to P_i.

Depending on the graph algorithm to be computed, emGraph may may require additional MPC primitives such as multiplication, secure comparison, equality, etc. These are abstracted via the $\mathcal{F}_{\mathsf{MPC}}$ functionality, which takes $\langle\cdot\rangle$-shared inputs from the parties and outputs the $\langle\cdot\rangle$-shares of the computed function result. In this work, we instantiate $\mathcal{F}_{\mathsf{MPC}}$ using the protocols of [5] in the semi-honest setting. Finally, emGraph also relies Permute + Share protocol of [3,7], which takes a secret-shared list T and a permutation π_i known only to a designated party P_i. It outputs $\langle \mathsf{T_O}\rangle$ such that $\mathsf{T_O} = \pi_i(\mathsf{T})$. Further details of Permute + Share appears ahead.

System model: We consider a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ 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 $\mathcal{V}_i$, such that $\mathcal{V} = \bigcup_{i=1}^{n} \mathcal{V}_i$. Each vertex $\mathsf{v} \in \mathcal{V}$ 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, $\mathcal{V}$ is known to all the parties. Each directed edge $\mathsf{e} \in \mathcal{E}$ consists of $(\mathsf{e.src}, \mathsf{e.dst})$, representing the source and destination label, respectively. Information about existence of edges e such that both $\mathsf{e.src}, \mathsf{e.dst} \in \mathcal{V}_i$, is known only to P_i. For edges where $\mathsf{e.src} \in \mathcal{V}_i$ and $\mathsf{e.dst} \in \mathcal{V}_j$ with $i \neq j$, the existence of e is known to both P_i and P_j. We let $\mathcal{E}_i$ represent the set of all edges $\mathsf{e} \in \mathcal{E}$ where $\mathsf{e.dst} \in \mathcal{V}_i$, and have $\mathcal{E} = \bigcup_{i=1}^{n} \mathcal{E}_i$. Each vertex and edge may also be associated with data components. The data component of a vertex $\mathsf{v} \in \mathcal{V}$, denoted as $\mathsf{v.data}$, is private to the party that owns v. Similarly, the data component of an edge $\mathsf{e} \in \mathcal{E}$, is denoted as $\mathsf{e.data}$. For edges where both $\mathsf{e.src}, \mathsf{e.dst} \in \mathcal{V}_i$, the data component is private only to P_i. For edges where $\mathsf{e.src} \in \mathcal{V}_i$ and $\mathsf{e.dst} \in \mathcal{V}_j$ with $i \neq j$, the data is known to both P_i and P_j.

Secure Shuffle and Permute+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 among all parties in $\mathcal{P}$, such that each entry of T is shared across the parties. 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 a weaker building block called Permute + Share, as described in [3, 7]. The Permute + Share protocol allows shuffling a secret-shared list using a permutation known to one party. Specifically, the Permute + 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 $\mathsf{T_O} = \pi_i(\mathsf{T})$. Secure shuffle can then be realised by performing n sequential invocations of Permute + Share, where each party contributes a random permutation ensuring that the final permutation applied is a composition of all permutations that is not known to any party.

The Permute + Share protocol described in [3] consists of two phases: preprocessing and online phase. During the preprocessing phase, the parties generate additive secret shares of a random list R 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 [3, 7]. Each 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 $\mathsf{T_O} = \pi_i(\mathsf{T})$. The online phase requires one round, with a cost of $O(n\mathsf{N})$.

For secure shuffle, a naive approach involves n sequential invocations of the Permute + Share protocol, with each party contributing one random permutation. 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 naive realisation by optimising the online phase communication to $O(n\mathsf{N})$ while maintaining the same number of rounds and similar preprocessing costs. We refer interested readers to [7] for further details of the optimised protocol.

Graphiti framework: Graphiti is a generic framework for the secure realisation of message-passing graph algorithms. In Graphiti, the input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is represented as a data-augmented graph (DAG) list or DAG-list G which includes an entry for each vertex and edge in $\mathcal{G}$. Each entry consists of a tuple $(\mathsf{src}, \mathsf{dst}, \mathsf{isV}, \mathsf{data})$ of four components: source (src), destination (dst), is_Vertex (isV) to distinguish entries of a vertex from an edge, and data to store various data items associated with the vertex/edge. Specifically, a node u is represented as $(\mathsf{u}, \mathsf{u}, 1, \mathsf{data})$, and a directed edge $\mathsf{e}(\mathsf{u}, \mathsf{v})$ is represented as $(\mathsf{u}, \mathsf{v}, 0, \mathsf{data})$. The data field can contain multiple sub-components, some specific to vertices and others specific to edges. To ensure the topology of the graph is hidden, every entry in the DAG-list includes all sub-components, with irrelevant ones set to 0 or dummy values. Undirected graphs can also be represented as DAG-list by accounting for every edge twice in both directions.

To perform the required operations across multiple iterations of a message-passing algorithm, the framework relies on the following set of primitives: (i) Propagate: enables vertices to propagate data along their outgoing edges, (ii) ApplyE: updates the data of all the edges based on a user-defined function f_{AE}, (iii) Gather: enables vertices to aggregate data from its incoming edges, and (iv) ApplyV: updates all vertices based on a user-defined function f_{AV}[2]. Secure realisation of these primitives on a secret-shared DAG-list enables the secure realisation of message-passing. To efficiently realise these primitives in MPC, the framework relies on three different orderings of the DAG-list. In the vertex ordering, all vertex entries in the list appear first, followed by all edge entries. In the source ordering, every vertex entry in the list appears immediately before entries of all its outgoing edges. Similarly, in the destination ordering, entries of all the incoming edges of a vertex appear immediately before the entry of the corresponding vertex. Using these different orderings, each iteration of the message-passing algorithm is achieved as follows. An iteration begins by invoking Propagate that entails performing computation on the vertex ordered DAG-list, followed by computations in the source ordered DAG-list. Hence, this requires an intermediate transition from vertex ordering to source ordering. ApplyE is then invoked to ensure each edge in the DAG-list is updated under the function f_{AE} and can be performed independent of any ordering. This is followed by invoking Gather, which entails performing computations over destination order, followed by computations in the vertex ordered DAG-list. Thus, Gather requires transitioning from source ordering to destination order as well as a transition back to vertex ordering. Finally, an invocation to ApplyV ensures data on each node is updated under the function f_{AV}. Similar to ApplyE, this can be performed in any ordering.

In the secure realisation of Propagate (Algorithm 1) and Gather (Algorithm 2), the computations performed on the DAG-list consist of linear operations and hence can be realised non-interactively via MPC. The only interactive operation involved is the transition of the DAG-list between different orderings. On the other hand, ApplyE and ApplyV do not require any transitions, but the computations performed are dependent on the functions $f_{\mathsf{AE}}, f_{\mathsf{AV}}$, respectively. We briefly outline how Graphiti realises these transitions between different orderings efficiently. Rather than relying on a secure sort to transition between the different orderings, Graphiti relies on a *shuffle-then-sort* paradigm outlined in [1]. The idea is to first randomly order the entries of the list via a secure shuffle. This is followed by performing a secure sort to transition to the required vertex/source/destination ordering. However, the mapping (permutation) of the elements from the random order to the required ordering can be made public. Further, these transitions remain the same across multiple iterations of the message-passing algorithm. Thus, Graphiti relies on a one-time initialisation phase to generate these public permutations via secure sort (see Fig. 6). This approach ensures that transitions during the message passing phase can be done via secure shuffle followed by applying a public permutation which is much more efficient than a secure sort. The formal protocols for each of the primitives, the details of initialisation and a pictorial representation of the overall process during each iteration of the message passing phase in Graphiti are provided in A.2. We point an interested reader to the same.

Complexity of Graphiti*:* The initialisation phase of Graphiti assumes the input DAG-list is provided in vertex ordering and involves two sequential invocations of secure shuffle followed by two parallel invocations of secure sort, required to generate mappings corresponding to source and destination order. When instantiated in the multiparty setting with n parties, secure shuffle[3] [7] incurs a round complexity of $O(n)$ and communication of $O(n(\mathsf{N}))$, where $\mathsf{N} = |\mathsf{V}| + |\mathsf{E}|$. Similarly, secure sort incurs $O(n(\log(\mathsf{N})))$ rounds and $O(n\mathsf{N}\log(\mathsf{N}))$ communication. Thus overall, the initialisation phase incurs $O(n\log(\mathsf{N}))$ rounds and $O(n\mathsf{N}\log(\mathsf{N}))$ communication. For each iteration of message passing, secure evaluation of Propagate and Gather using MPC incurs $O(n)$ rounds and $O(n\mathsf{N})$ communication. ApplyE is evaluated by securely computing the function f_{AE} on the appropriate data components of each entry of the DAG-list in parallel. Let r_{AE} and c_{AE} denote the round and communication complexity of computing f_{AE} securely. Then, ApplyE incurs $O(\mathsf{r}_{\mathsf{AE}})$ rounds and $O(\mathsf{c}_{\mathsf{AE}} \cdot \mathsf{N})$ communication. Similarly, ApplyV is evaluated by securely computing the function f_{AV} on the appropriate data components of the first $|\mathsf{V}|$ entries in the DAG-list in parallel. Let r_{AV} and c_{AV} denote the round and communication complexity of computing f_{AV} securely. Then, ApplyE incurs $O(\mathsf{r}_{\mathsf{AV}})$ rounds and $O(\mathsf{c}_{\mathsf{AV}} \cdot |\mathsf{V}|)$ communication. Thus, each message passing iteration incurs a cost of $O(n + \mathsf{r}_{\mathsf{AE}} + \mathsf{r}_{\mathsf{AV}})$ rounds and $O(n\mathsf{N} + \mathsf{c}_{\mathsf{AE}} \cdot \mathsf{N} + \mathsf{c}_{\mathsf{AV}} \cdot |\mathsf{V}|)$ communication[4].

4 DCC approach for message passing

In this section, we describe our new Decompose, Compute and Combine (DCC) paradigm for message-passing on graphs that allows attaining an efficient secure graph computation framework in the multiparty setting. Consider a single iteration of traditional message-passing on a graph $\mathcal{G}$, as done in Graphiti, where $\mathcal{G}$ (and the associated data components) is distributively held by the n parties. It consists of performing—(i) Propagate, (ii) ApplyE, (iii) Gather, and (iv) ApplyV. Observe that the state update of a vertex depends only on the data component of its immediate neighbours and the correspond-

[2]The framework allows adapting the primitives for edges in any direction.

[3]While secure shuffle can be realised in $O(1)$ rounds by applying a secret-shared permutation matrix, this approach incurs a communication and computation cost of $O(\mathsf{N}^2)$ which is prohibitive for a sparse graph.

