Graphiti: **Secure Graph Computation Made More Scalable**

Nishat Koti
koti@encrypto.cs.tu-darmstadt.de
Technical University of Darmstadt
Darmstadt, Germany

Varsha Bhat Kukkala
varshabhat15@gmail.com
Independent Researcher
Bangalore, India

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

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

Abstract

Privacy-preserving graph analysis allows performing computations on graphs that store sensitive information, while ensuring all the information about the topology of the graph as well as data associated with the nodes and edges remains hidden. The current work addresses this problem by designing a highly scalable framework, Graphiti, that allows securely realising any graph algorithm. Graphiti relies on the technique of secure multiparty computation (MPC) to design a generic framework that improves over the state-of-the-art framework of GraphSC by Araki et al. (CCS'21). The key technical contribution is that Graphiti has round complexity independent of the graph size, which in turn allows attaining the desired scalability. Specifically, this is achieved by (i) decoupling the Scatter primitive of GraphSC into separate operations of Propagate and ApplyE, (ii) designing a novel constant-round approach to realise Propagate, as well as (iii) designing a novel constant-round approach to realise the Gather primitive of GraphSC by leveraging the linearity of the aggregation operation. We benchmark the performance of Graphiti for the application of contact tracing via BFS for 10 hops and observe that it takes less than 2 minutes when computing over a graph of size 10^7. Concretely it improves over the state-of-the-art up to a factor of 1034× in online run time. Similar to GraphSC by Araki et al., since Graphiti relies on a secure protocol for shuffle, we additionally design a shuffle protocol secure against a semi-honest adversary in the 2-party with a helper setting. Given the versatility of shuffle protocol, the designed solution is of independent interest. Hence, we also benchmark the performance of the designed shuffle where we observe improvements of up to 1.83× in online run time when considering an input vector of size 10^7, in comparison to the state-of-the-art shuffle protocol in the considered setting.

CCS Concepts

• **Security and privacy → Cryptography**.

Keywords

Secure graph analysis, secure computation, secure shuffle

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3670393

ACM Reference Format:
Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, and Bhavish Raj Gopal. 2024. Graphiti: Secure Graph Computation Made More Scalable . In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 21 pages. https://doi.org/10.1145/3658644.3670393

1 Introduction

The expressive capabilities and ease of processing graphs has resulted in designing several graph algorithms that derive meaningful information about the underlying system being modelled as a graph $G(\mathsf{V}, \mathsf{E})$. Traditionally, these graph algorithms take as input the underlying graph data in the clear. However, since the graph may model sensitive user information, processing them in the clear may raise privacy concerns. We showcase these privacy concerns via a representative use case of contact tracing as required to analyse the spread of contagious diseases. For example, to trace the spread of COVID-19, one would require analysing the contact network of the individuals in the considered population. The contact network is a graph where each individual is represented as a node, while an edge represents the physical contact between the individuals. Contact tracing translates to performing a breadth-first search (BFS) on the contact network, starting at a node representing an infected person. This allows identifying all individuals who are within a predetermined threshold distance from the start node. Clearly, the information of who has come in contact with whom is regarded as sensitive data. Since individuals may not wish to disclose this information (i.e., the presence or absence of an edge) in the open, it must remain private. These privacy concerns make it challenging to analyse the global contact network. In this way, graphs that store sensitive data may be distributed across multiple data owners such that each is only aware of a subset of the edges that comprise the overall graph, while the labelling of the nodes is known to all the data owners. Thus, there is a need to design privacy-preserving solutions that facilitate graph analysis without leaking the global graph data in the clear. The current work focuses on addressing this via the technique of secure multiparty computation (MPC).

MPC is a cryptographic technique that allows n mutually distrusting parties to carry out computations on private inputs such that an adversary controlling up to $t < n$ parties does not learn anything other than the output of the computation. In the considered scenario, the private input is the graph that is distributedly held by multiple data owners, while the function comprises the desired graph algorithm. Note that any graph algorithm can be

securely computed using off-the-shelf generic MPC frameworks [4, 31]. However, performing this naively would require operating on the *adjacency matrix* representation of the graph. This would incur a complexity of $O(|V|^2)$, where $|V|$ denotes the number of nodes (or vertices) present in the graph. However, real-world graphs are known to be sparse, and hence, operating over the adjacency matrix representation is an overkill. With the intent of improving the efficiency of secure graph computation, the work of [22] designed the GraphSC framework. The efficiency improvements in [22] are introduced by leveraging the sparsity of real-world networks, where the framework operates on a list representation rather than the adjacency matrix representation of the underlying graph. This allows attaining $O(|V|+|E|)$ complexity, as opposed to $O(|V|^2)$ complexity. This leap in efficiency paved way for adapting GraphSC for securely realising applications, including BFS [2, 24], histograms and matrix factorisation [19, 20], computing Pagerank and its variants [16, 19], graph convolutional network evaluation [15], etc.