[4]In case of applications that require non-linear aggregation, Graphiti incurs $O(n\log(\mathsf{N}) + \mathsf{r}_{\mathsf{AE}} + \mathsf{r}_{\mathsf{AV}})$ rounds and $O(n\mathsf{N}\log(\mathsf{N}) + \mathsf{c}_{\mathsf{AE}} \cdot \mathsf{N} + \mathsf{c}_{\mathsf{AV}} \cdot |\mathsf{V}|)$ communication. See [1, 10] for further details.

ing incoming edges. Therefore, for all vertices $\mathcal{V}_i$ owned by a party P_i, their state update depends on the neighbours of $\mathcal{V}_i$ and the edges with destination in $\mathcal{V}_i$. Consequently, to update the state of all vertices in $\mathcal{V}$, it suffices to independently update the state of vertices in $\mathcal{V}_i$ for each $i \in [n]$, and then combine these updates to get the updated state of $\mathcal{V}$. To facilitate this, we define subgraphs $\mathcal{G}_i$ for $i \in [n]$, consisting of all vertices $\mathcal{V}$ and the edges with destination in $\mathcal{V}_i$ (denoted by $\mathcal{E}_i$). Note that these subgraphs, $\mathcal{G}_i = (\mathcal{V}, \mathcal{E}_i)$[5] for $i \in [n]$, are a proper subset of $\mathcal{G}$. 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 to correctly update the state of vertices in $\mathcal{V}_i$.

Thus, realising one iteration of message passing on $\mathcal{G}$ via 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 on each subgraph $\mathcal{G}_i$ for $i \in [n]$ to update the states of $\mathcal{V}_i$.
3. Combine: Extracting the updated state of $\mathcal{V}_i$ from $\mathcal{G}_i$ for $i \in [n]$ and combining them to get the updated $\mathcal{V}$.

Multiple iterations of message-passing can be realised by repeatedly performing steps 1-3, where $\mathcal{G}$ updated in step 3 of the current iteration is used in step 1 of the next iteration to freshly generate the subgraphs. In this way, our DCC approach correctly realises message-passing on $\mathcal{G}$ as done in Graphiti. When securely realising our DCC approach, since P_i knows the topology of the subgraph $\mathcal{G}_i$, secure computations on $\mathcal{G}_i$ (step 2) can be achieved in rounds independent of the number of parties. Moreover, efficient secure protocols to realise subgraph generation (step 1) and updating $\mathcal{G}$ (step 3) ensure the secure realisation of message-passing in rounds independent of number of parties. An illustrative example of DCC approach for message-passing appears in Fig. 1.

In the subsequent sections, we first describe how to securely realise our new approach for message-passing in round complexity independent of n but with quadratic communication. Following this, we describe how to reduce communication cost to linear in n without hampering the round complexity.

4.1 Secure realisation of DCC

Input representation: In our solution, the input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ (see §3) is represented as a tuple of lists $\mathsf{G} = (\mathsf{V}, \{\mathsf{E}_i\}_{i\in[n]})$, referred to as the data augmented graph list or DAG list. Here, G additionally accounts for the data components of the vertices and edges present in $\mathcal{G}$. Elaborately, G consists of a list of vertices, denoted as V, comprising all vertices in $\mathcal{V}$ together with the associated data. Further, V is ordered as $\mathsf{V}_1||\mathsf{V}_2||\ldots||\mathsf{V}_n$, where V_i denotes all the vertices in $\mathcal{V}$ belonging to P_i along with their data components and $||$ denotes concatenation. G also comprises n edge lists E_i for $i \in [n]$, that represents the subset of edges in $\mathcal{E}$ that have destination in $\mathcal{V}_i$ (vertices owned by P_i) together with the associated data. Each entry in V and E_i consists of four components: src and dst (to denote the source and destination of an edge, while src = dst = v.id for a vertex v), isV (set to 1 for vertices and 0 for edges) and a data component. At the start of the computation, during an input sharing phase (described in §5), 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. Thus, our framework assumes $|\mathsf{E}_i|$, i.e. number of incoming edges terminating at vertices in V_i, to be public. However, $|\mathsf{E}_i|$ does not reveal sensitive information about the graph's topology or the number of edges between specific pairs of parties, as leaked in [17, 20][6].

Each iteration of message-passing begins with the secret-shared DAG list $\langle\mathsf{G}\rangle = (\langle\mathsf{V}\rangle, \{\langle\mathsf{E}_i\rangle\}_{i\in[n]})$ and outputs updated secret shares $\langle\mathsf{G}\rangle$, where the data components of the vertices are updated based on operations performed during the iteration of message-passing. 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 (see §3). 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 simply set $\langle\mathsf{G}_i\rangle$ as $\langle\mathsf{G}_i\rangle = \langle\mathsf{V}\rangle \,||\, \langle\mathsf{E}_i\rangle$, where $||$ denotes concatenation operation.

Compute: This step entails performing message-passing computations on the DAG-lists. Specifically, it takes as input the secret shares of DAG-list G_i for $i \in [n]$ and outputs its updated shares, where the data component of the vertices is updated. For ease of explanation, we discuss the message-passing computations of a single DAG-list G_i. Similar computations are performed on all the other DAG-lists in parallel.

Naive approach: Given $\langle\mathsf{G}_i\rangle$, the parties can naively invoke the message-passing computations of Graphiti on it. This involves (see §3) the secure realisation of the Propagate, ApplyE, Gather, and ApplyV on $\langle\mathsf{G}_i\rangle$. Propagate involves non-interactive computations in vertex order of $\langle\mathsf{G}_i\rangle$, followed by transitioning to source order of $\langle\mathsf{G}_i\rangle$. Gather involves non-interactive computations in destination order of $\langle\mathsf{G}_i\rangle$ followed by a transition back to vertex order of $\langle\mathsf{G}_i\rangle$. An additional intermediate transition from source order to destination order of $\langle\mathsf{G}_i\rangle$ is required between Propagate and Gather. Each of

[5] When defining $\mathcal{G}_i$, note that it is easy to discard unnecessary edges that do not aid in the state update of vertices owned by P_i. However, discarding unnecessary vertices introduces several challenges when designing decompose and combine phases. Hence, for ease of explanation, we first discuss our approach when $\mathcal{G}_i$ is defined to comprise the entire vertex set $\mathcal{V}$, and subsequently handle the case where $\mathcal{G}_i$ is defined to only comprise $\mathcal{V}_i$.

[6] To mitigate any potential leakage, a practical alternative is to disclose a public upper bound on $|\mathsf{E}_i|$ rather than the exact value and pad E_i for $i \in [n]$ with dummy edges.

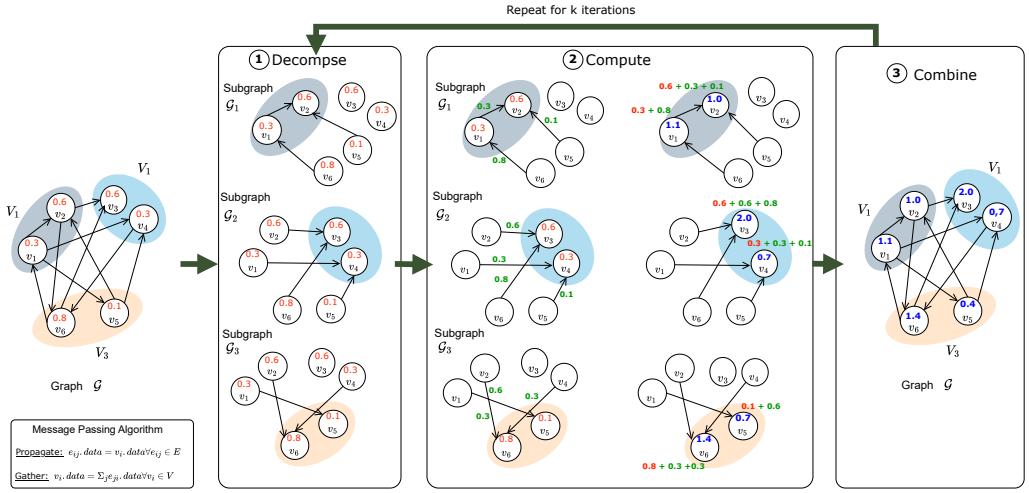

Figure 1: Example of message-passing via DCC when a graph $\mathcal{G}$ is decomposed into $\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3$ which is in the view of parties P_1, P_2, P_3, respectively. For simplicity, we consider the ApplyE and ApplyV functions to be identity functions and the aggregation operation in Gather to be addition. Fig. 7 illustrates the traditional approach for message passing on $\mathcal{G}$ without DCC

these transitions required for Propagate, Gather, and the intermediate step incur a round complexity of $O(n)$ due to the reliance on secure shuffle. Further, ApplyE and ApplyV involve applying the respective functions f_{AE} and f_{AV} to each entry in the DAG-list, regardless of the order of $\langle \mathsf{G}_i \rangle$. Specifically, let r_{AE} and r_{AV} be the round complexity required to securely apply the function f_{AE} and f_{AV} during ApplyE and ApplyV, respectively. Then ApplyE and ApplyV incur a round complexity of $O(\mathsf{r}_{\mathsf{AE}})$ and $O(\mathsf{r}_{\mathsf{AV}})$, independent of n. Thus, the overall round complexity of this approach is $O(n + \mathsf{r}_{\mathsf{AE}} + \mathsf{r}_{\mathsf{AV}})$, which scales linearly with n, constituting a bottleneck. However, we observe that the topological information of G_i, known to party P_i, can be leveraged to improve the round complexity of these transitions to be independent of n. This is described next.

Enhanced approach: We now describe how the transitions between the various orderings of G_i can be realised efficiently, with round complexity independent of n. We detail the steps involved for transitioning from vertex to source order. Similar steps can be performed for the other transitions as well. Recall that for the DAG-list G_i, party P_i is fully aware of its topology and thereby the labels of vertices and edges in the G_i. This allows P_i to define a mapping/permutation (say π_i^S) that reorders the DAG-list G_i present in vertex order to source order. This permutation is then securely applied to $\langle \mathsf{G}_i \rangle$ to generate its source order. Since π_i^S cannot be provided in clear to other parties as it may leak topology information about G_i, the reordering is performed as follows.

Parties first permute $\langle \mathsf{G}_i \rangle$ using a random permutation $\hat{\pi}_i^S$ known to P_i. This reorders $\langle \mathsf{G}_i \rangle$ to a random order. The random permutation is applied securely using the Permute + Share protocol, which enables permuting a secret-shared list based on a permutation known to one party and incurs a round complexity of $O(1)$. Following, this P_i, computes and sends the public permutation $\sigma_i^S = \pi_i^S \circ \hat{\pi}_i^{S^{-1}}$ that

Figure 2: Transition from vertex to source order of G_i.

maps the randomly ordered DAG-list $\langle \mathsf{G}_i \rangle$ to source order. Since σ_i^S maps a random order of $\langle \mathsf{G}_i \rangle$ to its source order, it does not reveal any information about the topology of G_i. The permutation σ_i^S is then applied by all parties, non-interactively, on their local shares $\langle \mathsf{G}_i \rangle$ in the random order to generate the source order $\langle \mathsf{G}_i \rangle$. A pictorial representation of the transition from vertex order to source order appears in Fig. 2. Further, observe that this mapping from vertex order to source order remains the same across multiple iterations of message-passing phase. Hence, the same random permutation can be used in every iteration. Thus, the public permutation σ_i^S can be computed once and sent during a one-time initialisation phase. This concretely reduces the overhead for every transition by one round. In this way, all the transitions required during message-passing can be realised in $O(1)$ rounds. Thus, the overall round complexity of this approach is $O(\mathsf{r}_{\mathsf{AE}} + \mathsf{r}_{\mathsf{AV}}) = O(1)$(for transitions in Propagate , Gather and the intermediate transition) + $O(\mathsf{r}_{\mathsf{AE}})$ (for ApplyE) + $O(\mathsf{r}_{\mathsf{AV}})$ (for ApplyV). Pictorial representation of the enhanced approach for performing message-passing computations on DAG-list G_2 in compute appears in Fig. 3.

Combine: This step takes as input secret shares of DAG-lists G_i for $i \in [n]$ and outputs and 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

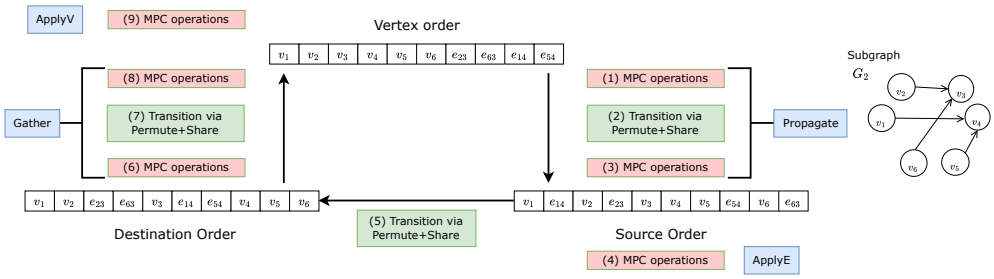

Figure 3: Enhanced approach for performing message passing computations on G_2 whose topology is known to P_2.

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 of communication. 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. This updated $\langle \mathsf{G} \rangle$ serves as the input for the next iteration of DCC.

Complexity of the solution: Each iteration of message passing involves three main steps. First, 'decompose' where the generation of G_i is achieved non-interactively and incurs no additional communication or rounds. Second, 'compute' where message-passing on each G_i is performed securely by realising the operations Propagate, ApplyE, Gather, and ApplyV. For a G_i of size $O(|\mathsf{V}|+|\mathsf{E}_i|)$, Propagate, Gather and the intermediate transition incur $O(1)$ rounds and communication of $O(n(|\mathsf{V}|+|\mathsf{E}_i|))$. ApplyE incurs $O(\mathsf{r}_{\mathsf{AE}})$ rounds and a communication of $O((|\mathsf{V}|+|\mathsf{E}_i|)\cdot \mathsf{c}_{\mathsf{AV}})$. Similarly, ApplyV incurs $O(\mathsf{r}_{\mathsf{AV}})$ rounds and a communication of $O(|\mathsf{V}|\cdot \mathsf{c}_{\mathsf{AV}})$. Here, r_{AE} and r_{AV} (respectively c_{AE} and c_{AV}) denote the round (respectively communication) complexity of securely computing the functions f_{AE} and f_{AV}. Thus, compute on G_i incurs $O(\mathsf{r}_{\mathsf{AE}}+\mathsf{r}_{\mathsf{AV}})$ rounds and $O((n(|\mathsf{V}|+|\mathsf{E}_i|)+(|\mathsf{V}|+|\mathsf{E}_i|)\cdot \mathsf{c}_{\mathsf{AE}}+|\mathsf{V}|\cdot \mathsf{c}_{\mathsf{AV}})$ communication. Since computations on each G_i can be performed in parallel, the total communication complexity for this step is $O(\sum_{i=1}^{n} n(|\mathsf{V}|+|\mathsf{E}_i|)+(|\mathsf{V}|+|\mathsf{E}_i|)\cdot \mathsf{c}_{\mathsf{AE}}+|\mathsf{V}|\cdot \mathsf{c}_{\mathsf{AV}}) = O(n^2|\mathsf{V}|+n|\mathsf{E}|+n(|\mathsf{V}|+|\mathsf{E}_i|)\cdot \mathsf{c}_{\mathsf{AE}}+n|\mathsf{V}|\cdot \mathsf{c}_{\mathsf{AV}}))$ and the round complexity is $O(\mathsf{r}_{\mathsf{AE}}+\mathsf{r}_{\mathsf{AV}})$ rounds. Finally, during 'combine', the generation of updated G from G_is is also achieved non-interactively. Therefore, the total cost incurred for one iteration of DCC is $O(\mathsf{r}_{\mathsf{AE}}+\mathsf{r}_{\mathsf{AV}})$ rounds and $O(n^2|\mathsf{V}|+n|\mathsf{V}|\cdot \mathsf{c}_{\mathsf{AE}}+n|\mathsf{V}|\cdot \mathsf{c}_{\mathsf{AV}}+n|\mathsf{E}|+|\mathsf{E}|\cdot \mathsf{c}_{\mathsf{AE}})$ communication. Thus, the round complexity is independent of n, and the communication is quadratic in n.

4.2 Achieving linear communication

We now describe how to achieve a communication complexity that is linear in n. The primary bottleneck arises due to Propagate and Gather incurring quadratic communication during the compute step. Specifically, the communication cost for Propagate and Gather in compute step for each DAG-list G_i is $O(n\cdot(|\mathsf{V}|+|\mathsf{E}_i|))$. Since there are n DAG-lists, the communication cost accumulates to $O(n^2|\mathsf{V}|)$. Hence, subsequently, we limit our discussion to improving the communication in Propagate and Gather.

To improve this communication complexity, we observe that it suffices for each G_i to include only the vertices in V_i and their immediate neighbours rather than all vertices V. This is because only these vertices contribute to updating the state of vertices in V_i. Let $\bar{\mathsf{V}}_i$ denote the set of vertices in V_i along with their immediate neighbours. Since the size of $\bar{\mathsf{V}}_i$ is bounded by $|\bar{\mathsf{V}}_i| \leq 2|\mathsf{E}_i|$, this reduces the size of DAG-list G_i to $O(|\mathsf{E}_i|)$. Consequently, the communication cost incurred for Propagate and Gather in each G_i is reduced to $O(n\cdot|\mathsf{E}_i|)$. As a result, the total communication cost across all G_is reduces to $O\left(\sum_{i=1}^{n} n|\mathsf{E}_i|\right) = O(n|\mathsf{E}|)$.

While this optimisation reduces the communication cost of message-passing, it introduces the new challenge of generating the DAG-list representation G_i to only include the newly defined $\bar{\mathsf{V}}_i$ during the decompose phase. Specifically, constructing G_i now requires identifying the immediate neighbours of vertices in V_i. The labels of these neighbours are private and not known to all parties. Thus, during each iteration of message-passing, their secret shares must be securely extracted from the secret-shared list of vertex set V present in G. Once extracted, these can be concatenated with E_i to form the DAG-list representation of the newly defined G_i.

A naive approach to securely extract $\bar{\mathsf{V}}_i$ would involve invoking the Permute + Share primitive. Party P_i, knowing the labels of vertices in $\bar{\mathsf{V}}_i$, defines a permutation that reorders the vertex list V such that all vertices in $\bar{\mathsf{V}}_i$ appear first, followed by the remaining vertices. The parties then set the first $\min(|\mathsf{V}|, 2|\mathsf{E}_i|)$ entries of the permuted list as the secret shares of $\bar{\mathsf{V}}_i$. This process incurs the cost of one invocation of Permute + Share, which requires one round and $O(n|\mathsf{V}|)$ communication for a list of size $|\mathsf{V}|$. Since there are n DAG-list, and their generation can occur in parallel, the total cost of DAG-list generation using this naive approach is one round and $O(n^2|\mathsf{V}|)$ communication. This is unlike the previous case where the subgraph generation was non-interactive.

To improve the communication cost of DAG-list generation, we observe that while different permutations need to be applied to V to generate each of the DAG-list G_i, these permutations are applied to the same list V. Leveraging this

insight, we design an amortised Permute + Share protocol that reduces the cost of generating all DAG-lists. The protocol takes as input a secret-shared list V and n permutations (each known to one party) and outputs n secret-shared lists, each permuted according to one of the input permutations. This amortised approach reduces the total cost of DAG-list generation to $O(1)$ rounds and $O(n|\mathsf{V}|)$ communication. We next describe details of Amortised Permute + Share protocol.

Amortised Permute + Share: This protocol enables the generation of multiple permuted lists from a single secret-shared input list. Specifically, 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 $\langle \mathsf{T}_j \rangle$ for $j = 1$ to n, such that $\mathsf{T}_j = \pi_j(\mathsf{T})$.

Recall that in Permute + Share, the preprocessing phase involves the generation of shuffle correlation consisting of secret-shares $\langle \mathsf{R} \rangle$ and $\langle \pi_i(\mathsf{R}) \rangle$ for a random mask R, while the online phase involves reconstructing the masked input list $(\mathsf{T} + \mathsf{R})$ towards the designated party P_i. The designated party then computes its output share as $\pi_i(\mathsf{T} + \mathsf{R}) - \langle \pi_i(\mathsf{R}) \rangle_i$, while all other parties $P_j \in \mathcal{P}$ set their shares as $\langle \mathsf{T}_\mathsf{O} \rangle_j = -\langle \pi_i(\mathsf{R}) \rangle_j$ for $j \neq i$. A straightforward realisation of Amortised Permute + Share would involve running n parallel instances of Permute + Share, where each party contributes its permutation π_i in a distinct instance. This requires generating n distinct shuffle correlations $(\langle \mathsf{R}_i \rangle, \langle \pi_i(\mathsf{R}_i) \rangle)$ during the preprocessing phase and reconstructing n different masked lists $(\mathsf{T} + \mathsf{R}_i)$ in the online phase. However, since the reconstruction in Permute + Share incurs a communication of $O(n\mathsf{N})$, communication cost of naive approach accumulates to $O(n^2\mathsf{N})$.

Towards improving the communication complexity, we avoid using separate masks for each party and instead use a single shared mask R not known to any party. Elaborately, during the preprocessing phase, the parties generate $\langle \mathsf{R} \rangle$ along with $\langle \pi_1(\mathsf{R}) \rangle$, $\langle \pi_2(\mathsf{R}) \rangle$, …, $\langle \pi_n(\mathsf{R}) \rangle$, where R is permuted according to the permutations supplied by each party. In the online phase, the parties reconstruct a single masked input, $\mathsf{T} + \mathsf{R}$, among all. This reconstruction of $\mathsf{T} + \mathsf{R}$ can be efficiently realised using the P_{king} approach [6]. Here, each party sends its share of $\mathsf{T} + \mathsf{R}$ to a designated king party, which reconstructs the masked list and sends it to all other parties. This approach requires only two rounds and incurs a communication cost of $O(n\mathsf{N})$. After reconstruction, each party P_i computes its share of the n output lists i.e $\langle \mathsf{T}_j \rangle$ for $j = 1$ to n as follows: it sets $\langle \mathsf{T}_j \rangle_i = \pi_i(\mathsf{T} + \mathsf{R}) - \langle \pi_i(\mathsf{R}) \rangle_i$ for $j = i$ and $\langle \mathsf{T}_j \rangle_i = -\langle \pi_i(\mathsf{R}) \rangle_i$ for $j \neq i$. In this way, the Amortised Permute + Share protocol can be realised in $O(1)$ rounds and $O(n\mathsf{N})$ communication. The formal protocol for Amortised Permute + Share appears in Fig. 9 and its proof of security appears in §C. We next discuss how Amortised Permute + Share facilitates efficient DAG-list generation during the decompose phase. Note that compute and combine follow along same lines as in §4.1 and §4.1, respectively, and hence, we do not repeat the details.