GraphSC framework [22]. The GraphSC framework operates on a list-based representation of a data-augmented directed graph $G(V, E, \text{data})$. Here, data is a set of user-defined values associated with each node and edge of the graph. The graph is expressed as a list, referred to as the DAG-list G, which comprises an entry corresponding to every node and edge in the graph. Thus, the DAG-list is of size $|V| + |E|$. Each entry in the DAG-list is encoded such that it facilitates processing the graph without having to disambiguate between entries corresponding to those of a node from those of an edge. Given the DAG-list, the GraphSC framework enables securely evaluating a graph algorithm expressed as a message-passing algorithm. The latter involves updating the data component of the nodes and edges across several iterations in a message passing phase. The GraphSC framework achieves this via the primitives of Scatter, Gather and Apply. Informally, Scatter involves propagating data present at the source node onto its outgoing edges. This is done by performing a linear scan of the DAG-list, with the entries appearing in a specific order known as the *source order*. The source order is an ordering of the DAG-list where every node (in the sequence from 1 to $|V|$) appears immediately before all of its outgoing edges. Since Scatter, by definition, propagates information on outgoing edges, a linear scan of the DAG-list, where information is picked up at every node followed by dropping off the same at every (outgoing) edge entry that follows, allows realising Scatter. On the other hand, Gather involves aggregating data present on the incoming edges of a given node. Similar to Scatter, a linear scan of the DAG-list with the entries in *destination order* allows realising Gather. Here, destination order is an ordering of the DAG-list such that all incoming edges of a particular node are placed immediately before that node. During a linear scan of the destination ordered DAG-list, data is picked up at every incoming edge and aggregated, while the aggregated data is dropped off at the following node. Finally, primitive Apply allows the data at every node to be updated based on data aggregated during Gather. Since this operation can be performed in parallel on all node entries, Apply does not require a linear scan of DAG-list. In summary, the GraphSC framework showcases how these Scatter-Gather-Apply primitives allow realising one iteration of the message-passing algorithm. Thus, securely realising a graph algorithm boils down to securely realising Scatter-Gather-Apply via MPC. A detailed discussion of GraphSC is deferred to §2.2.

Improved GraphSC framework [2]. When designing a secure solution for the Scatter-Gather-Apply primitives via MPC, several parameters determine the efficiency of the resulting solution—(i) round complexity, which accounts for the sequential interactions among the computing parties, (ii) communication complexity, which accounts for the information (messages) exchanged, and (iii) computation complexity, which accounts for the local computations performed at each party. In this regard, the GraphSC framework of [22] relies on garbled circuits (GC) to securely realise the primitives. Although GC-based solutions have constant round complexity, they are known to be both communication and computation-intensive. Hence, [2] instead relies on secret-sharing-based techniques to improve the GraphSC framework of [22] to reduce the overhead due to communication and computation. However, this introduces the challenge of reducing the round complexity[1].

Specifically, the challenge lies in designing a round efficient solution for—(a) Scatter-Gather primitives, and (b) transitioning between the source order and destination order of the DAG-list when invoking the Scatter, Gather. In this regard, [2] primarily focuses on (b) and makes the following observation. [22] relies on a secure sort protocol to transition between source order and destination order of DAG-list. However, a secret-sharing-based secure sort protocol would incur a high round complexity. Hence, [2] showcases efficiency improvements that can be achieved by replacing secure sort with a secure shuffle[2] followed by (partially) insecure sort[3]. This approach is shown to be much more efficient and requires constant rounds for transitioning between the different orderings of the DAG-list given an initialisation phase[4]. However, secure realisations of Scatter-Gather (step (a)), continue to have a linear round complexity in the size of the DAG-list, $N = |V| + |E|$. To this end, [2] attempts to provide a round optimised (RO) variant of Scatter-Gather, albeit requiring a higher communication cost. Specifically, assuming a bound B on the maximum degree of nodes, this solution has a round complexity of $O(\log(B))$. However, since several real-world graphs have B in the $O(N)$ [26], the resultant solution continues to have a round complexity dependent on N.