DAG-list generation during decompose phase: Given the vertex set V, which is additively shared among all parties, the DAG-list generation proceeds as follows. Each party P_i defines a permutation π_i^G that reorders the vertex list such that the vertices in $\bar{\mathsf{V}}_i$ (i.e., vertices associated with P_i and their immediate neighbours) appear first, followed by the remaining vertices. Note that these permutations depend on the input graph structure, which may not be available during the preprocessing phase. However, the Amortised Permute + Share protocol requires permutations to be predefined during preprocessing. To address this, the parties invoke the Amortised Permute + Share protocol, with the random permutations $\hat{\pi}_i^G$ for $i \in [n]$, which are applied to the secret-shared vertex list V in parallel. The output of this step is a set of permuted, secret-shared vertex lists, where the permutations applied are $\hat{\pi}_i^G$ for $i \in [n]$. Next, each party P_i computes and shares the public permutation $\sigma_i^G = \pi_i^G \circ \hat{\pi}_i^{G^{-1}}$ for $i \in [n]$ with all other parties. Subsequently, each party applies the public permutation σ_i^G locally to the output of the Amortised Permute + Share protocol. This final step effectively reorders the vertex list as per π_i^G, ensuring that vertices in $\bar{\mathsf{V}}_i$ are positioned at the beginning of the list. The resulting secret-shared outputs, denoted as $\langle \mathsf{V}'_i \rangle$ for $i = 1$ to n, represent the permuted vertex lists for each DAG-list. To generate the DAG-list for subgraph G_i, the parties extract the first $\min(|\mathsf{V}|, 2|\mathsf{E}_i|)$ entries from $\langle \mathsf{V}'_i \rangle$ and concatenate these with the corresponding secret-shared edge list $\langle \mathsf{E}_i \rangle$. Observe here that the mapping for reordering the vertices in V remains the same across iterations. Hence, the same random permutation $\hat{\pi}_i^G$ for $i \in [n]$ can be used in every iteration. Thus, the public permutation σ_i^G for $i \in [n]$ can be computed once and sent during a one-time initialisation phase by P_i. This concretely reduces the overhead for the subgraph generation by one round. The resulting secret-shared outputs, denoted as $\langle \mathsf{V}'_i \rangle$ for $i = 1$ to n, represent the permuted vertex lists for each subgraph. To generate the DAG-list G_i, the parties extract the first $\min(|\mathsf{V}|, 2|\mathsf{E}_i|)$ entries from $\langle \mathsf{V}'_i \rangle$ and concatenate these with the corresponding secret-shared edge list $\langle \mathsf{E}_i \rangle$.

Complexity of the solution: Our DCC approach to message-passing consists of three main steps. First, 'decompose' where DAG-list generation is realised using the Amortised Permute + Share protocol. This step involves an invocation of Amortised Permute + Share on V, which incurs a communication complexity of $O(n(|\mathsf{V}|)$, and $O(1)$ rounds. Second, 'compute' where message-passing computations are performed in the DAG-list G_i, for $i \in [n]$, using the enhanced approach described in §4.1. In this step, G_i is of size $O(|\mathsf{E}_i|)$ and hence incurs a cost of $O(\mathsf{r}_{\mathsf{AE}} + \mathsf{r}_{\mathsf{AV}})$ rounds and $O((n + \mathsf{c}_{\mathsf{AE}} + \mathsf{c}_{\mathsf{AV}})|\mathsf{E}_i|)$ communication. Since all G_i's can be processed in parallel, 'compute' step incurs $O(\mathsf{r}_{\mathsf{AE}} + \mathsf{r}_{\mathsf{AV}})$ rounds and a communication of $\sum_{i \in [n]} O((n + \mathsf{c}_{\mathsf{AE}} + \mathsf{c}_{\mathsf{AV}})|\mathsf{E}_i|) = O((n + \mathsf{c}_{\mathsf{AE}} + \mathsf{c}_{\mathsf{AV}})|\mathsf{E}|)$. Finally, 'combine' where the extraction happens similar to as described in §4.1, is non-interactive and incurs no additional rounds or communication. Thus, one iteration of DCC incurs $O(\mathsf{r}_{\mathsf{AE}} + \mathsf{r}_{\mathsf{AV}})$ rounds and $O(n|\mathsf{V}| + (n + \mathsf{c}_{\mathsf{AE}} + \mathsf{c}_{\mathsf{AV}})|\mathsf{E}|)$ commu-

nication. Hence, we achieve communication complexity that is linear in n, without hampering rounds.

5 emGraph Framework

In this section, we formally describe the end-to-end protocol of emGraph for computing any message-passing graph algorithm. The ideal functionality for the same, $\mathcal{F}_{\mathsf{MPA}}$, appears in Fig. 11 and the corresponding secure protocol of emGraph, denoted as Π_{MPA}, appears in Fig. 12. The protocol takes as input the local view of the graph from each party P_i ($i \in [n]$), consisting of V_i and E_i. Additionally, the protocol also takes as input the description of the message-passing graph algorithm alg agreed upon by all parties. Without loss of generality, we assume that all parties receive the updated state of their vertices (i.e. V_i) as the output. However, the protocol can be modified to reconstruct the output towards any intended recipient. We next give an elaborate overview of the protocol. The protocol consists of four phases: Input Sharing, Initialisation, Message Passing, and Reconstruction which are elaborated below.

Input sharing: In this phase, parties secret-share their input with all other parties. Specifically, $P_i \in \mathcal{P}$ generates $\langle\cdot\rangle$-shares of V_i and E_i, as defined in §4.1. Following this, parties set $\langle\mathsf{G}\rangle = \langle\mathsf{V}\rangle, \{\langle\mathsf{E}_i\rangle\}_{i\in[n]}$ where $\langle\mathsf{V}\rangle = \langle\mathsf{V}_1\rangle || \ldots || \langle\mathsf{V}_n\rangle$. Observe that, as described in §3, the generation of secret shares of V_i and E_i for $i \in [n]$ can occur non-interactively, given the shared key setup. Thus, this phase does not incur any overhead in terms of rounds and communication.

Initialisation: In this phase, parties pre-compute the public permutations that are repeatedly used during for DAG-list generation (during 'decompose') and message passing computation (during 'compute') of G_i, for $i \in [n]$.

- To facilitate extraction of the vertices $\langle\bar{\mathsf{V}}_i\rangle$ from $\langle\mathsf{V}\rangle$ during the decompose phase, P_i generates and distributes among the rest of the parties the following mappings: π_i^G and σ_i^G. These are distributed as follows. Party P_i does the following. With the knowledge of the labels of vertices in $\bar{\mathsf{V}}_i$ and their position in V, P_i defines a mapping π_i^G which reorders the V such that all the vertices in $\bar{\mathsf{V}}_i$ appear first followed by other vertices. Following this, P_i samples a random permutation $\hat{\pi}_i^G$ and computes the public permutation $\sigma_i^G = \pi_i^G \circ \hat{\pi}_i^{G^{-1}}$ and send it to all the parties[7]. Looking ahead, the extraction of $\langle\bar{\mathsf{V}}_i\rangle$ during decompose is achieved by applying the permutation $\hat{\pi}_i^{G^{-1}}$ using the Amortised Permute + Share protocol followed by locally applying the public permutation σ_i^G on $\langle\mathsf{V}\rangle$. This reorders the $\langle\mathsf{V}\rangle$ as required while hiding the permutation π^G from other parties.
- Similarly, to facilitate transitions between the different ordering of $\langle\mathsf{G}_i\rangle$ efficiently, party P_i generates and distributes among the rest of the parties the certain mappings, as follows. With the knowledge of the topology of G_i, P_i defines three permutations, i.e π_i^S - Permutation that rearranges G_i in vertex order to source order, π_i^D - Permutation that rearranges G_i in source order to destination order, π_i^V - Permutation that rearranges G_i in destination order to vertex order. Following this, P_i samples random permutations $\hat{\pi}_i^S$, $\hat{\pi}_i^D$ and $\hat{\pi}_i^V$ and sends the public mapping $\sigma_i^S = \pi_i^S \circ \hat{\pi}_i^{S^{-1}}$, $\pi_i^D = \pi_i^D \circ \hat{\pi}_i^{D^{-1}}$ and $\sigma_i^V = \pi_i^V \circ \hat{\pi}_i^{V^{-1}}$. Note that the transitions between different orderings of G_i can be performed efficiently by one invocation of Permute + Share followed by applying a public permutation, as illustrated in Fig. 2 and described in Fig. 12.

Message-passing via DCC: Consists of multiple iterations of message-passing as required for the graph algorithm. Each iteration comprises secure realisation of Decompose, Compute and Combine as described in §4. Specifically, generation of $\langle\mathsf{G}_i\rangle$ from $\langle\mathsf{G}\rangle$ for $i \in [n]$ in Decompose step involves one call to Amortised Permute + Share. The Compute step involves securely performing message-passing computation, which is done via secure realisation of Propagate, ApplyE, Gather and ApplyV over G_i for $i \in [n]$. Elaborately, for each G_i, this involves,

- Propagate: The parties evaluate steps 1-3 of algorithm 1 on $\langle\mathsf{G}_i\rangle$. Then, parties invoke Permute + Share [3] to apply $\hat{\pi}_i^{S^{-1}}$ on $\langle\mathsf{G}_i\rangle$, followed by applying the public permutation σ_i^S on their local shares $\langle\mathsf{G}_i\rangle$ to transition to the source order. Finally, the parties evaluate steps 5-7 of algorithm 1 on $\langle\mathsf{G}_i\rangle$.
- Apply-Edges: Parties invoke $\mathcal{F}_{\mathsf{MPC}}$ to securely realise the computation of f_E on data component at each entry in $\langle\mathsf{G}_i\rangle$.
- Transition from source to destination order: Parties invoke Permute + Share [3] to apply the permutation $\hat{\pi}_i^{D^{-1}}$ on $\langle\mathsf{G}_i\rangle$, followed by applying the public permutation σ_i^D to transition to its destination order.
- Gather: In the destination order of $\langle\mathsf{G}_i\rangle$, parties evaluate steps 1-3 of Gather as described in algorithm 2 on $\langle\mathsf{G}_i\rangle$. Following this, parties invoke Permute + Share to apply $\hat{\pi}_i^{V^{-1}}$ on $\langle\mathsf{G}_i\rangle$, followed by applying the public permutation σ_i^V on their local shares $\langle\mathsf{G}_i\rangle$ to generate its vertex order. Finally, the parties evaluate steps 5-7 of algorithm 2 on $\langle\mathsf{G}_i\rangle$.
- Apply-Vertices: Parties invoke $\mathcal{F}_{\mathsf{MPC}}$ to securely realise the computation of f_V on data component at each entry in $\langle\mathsf{G}_i\rangle$.

Finally, in the 'combine' step, parties extract and combine the updated vertex set $\langle\mathsf{V}_i\rangle$ and edge set $\langle\mathsf{E}_i\rangle$ from $\langle\mathsf{G}_i\rangle$ for $i \in [n]$ to generate $\langle\mathsf{G}\rangle$, as described in §4.1, that is used in the next iteration of DCC.

Reconstruction: After multiple iterations of message passing, the parties decompose the $\langle\cdot\rangle$-shares of V as $\mathsf{V}_1||...||\mathsf{V}_n$ and reconstruct the result of V_i towards party P_i.

Complexity of emGraph: The input sharing phase is non-interactive and does not incur any rounds or communication. The initialisation phase involves party P_i for $i \in [n]$ sending the four public permutations each of size $O(|\mathsf{E}|_i)$

[7] In practice, only the first $\min(|\mathsf{V}|, 2|\mathsf{E}_i|)$ entries of σ_i^G (that correspond to the mapping of $\bar{\mathsf{V}}_i$) is communicated. Hence the overall cost for communicating all the permutations σ_i^G, for $i \in [n]$, is $O(n|\mathsf{E}|)$.

to all the parties, thus this step incurs one round and a total of $O(\sum_{i\in[n]} n|\mathsf{E_i}|) = O(n|\mathsf{E}|)$ communication. Each iteration of message-passing via DCC, as discussed in §4.2 incurs $O(\mathsf{r_{AE}} + \mathsf{r_{AV}})$ rounds and $O(n|\mathsf{V}| + (n + \mathsf{c_{AE}} + \mathsf{c_{AV}})|\mathsf{E}|)$ communication. Finally, in the reconstruction step $O(|\mathsf{V}_i|)$ entries are reconstructed towards P_i. This incurs requires one round and $O(n\mathsf{V})$ communication. Thus, the total cost of securely evaluating any message-passing graph algorithm alg using emGraph is $O(k \cdot (\mathsf{r_{AE}} + \mathsf{r_{AV}}))$ and $O(k \cdot n|\mathsf{V}| + k \cdot (n + \mathsf{c_{AE}} + \mathsf{c_{AV}})|\mathsf{E}|)$. Here k is the number of iterations of message-passing required by alg.

Optimisation: Recall that in Graphiti, the DAG-list representation of a graph is a list where each entry consists of the following components: src, dst, isV, and data. However, in emGraph, it suffices to have a simplified list representation where each entry contains only two components: isV and data. The presence of src, dst components in the DAG-list in Graphiti aid in reordering the DAG-list during message-passing. However, in emGraph, we leverage the fact that this information with respect to the G_i is known to party P_i when reordering the G_i. Hence, emGraph does not require these components to be part of the DAG-list.

Security of emGraph**:** We prove the security of emGraph in the standard real-world/ideal-world simulation paradigm. The detailed security proof appears in §C.

6 Benchmarks

We implement and empirically benchmark the performance of emGraph and compare it against the state-of-the-art framework, Graphiti [10]. To ensure a fair comparison, both frameworks are instantiated using the generic MPC protocol of [5] in the semihonest setting. The protocols are implemented from scratch in C++ using the code base of [18]. We note that our code[8] is developed for benchmarking. The benchmarks are conducted on Ubuntu servers equipped with AMD Ryzen Threadripper PRO 5965WX processors and 256GB RAM. Each party is executed as a separate process on the same machine, with the simulated network connection of 100ms latency and 100MBps bandwidth. Our code accounts for multithreading and each party is initiated with a maximum of 10 threads. Since the performance depends only on the size of the graph and not its structure, we benchmark both emGraph and Graphiti on a randomly generated scale-free graph. We split the vertices in the graph equally among all the parties, i.e., we assume each party approximately owns $|\mathsf{V}|/n$ vertices and the corresponding edges of the graph. For all our experiments we consider the online run time and online communication as the parameters for benchmark. We focus on initialisation and the message-passing via DCC phases as these are the most compute-intensive whereas input sharing and reconstruction happen as a one-off event.

[8] https://anonymous.4open.science/r/EmGraph

Initialisation: We first compare the initialisation costs of emGraph and Graphiti [10]. Since the initialisation phase of Graphiti has round complexity dependent on both the number of parties and the graph size, we provide comparisons by varying both parameters.

Table 2 reports the comparison of initialisation while varying the number of parties from 2 to 25 where $\mathsf{N} = |\mathsf{V}| + |\mathsf{E}|$ is 10^5. Our initialisation clearly outperforms that of Graphiti. Specifically, we see improvements of up to 2200× in runtime and 270× in communication for 15 parties and graph of size 10^5 ($|\mathsf{V}| + |\mathsf{E}|$). Note that Graphiti's initialisation phase is infeasible for more than 15 parties in our considered experimental setup due to memory constraints. The significant improvements in the runtime can be attributed to the lightweight initialisation phase of emGraph, which only involves parties communicating some permutations to each other in one round of interaction. In contrast, Graphiti has an initialisation cost proportional to number of parties, which is also computationally expensive as it requires secure sorting operations[9]. Note that in contrast to the theoretical expectation, the runtime for Graphiti does not scale linearly with number of parties. We believe this is because the computational cost incurred by secure sorting operations overshadows the expected linear runtime increase from additional rounds.

Ref	#Parties	Runtime (s)	Comm. (MB)
Graphiti [10]	2	239.94	160.07
emGraph		0.10	1.40
Graphiti [10]	5	380.39	1360.32
emGraph		0.11	7.52
Graphiti [10]	10	562.89	5760.79
emGraph		0.14	24.12
Graphiti [10]	15	786.71	13161.35
emGraph		0.35	48.72
Graphiti [10]	20	-	-
emGraph		0.36	76.76
Graphiti [10]	25	-	-
emGraph		0.57	101.76

Table 2: Comparison of initialisation phase for varying number of parties and graph of size $|\mathsf{V}| + |\mathsf{E}| = 10^5$.

Table 3 shows the comparison as the graph size increases from 10^4 to 10^7, with the number of parties fixed at $n = 5$. Here too, emGraph witnesses an improvement of 3450× for a graph of size 10^5. Again due to computational constraints in our experimental setup (all parties running on the same machine), Graphiti's initialisation phase runs out of memory for graph sizes larger than 10^5. Finally, we also observe that with increasing graph size, there is minimal increase in runtime of our initialisation. For graph size 10^7, the increase is slightly higher and is attributed to the increase in communication as for this graph size, the bandwidth becomes a bottleneck.

[9] The secure sorting operations in the initialisation phase of Graphiti, are micro-benchmarked based on the secure variant of quick-sort protocol described in [1].

$\|V\|+\|E\|$	Ref	Runtime (s)	Comm. (MB)
10^4	Graphiti [10]	104.14	136.02
	emGraph	0.10	0.75
10^5	Graphiti [10]	380.39	1360.32
	emGraph	0.11	7.52
10^6	Graphiti [10]	-	-
	emGraph	0.13	75.20
10^7	Graphiti [10]	-	-
	emGraph	0.36	752.0

Table 3: Comparison of initialisation phase for varying graph size ($|V|+|E|$) and 5 parties.

Overall, these results in Table 2 and Table 3 highlight the efficiency and scalability of emGraph's initialisation phase, particularly in scenarios with a large number of parties or graphs of large size.

Message passing via DCC: We next compare the message-passing costs of emGraph and Graphiti for the PageRank computation application described in §B.2. Since the message-passing phase of Graphiti scales with the number of parties, we vary the number of parties from $n = 2$ to $n = 25$ and report the results in Table 4. As expected, the DCC-based message-passing approach in emGraph outperforms Graphiti, where we witness runtime improvements of up to 11× for $n = 25$ parties. Moreover, the relative improvement of emGraph over Graphiti grows with the number of parties, increasing from 1.5× for $n = 2$ to 11× for $n = 25$. This improvement can be attributed to the constant round complexity of 5 rounds in emGraph, compared to the round complexity of $3n$ in Graphiti. Although, emGraph incurs a slightly higher communication cost than Graphiti, its impact on runtime is negligible. This shows that the improvements in the rounds overshadow the impact of slightly increase in communication.

Ref	#Parties	Runtime (s)	Comm. (MB)
Graphiti [10]	2	1.14	4.80
emGraph		0.74	2.64
Graphiti [10]	’5	2.99	19.20
emGraph		0.78	13.44
Graphiti [10]	10	6.06	43.20
emGraph		0.91	41.04
Graphiti [10]	15	9.12	67.20
emGraph		1.07	80.64
Graphiti [10]	20	12.18	91.20
emGraph		1.25	123.12
Graphiti [10]	25	15.26	115.20
emGraph		1.39	155.52

Table 4: Comparison of one iteration of message passing for varying number of parties and graph of size $|V|+|E| = 10^5$.

Overall improvements: By combining data reported in Tables 2, 3, and 4, we estimate total online runtime for 10 iterations of PageRank computation(computed as ‘initialisation cost’ + 10 times the cost of ‘message passing’). For Graphiti, overall runtime is estimated to be 877.91 seconds and the overall communication is estimated to be 13GB for graph of 10^5 and 15 parties. In contrast, emGraph achieves significantly improved runtime of just 10.81 seconds and communication of 130MB, representing an 80× and 106× improvement in runtime and communication respectively, over Graphiti.

For completeness, we also provide the end-to-end benchmarks of PageRank computation using emGraph in Table 5. Here, we vary the number of parties, n, from 2 to 25, and the graph size ($|V|+|E|$) from 10^4 to 10^6. As expected, the runtime shows minimal variation with changes in both the number of parties and the graph size, which can be attributed to emGraph's constant round complexity that remains independent of these factors. However, we observe a slightly higher increase in runtime for a graph size of 10^6 when varying n, compared to graphs of sizes 10^4 and 10^5. This discrepancy is likely due to the computational resource constraints in our experimental setup, where all parties are initialised as separate processes on the same machine. As the number of parties increases, the computational resources available to each party decrease. We anticipate even better performance when each party is run on a dedicated machine. Finally, emGraph demonstrates its efficiency and practicality by processing a graph of size 10^6 distributed among 25 data owners in under a minute, making it highly suitable for real-world applications.

Graph size	#Parties	Runtime (s)	Comm.(MB)
10^4	2	9.17	2.80
	5	9.21	14.08
	10	9.82	42.48
	15	10.20	82.88
	20	10.57	126.16
	25	10.90	159.36
10^5	2	9.45	28.00
	5	9.93	140.80
	10	11.16	424.80
	15	12.28	828.80
	20	13.40	1261.60
	25	14.19	1593.60
10^6	2	11.02	280.00
	5	11.84	1408.00
	10	18.21	4248.00
	15	25.84	8288.00
	20	36.45	12616.00
	25	45.04	15936.00

Table 5: End-to-end runtime for 10 iterations of PageRank computation using emGraph for varying number of parties and varying graph of size.