Graphiti *framework.* The current work takes a stride ahead and puts forth an improved framework, Graphiti. Specifically, while Graphiti continues to use the shuffle-then-sort paradigm of [2] to address (b), Graphiti addresses (a) by proposing a novel approach for Scatter-Gather, which when realised via MPC, has a round complexity that is *independent* of N. In this way, Graphiti is able to reduce the overhead due to Scatter-Gather, which constitutes a major fraction of the task in the message-passing phase. It is worthwhile to note that Graphiti also has a better communication and computation complexity than [2]. To put the improvements of Graphiti in perspective, Table 1 showcases that Graphiti has the

[1] Secret-sharing-based protocols for general MPC are known to have round complexity that is proportional to multiplicative depth of the circuit being evaluated.
[2] Given secret-shares of a list of elements as input, the protocol outputs shares of the list where each entry is reordered based on an unknown random permutation.
[3] Unlike a secure comparison-based sort protocol, where the input and the results of the intermediate comparison are hidden, in a partially insecure sort the results of the comparison are made known in the clear. Hence, parties learn the permutation that results in the sorted list in the clear.
[4] Computations repeated across iterations are pushed to a one-time initialisation phase.

best features from all the existing frameworks for secure graph computation. Thus, existing applications such as BFS [2, 24], histograms and matrix factorisation [19, 20], computing Pagerank and its variants [16, 19], graph convolutional network evaluation [15], etc. can be directly improved by using Graphiti as a drop-in substitute for the underlying secure graph computation frameworks. Finally, it is worthwhile to note that Graphiti is designed as a generic framework and can be instantiated with an appropriate MPC of choice based on the application scenario. This not only allows Graphiti to inherit the latter's security guarantees and efficiency, but also opens up the possibility of utilising the future advancements of MPC in a seamless way.

Parameter	Matrix	GraphSC [22]	GraphSC [2]	GraphSC [2] (RO)	Graphiti
Rounds	○	○	●	◉	○
Communication	●	◉	⊙	◉	⊙
Computation	●	◉	⊙	◉	○

○ - low, ⊙ - moderate, ◉ - medium, ◉ - high, ● - very high.

Table 1: Comparison of secure graph computation frameworks.

1.1 Our contributions

We next outline our contributions and highlight the novel techniques that allow us to attain improved round, communication, and computation complexity. Further, we benchmark Graphiti to showcase the concrete improvements and report the results.

Decoupling Scatter. GraphSC framework in [22] defines Scatter primitive to account for both, propagating data from source node onto outgoing edges, as well as updating data on edges based on the propagated data. Since the improved framework in [2] builds on [22], they continue relying on the same definition. However, we observe that splitting the computation of propagating node data and updating edge data as two distinct phases improves the overall efficiency of Scatter. Hence, in the current work, the task of only propagating node data is accounted within Propagate primitive, while updating edge data is separated out as ApplyE primitive. Collectively, Propagate and ApplyE constitute Scatter primitive. Let f_{AE} denote the function to be applied when updating data on edges. The Scatter in [2] requires $O(\mathsf{N} \cdot \mathsf{r}_{\mathsf{AE}})$ round complexity, where N denotes the size of the DAG-list ($\mathsf{N} = |\mathsf{V}|+|\mathsf{E}|$) and r_{AE}[5] denotes the round complexity of realising f_{AE} via MPC. This complexity is due to the linear scan of the DAG-list, during which, at each edge entry, data is propagated, followed by updating it using f_{AE}. Decoupling immediately improves the overall round complexity of Propagate and ApplyE to $O(\mathsf{N}+\mathsf{r}_{\mathsf{AE}})$, where Propagate has a complexity $O(\mathsf{N})$ and ApplyE has $O(\mathsf{r}_{\mathsf{AE}})$. We, in fact, go a step further and devise a novel approach for achieving Propagate that allows us to realise it with a round complexity independent of N, thereby allowing Scatter to attain a N-independent round complexity.

A new approach to Propagate. To achieve a round complexity independent of N, we do the following—(i) Although the DAG-list is encoded such that node entries cannot be disambiguated from those of edges, the approach for Scatter in [2] requires performing different operations for entries corresponding to nodes from those of edges. To perform these operations obliviously, [2] relies on multiplication operation, which is interactive and hence adds to the round complexity. Unlike this, we design an approach for Propagate that does not distinguish between operations performed at an edge or node entry. (ii) Our approach is designed to rely only on addition/subtraction operations that can be performed non-interactively within MPC. This is unlike in [2], which requires expensive interactive multiplications. (iii) In the process, we introduce a new ordering of the DAG-list called vertex-ordering that aids in achieving (i) and (ii). Here, vertex ordering is one where all nodes appear in sequence from 1 to |V|, followed by all edges. Thus, secure implementation of Propagate involves performing non-interactive operations in both vertex order and source order of DAG-list and a transition from vertex order to source order. The cost of the latter can be reduced to the cost of one invocation to shuffle protocol. Thus, round complexity of Propagate is $O(\mathsf{r}_{\mathsf{shfl}})$. Since there exist constant round shuffle protocols [9, 14, 18, 24], round complexity of our Propagate followed by ApplyE (which make up Scatter) is indeed independent of N. This directly impacts the scalability of Graphiti since the size of the input graph is no longer a bottleneck for achieving Scatter.