7 Conclusion

We design emGraph, a generic secure framework for efficiently evaluating message-passing graph algorithms in the multiparty setting. 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 attain the claimed efficiency and scalability, emGraph completely eliminates the need for expensive operations such as secure sort and secure shuffle that constituted the primary bottleneck in prior works. emGraph instead relies on a much more simpler and efficient Permute + Share primitive.

All this allows emGraph to attain improvements of up to 80× in comparison to prior works. Finally, we also design a new primitive called Amortised Permute + Share which can be of independent interest.

References

[1] Toshinori Araki, Jun Furukawa, Kazuma Ohara, Benny Pinkas, Hanan Rosemarin, and Hikaru Tsuchida. Secure graph analysis at scale. In *ACM CCS*, 2021.

[2] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. Practical fully secure three-party computation via sublinear distributed zero-knowledge proofs. In *ACM CCS*, 2019.

[3] Melissa Chase, Esha Ghosh, and Oxana Poburinnaya. Secret-shared shuffle. In *ASIACRYPT*, 2020.

[4] Harsh Chaudhari, Ashish Choudhury, Arpita Patra, and Ajith Suresh. ASTRA: High Throughput 3PC over Rings with Application to Secure Prediction. In *ACM CCSW@CCS*, 2019.

[5] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In *ESORICS*, pages 1–18, 2013.

[6] Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In *CRYPTO*, 2007.

[7] Saba Eskandarian and Dan Boneh. Clarion: Anonymous communication from multiparty shuffling protocols. In *NDSS*, 2022.

[8] Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, and Bhavish Raj Gopal. Entrada to secure graph convolutional networks. *Cryptology ePrint Archive*, 2023.

[9] Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, Bhavish Raj Gopal, et al. Find thy neighbourhood: Privacy-preserving local clustering. *PoPETs*, 2023.

[10] Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, and Bhavish Raj Gopal. Graphiti: Secure graph computation made more scalable. In *ACM CCS 2024*, 2024.

[11] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. SWIFT: Super-fast and robust privacy-preserving machine learning. In *USENIX Security*, 2021.

[12] Payman Mohassel and Peter Rindal. ABY^3: A mixed protocol framework for machine learning. In *ACM CCS*, 2018.

[13] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. Graphsc: Parallel secure computation made easy. In *IEEE S&P*, 2015.

[14] Arpita Patra and Ajith Suresh. BLAZE: Blazing Fast Privacy-Preserving Machine Learning. In *NDSS*, 2020.

[15] A Pranav Shriram, Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, Bhavish Raj Gopal, and Somya Sangal. Ruffle: Rapid 3-party shuffle protocols. *PoPETS*, 2023.

[16] Bhavish Raj Gopal. Privacy-preserving graph analysis. In *ACM CCS 2024*, 2024.

[17] Marie Beth van Egmond, Vincent Dunning, Stefan van den Berg, Thomas Rooijakkers, Alex Sangers, Ton Poppe, and Jan Veldsink. Privacy-preserving anti-money laundering using secure multi-party computation. *Cryptology ePrint Archive*, 2024.

[18] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.

[19] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *FOCS*, 1982.

[20] Zhenhua Zou, Zhuotao Liu, Jinyong Shan, Qi Li, Ke Xu, and Mingwei Xu. Cognn: Towards secure and efficient collaborative graph learning. In *ACM CCS 2024*, 2024.

A Prelims

A.1 Threat model

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 $\mathsf{F} : \{0,1\}^\kappa \to \mathbb{Z}_{2^\ell}$ F is a secure PRF. The functionality, $\mathcal{F}_{\mathsf{setup}}$ appears in Fig. 4.

The shared key setup enables all parties to the non-interactive generation of $\langle\cdot\rangle$-shares of a random 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 $\mathcal{F}_{\mathsf{setup}}$

$\mathcal{F}_{\mathsf{MPA}}$ 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 $i \in [n]$.

Figure 4: Ideal functionality for setup.

A.2 Graphiti

Graphiti securely evaluates a graph algorithm by invoking multiple iterations of the message-passing phase. Each iteration involves performing Propagate, which entails performing computations on the DAG-list in the vertex order, followed by a transition to the source order. This is followed by ApplyE that ensures that data on each edge in the DAG-list is updated under the function $f_{\mathsf{AE}}(\cdot)$. This is followed by a transition from source order to the destination order to perform Gather. Gather entails performing computations on the destination order, and transitioning to the vertex order. Finally, ApplyV ensures that data on each node in the DAG-list is updated under the function $f_{\mathsf{AV}}(\cdot)$. A pictorial representation of the overall process during each iteration of the message passing phase in Graphiti appears in Fig. 5. We next describe the approach of Graphiti to realise these primitive Propagate and Gather.

Propagate. The formal details of Propagate appear in Algorithm 1. Let G represent the DAG-list where $\mathsf{N} = |\mathsf{V}| + |\mathsf{E}|$ denotes the size of DAG-list. $\mathsf{G}[i]$ denotes the i^{th} entry. Let $\mathsf{G}[i].\mathsf{data_s}$ denote the data to be propagated (if the entry is a vertex) and $\mathsf{G}[i].\mathsf{data_r}$ denotes the data component that receives the propagated information . Initially, $\mathsf{G}[i].\mathsf{data_r} = 0$ for all entries, and $\mathsf{G}[i].\mathsf{data_s} = 0$ for edge entries. Consider a linear scan of the DAG-list sorted in source order where, during the scan, the $\mathsf{data_r}$ is updated as the data present at all the vertices preceding this edge. This is realised by performing a cumulative sum of the $\mathsf{data_s}$ present at every entry preceding the current entry, i.e. $\mathsf{G}[i].\mathsf{data_r} = \sum_{j=1}^{i-1} \mathsf{G}[j].\mathsf{data_s}$ for $i = 1$ to N. Since this is a linear operation, it can be realised non-interactively via MPC. However, after the linear scan, the sum at each edge possesses the data to be propagated from its source vertex plus the data present on the vertices that appear before it in the source ordered DAG-list. To remove the latter part contributing to the sum, the vertex order is used to adjust the $\mathsf{data_s}$ component of the vertices such that the cumulative sum computes the intended data to be propagated. Specifically, the value to be propagated by a vertex is computed as $\mathsf{G}[i].\mathsf{data_s}' = \mathsf{G}[i].\mathsf{data_s} - \mathsf{G}[i-1].\mathsf{data_s}$ for $i = 1$ to $|\mathsf{V}|$. Since this step is also comprised entirely of linear operations, it can also be performed non-interactively within MPC. Following this, the G is ordered in the source order, and the propagated data is computed as $\mathsf{G}[i].\mathsf{data_r} = \sum_{j=1}^{i-1} \mathsf{G}[j].\mathsf{data_s}'$ for $i = 1$ to N. In the secure realisation of Propagate, the only interactive operation involved is the transition between vertex order and source order of DAG-list. This can be realised in rounds independent of graph size, akin to the transition between source and destination order in [1] as elaborated ahead. Propagate is followed by ApplyE. Similar to ApplyV, this operation is local to entries of DAG-list and hence does not require any linear scan or transition between different ordering.

Gather. The formal details of Gather appear in Algorithm 2. During Gather, each vertex aggregates $\mathsf{data_r}$ component from its incoming edges using an aggregation operation denoted as $\oplus$. Graphiti observed that many graph applications such as histogram [13], matrix factorisation [13], BFS [1], Pagerank [9], clustering [9], graph neural networks (GNN) [8] etc. can be represented using a linear aggregation operation and leveraged this to design an efficient algorithm for Gather. For example, consider addition as the aggregation operation. Let $\mathsf{G}[i].\mathsf{data_g}$ (initialised to 0) denote the data component of an entry that stores aggregated information. To begin with, a linear scan of the destination ordered DAG-list is performed. During the scan, each entry computes a cumulative sum of the $\mathsf{data_r}$ component of all entries that precede it and stores it in the $\mathsf{data_g}'$ component i.e $\mathsf{G}[i].\mathsf{data_g}' = \sum_{j=1}^{i-1} \mathsf{G}[j].\mathsf{data_r}$ for all $i = 1$ to N. Since this only requires linear operations, it can be realised non-interactively within MPC. The $\mathsf{data_g}'$ component of a vertex aggregates not only the $\mathsf{data_r}$ component from its incoming edges but also the $\mathsf{data_r}$ component accumulated by the preceding entries in DAG-list. However, this additional information aggregated by the vertex is indeed the data aggregated by the preceding vertex in the DAG-list. This additional information that is gathered by each vertex is removed by transitioning to the vertex order of the DAG-list where $\mathsf{data_g}$ component is computed as $\mathsf{G}[i].\mathsf{data_g} = \mathsf{G}[i].\mathsf{data_g}' - \mathsf{G}[i-1].\mathsf{data_g}'$ for $i = 1$ to N. This also comprises entirely of linear operations and is non-interactive. The transition from destination ordering to vertex ordering can be realised in rounds independent of graph size [1]. Note that the above approach works only for a linear aggregation operation. For algorithms requiring non-linear aggregation, Graphiti falls back to the approach of [1]. We refer an interested reader to [1, 10] for further details. Finally, Gather is followed by ApplyV. Observe that at the end of Gather, the DAG-list is sorted in the vertex order where all the vertices appear together. Thus, the function f_{AV} is applied in parallel only to the first $|\mathsf{V}|$ entries of the DAG-list that corresponds to the vertices.

Algorithm 1: Propagate

Input: DAG-list G in vertex order

for $i = |\mathsf{V}|$ *to* 2 **do**
  $\mathsf{G}[i].\mathsf{data_r} = \mathsf{G}[i].\mathsf{data_s} - \mathsf{G}[i-1].\mathsf{data_s}$;
end
Transition to the source order of the DAG-list ;
for $i = \mathsf{N}$ *to* 1 **do**
  $\mathsf{G}[i].\mathsf{data_r} = \sum_{j=1}^{i} \mathsf{G}[j].\mathsf{data_r} - \mathsf{G}[i].\mathsf{data_s}$;
end

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, [1] 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-

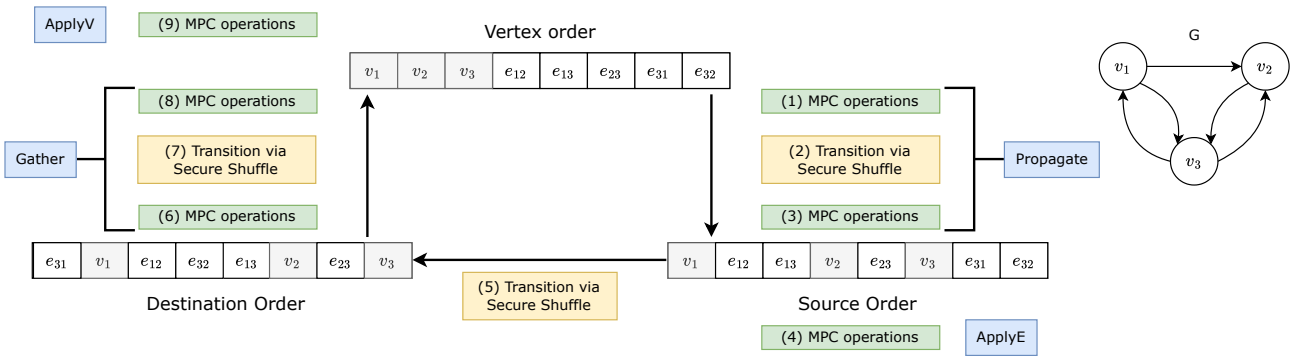

Figure 5: Steps in one iteration of message-passing round in Graphiti.

Algorithm 2: Gather

Input: DAG-list G in destination order

for $i = 1$ *to* N **do**
  $\mathsf{G}[i].\mathsf{data}_g = \sum_{j=1}^{i} \mathsf{G}[j].\mathsf{data}_r$;
end
Transition to the vertex order of the DAG-list;
for $i = \mathsf{N}$ *to* 2 **do**
  $\mathsf{G}[i].\mathsf{data}_g = \mathsf{G}[i].\mathsf{data}_g - \mathsf{G}[i-1].\mathsf{data}_g$;
end

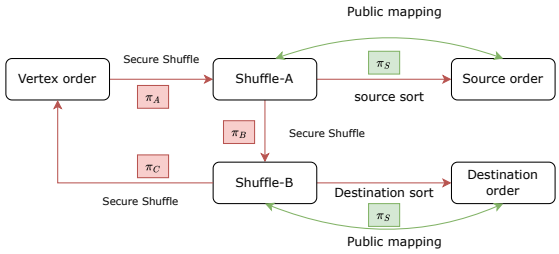

Figure 6: Initialisation in Graphiti.

time initialisation phase. The initialisation phase begins with the DAG-list in vertex order. Using a secure shuffle protocol, 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. 6.

B emGraph

B.1 DCC approach for message passing

An illustration of the traditional approach for message-passing appears in Fig. 7. In contrast the DCC approach realises message passing by (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) computing message-passing algorithm on the subgraphs $\mathcal{G}_i$ in parallel, instead of doing the same on the global graph $\mathcal{G}$, (iii) combining the results on these subgraphs to realise message-passing on the global graph $\mathcal{G}$ as illustrated in Fig. 1.

B.2 PageRank computation

Here, we discuss how PageRank computation can be realised as a message passing. Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, the PageRank algorithm is used to rank the nodes in $\mathcal{V}$ by importance. The algorithm involves multiple iterations where the rank (state) of every node u is updated as follows:

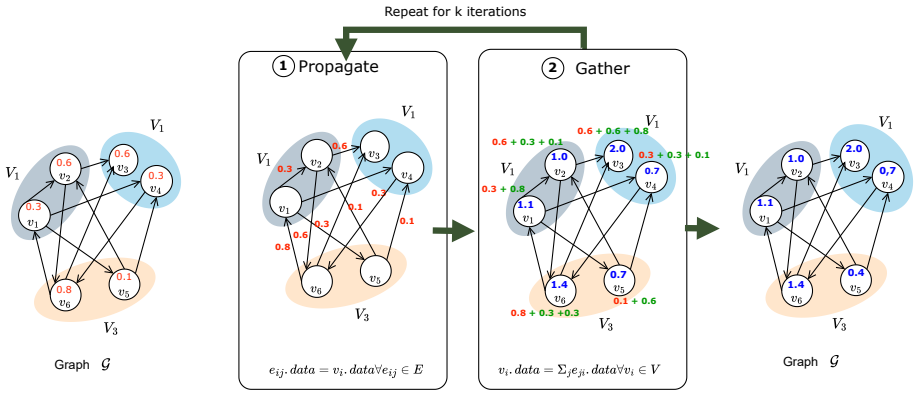

Figure 7: Operations involved in the traditional approach for message passing on global graph $\mathcal{G}$

$$PR(\mathsf{u}) = \frac{1-\alpha}{|\mathsf{V}|} + \alpha \sum_{\mathsf{e}(\mathsf{v},\mathsf{u}) \in \mathsf{E}} \frac{PR(\mathsf{v})}{\deg(\mathsf{v})}$$

where α is the damping factor, which is usually set to 0.85 and $\deg(\mathsf{v})$ is the out degree of the vertex v.

The PageRank computation is expressed as a message-passing algorithm as follows. Every vertex $\mathsf{v} \in \mathsf{V}$ ($\mathcal{V}$ with the corresponding data components) comprises two data components, one for the PageRank (PR) of the vertex and the other storing the precomputed value $\rho = \frac{1}{\deg(\mathsf{v})}$. Every edge $\mathsf{e}(\mathsf{u},\mathsf{v})$ comprises a single data component data_r to store the data propagated by vertices. The initial PageRank value of each vertex is set to $PR(\mathsf{v}) = \frac{1}{|\mathsf{V}| \cdot \deg(\mathsf{v})}$ The algorithm proceeds iteratively, with each iteration comprising Propagate, Gather and ApplyV. An explicit call for ApplyE is not required. Propagate is defined to propagate the PR component of a vertex onto its outgoing edges i.e. $\mathsf{e}.\mathsf{data}_\mathsf{s} = \mathsf{u}.PR$ for every $\mathsf{e}(\mathsf{u},\mathsf{v}) \in \mathsf{E}$. Gather is defined to compute a cumulative sum of the data_r values of all incoming edges and is stored in the PR component of the node. This ensures that each vertex receives and accumulates PageRank contributions from its neighbours. ApplyV updates the PR component of each vertex based on its aggregated value and the damping factor α. For all iterations except the last, PR component that has to be propagated is computed as:

$$\mathsf{v}.PR = \mathsf{v}.\rho \times \left(\frac{1-\alpha}{|\mathsf{V}|} + \alpha \cdot \mathsf{v}.PR \right).$$

In the final iteration, ApplyV simplifies to directly updating the PageRank value as:

$$\mathsf{v}.PR = \frac{1-\alpha}{|\mathsf{V}|} + \alpha \cdot \mathsf{v}.PR.$$

C 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 + Share protocol, followed by the security of emGraph.

The ideal functionality for Amortised Permute + Share appears in Fig. 8, and the secure protocol that realises it, Π_{APS}, appears in Fig. 9.

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

Without loss of generality, let $C \subset \mathcal{P}$ denote the parties corrupted by adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{APS}}$ interacts with parties in $\mathcal{P}$ and $\mathcal{S}$. It receives as input $\langle \mathsf{T} \rangle_i$-$P_i$'s share of the input list T and π_i from party P_i for $i = 1$ to n.
$\mathcal{F}_{\mathsf{APS}}$ proceeds as follows.

- Reconstruct input T using $\langle \cdot \rangle$-shares of parties in $\mathcal{P}$.
- Generate $\mathsf{T}_j = \pi_j(\mathsf{T})$ for $j = i$ to n.
- Generate random $\langle \cdot \rangle$-shares of T_j for $j = 1$ to n.
- Send $(\mathsf{Output}, \langle \mathsf{T}_j \rangle_i)$ to P_i for $i = 1$ to n and $j = 1$ to n.

Figure 8: Ideal functionality for Amortised Permute + Share.

Lemma C.1: The protocol, Π_{APS} (Fig. 9) securely realises the functionality $\mathcal{F}_{\mathsf{APS}}$ (Fig. 8) against a semi-honest adversary that corrupts at most $n-1$ parties in $\mathcal{P}$.

Proof. Observe that the protocol relies on invoking $\mathcal{F}_{\mathsf{Share\ Translation}}$. Hence, the security follows from the security of the underlying protocols for $\mathcal{F}_{\mathsf{Share\ Translation}}$. Apart from this, the only communication is in the online phase during reconstruction. Here, observe that T' and the honest parties

share of T' are both lists of random values. Thus, if P_{king} is one of the corrupt parties, then the simulator also sends random values as the shares of honest parties to P_{king}. If $P_{\text{king}} \notin C$, then the simulator picks a random list T' and sends it to all parties in the C. □

Protocol Π_{APS}

Inputs: Each party P_i inputs $\langle \mathsf{T} \rangle_i$ share of the list T with N entries and permutation π_i.
Outputs: Additive shares of n lists $\langle \mathsf{T}_j \rangle$ for $j = 1$ to n, such that $\mathsf{T}_j = \pi_j(\mathsf{T})$ shared between parties in $\mathcal{P}$.

Preprocessing:

- Parties generate $\langle \cdot \rangle$-shares of a random value R (see §A.1).
- Parties generate $\langle \cdot \rangle$-shares of $\pi_i(\mathsf{R})$ for $i = 1$ to n by invoking pairwise share translation functionality $\mathcal{F}_{\text{Share Translation}}$ [7].

Online:

- Parties reconstruct $\mathsf{T}' = \mathsf{T} + \mathsf{R}$ using th P_{king} approach. For this, each party P_i for $i \in [n]$ locally computes $\langle \mathsf{T} \rangle_i + \langle \mathsf{r} \rangle_i$ and sends to the designated king party P_{king} the party P_{king} reconstructs T' and sends it back to all the parties.
- Party P_i for $i = 1$ to n sets $\langle \mathsf{T}_i \rangle_i = \pi_i(\mathsf{T}') - \langle \pi_i(\mathsf{R}) \rangle_i$ and $\langle \mathsf{T}_j \rangle_i = -\langle \pi_j(\mathsf{R}) \rangle_i$ for $j = 1$ to n and $j \neq i$.

Figure 9: Amortised Permute + Share protocol.

Simulator $\mathcal{S}_{\Pi_{\text{APS}}}$

$\mathcal{S}_{\Pi_{\text{APS}}}$ proceeds as follows.

Preprocessing:

- Using the keys commonly held with $\mathcal{A}$ (generated as part of $\mathcal{F}_{\text{setup}}$), sample the common randomness.
- Simulator emulates the $\mathcal{F}_{\text{Share Translation}}$ on behalf of the honest parties.

Online:

- If $P_{\text{king}} \in C$, then the simulator sends random values as the shares of honest parties to P_{king}. It receives T' from P_{king} and forwards it to other parties in C.
- If $P_{\text{king}} \notin C$, then the simulator picks a random list T' and sends it to all parties in the C.

Figure 10: Simulator for Amortised Permute + Share protocol.

The ideal functionality for computing any message-passing graph algorithm appears in Fig. 11, and the secure protocol that realises it, Π_{MPA}, appears in Fig. 12.

Lemma C.2: The protocol, Π_{MPA} (Fig. 12) securely realises the functionality $\mathcal{F}_{\text{MPA}}$ (Fig. 11) against a semi-honest adversary that corrupts at most $n-1$ parties in $\mathcal{P}$.

Proof. Observe that the protocol relies on invoking $\mathcal{F}_{\text{Permute+Share}}$, $\mathcal{F}_{\text{APS}}$ and $\mathcal{F}_{\text{MPC}}$. Hence, the security follows from the security of the underlying protocols for the same. Apart from this, the only communication is in the online phase during reconstruction, which the simulator perfectly simulates by adjusting the shares of honest parties. □

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

Without loss of generality, let $C \subset \mathcal{P}$ denote the set of parties corrupted by adversary $\mathcal{S}$. $\mathcal{F}_{\text{MPA}}$ interacts with parties in $\mathcal{P}$ and $\mathcal{S}$. It receives as input $\mathsf{V}_i, \mathsf{E}_i$ from party P_i for $i \in [n]$. All parties in $\mathcal{P}$ agree on the message-passing algorithm alg, which defines the computation details such as the number of iterations for message passing (k), data to be propagated/aggregated, the functions $f_{\mathsf{AE}}, f_{\mathsf{AV}}$ etc. They send alg to $\mathcal{F}_{\text{MPA}}$. $\mathcal{F}_{\text{MPA}}$ proceeds as follows.

- Construct the graph $\mathsf{G} = (\mathsf{V}, \mathsf{E})$ where $\mathsf{V} = \mathsf{V}_1 || ... || \mathsf{V}_n$ and $\mathsf{E} = \mathsf{E}_1 || ... || \mathsf{E}_n$.
- For $i = 1$ to k
- Perform k iterations of message passing algorithm alg in G.
- Decompose the V as V_i for $i \in [n]$.
- Send $(\mathsf{Output}, \mathsf{V}_i)$ to the party P_i for $i = 1$ to n.

Figure 11: Ideal functionality for computing message passing algorithm.

Protocol Π_{MPA}

Inputs: Party P_i holds as input its local view of the graph $\mathsf{V}_i, \mathsf{E}_i$ for $i \in [n]$. All parties in $\mathcal{P}$ agree on the message-passing algorithm alg, which defines the computation details such as the number of iterations for message passing (k), data to be propagated/aggregated, the functions $f_{\mathsf{AE}}, f_{\mathsf{AV}}$ etc.
Outputs: Party P_i receives V_i, where the data components in V_i are updated according to the algorithm alg.

Setup:

- Parties invoke the $\mathcal{F}_{\text{setup}}$ for key distribution (see Fig. 4).

Input Sharing:

- Each party P_i secret-shares the inputs V_i and E_i (see §A.1).
- Parties generate the secret-shared list representation of the graph as $\langle \mathsf{G} \rangle = \langle \mathsf{V} \rangle, \{\langle \mathsf{E}_j \rangle\}_{j \in [n]}$ where $\langle \mathsf{V} \rangle = \langle \mathsf{V}_1 \rangle || \ldots || \langle \mathsf{V}_n \rangle$. Here, each entry in V and E_i for $i \in [n]$ consists of two component isV and data as described in §4.1.

Initialisation: Each party $P_i \in \mathcal{P}$ does the following.

- Defines the following permutations: π_i^G - that rearranges the V such that all the vertices in V_i and its immediate neighbours appear together, π_i^S - that rearranges the DAG-list of subgraph G_i in vertex order to source order, π_i^D - that rearranges the DAG-list of subgraph G_i in source order to destination order, π_i^V - that rearranges the DAG-list of subgraph G_i in destination order to vertex order.
- Samples the following random permutations: $\hat{\pi}_i^G$, $\hat{\pi}_i^S$, $\hat{\pi}_i^D$, $\hat{\pi}_i^V$.

- Computes and sends the public mapping $\sigma_i^G = \pi_i^G \circ \hat{\pi}_i^{G^{-1}}$, $\sigma_i^S = \pi_i^S \circ \hat{\pi}_i^{S^{-1}}$, $\pi_i^D = \pi_i^D \circ \hat{\pi}_i^{D^{-1}}$, $\sigma_i^V = \pi_i^V \circ \hat{\pi}_i^{V^{-1}}$ to every other party

Message Passing: For k iterations do the following.

Decompose:

– The parties invoke $\mathcal{F}_{\mathsf{APS}}$ with inputs $\langle \mathsf{V} \rangle_i$ and permutation $\hat{\pi}_i^G$ from party P_i for $i = 1$ to n to generate n secret shared lists $\langle \mathsf{V}_j \rangle$ for $j \in [n]$.

– The parties locally apply the public permutation σ_j^G on $\langle \mathsf{V}_j \rangle$ and extract the first $min(\mathsf{V}, 2|\mathsf{E_i}|)$ entries to generate $\langle \bar{\mathsf{V}}_j \rangle$ for $j \in [n]$.

– Parties generate the $\langle \cdot \rangle$-shares of DAG-list G_j in vertex order as $\langle \bar{\mathsf{V}}_j \rangle || \langle \mathsf{E}_j \rangle$ for $j \in [n]$.

Compute:

– For $i = 1$ to n in parallel:

- **Propagate:** The parties invoke $\mathcal{F}_{\mathsf{MPC}}$ to evaluate steps 1-3 of Propagate as described in algorithm 1 on $\langle \cdot \rangle$ shares of G_i. Following this, the parties invoke $\mathcal{F}_{\mathsf{Permute+Share}}$ to apply $\hat{\pi}_i^S$ followed by applying the public permutation σ_i^S on their local shares of DAG-list G_i to generate the source order. Finally, the parties invoke $\mathcal{F}_{\mathsf{MPC}}$ to evaluate steps 5-7 of algorithm 1 on $\langle \cdot \rangle$-shares of G_i.
- **Apply-Edges:** Parties invoke $\mathcal{F}_{\mathsf{MPC}}$ to compute f_E on the shares of the data component at each entry in the DAG-list G_i.
- **source order to destination order:** The parties invoke one instance of $\mathcal{F}_{\mathsf{Permute+Share}}$ which applies $\hat{\pi}_i^D$ on shares of DAG-list G_i followed by applying the public permutation σ_i^D to get the destination order.
- **Gather:** The parties invoke $\mathcal{F}_{\mathsf{MPC}}$ to evaluate steps 1-3 of Gather as described in algorithm 2 on $\langle \cdot \rangle$ shares of G_i. Following this, the parties invoke $\mathcal{F}_{\mathsf{Permute+Share}}$ to apply $\hat{\pi}_i^V$ followed by applying the public permutation σ_i^V on their local shares of DAG-list G_i to generate the vertex order. Finally, the parties invoke $\mathcal{F}_{\mathsf{MPC}}$ to evaluate steps 5-7 of algorithm 2 on $\langle \cdot \rangle$-shares of G_i.
- **Apply-Vertices:** Parties invoke $\mathcal{F}_{\mathsf{MPC}}$ to compute f_V on the shares of the data component at each entry in the DAG-list G_i.

Combine:

– Parties non-interactively extract the shares of V_i from the DAG-list G_i i.e $\langle \mathsf{V}_i \rangle = \langle \mathsf{G}_i \rangle [1 : |\mathsf{V}_i|]$ for $i = 1$ to n. Parties set $\langle \mathsf{V} \rangle = \langle \mathsf{V}_1 \rangle || .. \langle \mathsf{V}_n \rangle$.

– Parties non-interactively extract the shares of E_i from the DAG-list G_i i.e $\langle \mathsf{E}_i \rangle = \langle \mathsf{G}_i \rangle [|\bar{\mathsf{V}}_i| + 1 :]$ for $i = 1$ to n.

– Parties set $\langle \mathsf{G} \rangle = (\langle \mathsf{V} \rangle, \{\mathsf{E}_i\}_{i \in [n]})$ as the updated list representation of the graph.

Reconstruction:

- Parties reconstruct $\langle \mathsf{V}_i \rangle$ towards party P_i for $i \in [n]$. This entails every party communicating its share of V_i to party P_i.

Figure 12: Secure protocol of emGraph for message passing algorithm.

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

$\mathcal{S}_{\Pi_{\mathsf{APS}}}$ proceeds as follows.

Setup:

– The simulator emulates $\mathcal{F}_{\mathsf{setup}}$ on behalf of the honest parties.

– Using the keys commonly held with $\mathcal{A}$ (generated as part of $\mathcal{F}_{\mathsf{setup}}$), sample the common randomness.

Input Sharing:

– There is nothing to simulate as this step is non-interactive.

Initialisation:

- On behalf of each honest party $P_i \in \mathcal{P}$ the simulator picks and sends random permutations $\sigma_i^G, \sigma_i^S, \sigma_i^D$ and σ_i^V.

Message Passing: For k iterations do the following.

Subgraph Generation:

– The simulator emulates $\mathcal{F}_{\mathsf{APS}}$ on behalf of the honest parties.

Subgraph Message Passing:

– For each subgraph G_i for $i = 1$ to n:

- **Propagate:** The simulator emulates $\mathcal{F}_{\mathsf{Permute+Share}}$ on behalf of honest parties. Note that steps 1-3 and 5-7 of Propagate (Algorithm 1) are non-interactive in MPC. Hence, there is nothing to simulate for these steps.
- **Apply-Edges:** The simulator emulates $\mathcal{F}_{\mathsf{MPC}}$ on behalf of honest parties.
- **source order to destination order:** The simulator emulates $\mathcal{F}_{\mathsf{Permute+Share}}$ on behalf of honest parties.
- **Gather:** The simulator emulates $\mathcal{F}_{\mathsf{Permute+Share}}$ on behalf of honest parties. Note that steps 1-3 and 5-7 of Gather (Algorithm 2) are non-interactive in MPC. Hence, there is nothing to simulate for these steps.
- **Apply-Vertices:** The simulator emulates $\mathcal{F}_{\mathsf{MPC}}$ on behalf of honest parties.

Combine:

– There is nothing to simulate as this step is non-interactive.

Reconstruction:

- For every party $P_i \in C$, the simulator simulates the reconstruction phase by adjusting the shares of honest parties with respect to the output V_i. Specifically, it sends random shares on behalf of honest parties $P_{k+1}, ..., P_{n-1}$ and sends $\mathsf{V}_i - \sum_{j=1}^{n-1} \langle \mathsf{V}_i \rangle_j$ on behalf of P_n.

Figure 13: Simulator for message passing algorithm.

D Additional Benchmarks

Preprocessing cost. For completeness we report the preprocessing cost of emGraph. For simplicity, we instantiate the preprocessing in the presence of a trusted helper. We note that the cost of the preprocessing phase consists of the trusted helper generating the necessary correlated randomness and sending it to the computing parties in one round of interaction. In the absence of a trusted helper, the correlated randomness can be generated using generic MPC protocols in [3, 5]. Table 6 reports the preprocessing cost of the emGraph for the 10 iterations of PageRank computation.

Graph size	#Parties	Runtime (s)	Comm.(MB)
10^4	2	0.18	1.40
	5	0.33	1.88
	10	0.76	2.68
	15	1.38	3.48
	20	2.03	4.04
	25	2.52	4.24
10^5	2	1.21	14.00
	5	3.07	18.80
	10	7.51	26.80
	15	13.39	34.80
	20	19.62	40.40
	25	24.34	42.40
10^6	2	18.57	280.00
	5	30.73	188.00
	10	75.35	268.00
	15	135.06	348.00
	20	200.51	404.00
	25	259.05	424.00

Table 6: End-to-end runtime for 10 iterations of PageRank computation using emGraph for varying number of parties and graph of size $|\mathsf{V}| + |\mathsf{E}| = 10^5$.