New approach to Gather. Unlike Propagate, we observe that aggregating data from incoming edges into a node as part of Gather is already decoupled from the Apply primitive that accounts for updating the node data based on aggregated information. Hence, to keep it consistent, we refer to the latter as the ApplyV primitive in our framework of Graphiti. Since ApplyV can already be performed in parallel across all node entries, we focus on efficiently realising Gather. For this, we make the observation that several graph algorithms can be realised using a linear aggregation operation during Gather. Thus, we leverage this observation in conjunction with the approach as done in Propagate described above (see points (i-iii)) to realise Gather with a round complexity independent of N, unlike that of [2] which required $O(\mathsf{N})$ round complexity. The definition of ApplyV remains the same as Apply originally in [22], where node data is updated via some function f_{AV}.

Graphiti *framework.* Securely realising each iteration of the message-passing phase in Graphiti involves invoking the following primitives in the given sequence—Propagate, ApplyE, Gather and ApplyV. The detailed cost of realising each of these primitives in Graphiti for one iteration of message-passing phase is listed in Table 2, where we compare against the improved GraphSC framework of [2]. Note that since the focus of the current work primarily lies in the Scatter-Gather primitives, the cost of the one-time initialisation, which is similar to the cost of [2], is not accounted for in the table. As stated earlier, apart from the linear variant, [2] also designs a round optimised (RO) variant, which is also reported in the table. Note that the RO variant comes at the cost of increasing the communication complexity from $O(\mathsf{N})$ to $O(\mathsf{N}\log|\mathsf{V}|)$. Further, [2] discusses the RO variant for the specific application of BFS. While the authors claim the techniques can be extended to other applications, it does not follow trivially since one has to account for computations specific to the graph algorithm under consideration. Despite this, as seen in Table 2, our solution not only improves with respect to round complexity but also in terms of communication and computation. The latter are a result of realising Scatter-Gather primitives via non-interactive linear operations as opposed to requiring interactive (non-linear) multiplication operations in [2].

[5]We use $\mathsf{r}_{(\cdot)}$ to denote the round complexity of function $f_{(\cdot)}$ when realised via MPC.

Thus, Graphiti provides overall improvement over [2]. Finally, Table 3 captures scalability of Graphiti and [2] with varying graph size (N) when accounting for the simplistic case of BFS.

On the expressiveness of Graphiti. Note that during Gather, a node aggregates information from its incident edges via an aggregation operation. We observe that many graph applications such as histogram [22], matrix factorisation [22], BFS [2], Pagerank [16], clustering [16], graph neural networks (GNN) [15] etc. can be represented using a linear aggregation operation. In general, any graph algorithm that can be represented via computations of the form $A \cdot \vec{x}$ can be represented as a message passing algorithm with linear aggregation operation (see [29]). Here A represents the adjacency matrix of the graph, and $\vec{x}$ represents the vector of data component associated with vertices. Thus, a linear aggregator operator, as used in Graphiti, suffices for most graph algorithms. On the other hand, note that Scatter in Graphiti is agnostic to the requirements of the aggregation operation. Thus, even for graph algorithms that require a non-linear aggregation, our improved approach for Scatter can be used in conjunction with the Gather of [2] to yield an efficient solution. The resulting solution will continue to improve the overall complexity of evaluating graph algorithms compared to [2].

Benchmarks. Graphiti is generic, i.e., our approach to realising Propagate-ApplyE-Gather-ApplyV primitives is independent of any particular MPC setting or protocol. However, to benchmark the performance of Graphiti, we follow on similar lines to [22] and instantiate it in the simplest setting of two parties (2PC) in the presence of a semi-honest adversary. To attain a fast response time, which is the time taken from submission of input to obtaining the output, we operate in the *preprocessing* paradigm. Here, heavy, input-independent computations are carried out in a preprocessing phase, followed by a fast input-dependent online phase. To improve the efficiency of the preprocessing phase, as done in several works [5, 6, 25, 27, 28], we rely on an additional trusted (incorruptible) helper party only during the preprocessing phase to generate the required preprocessing data. Since Graphiti also relies on a secure shuffle protocol, as an independent contribution, we design the same in the considered 2-party with a helper setting. Since secure shuffle is a versatile primitive in itself and may be of independent interest, we also benchmark its performance and compare it against the state-of-the-art shuffle protocol of [9] by adapting it to the considered setting. Before we highlight the key results with respect to the performance of Graphiti as well as shuffle, we remark the following. While Graphiti is generic and can be instantiated with an appropriate MPC of choice, the considered setting of semi-honest 2PC in the preprocessing paradigm is chosen for benchmark purposes. Thus, depending on the deployment scenario, one can choose to operate in the semi-honest/malicious adversarial model, preprocessing/all-online paradigm, with/without a trusted helper, etc. While these design choices affect the run time, Graphiti will always witness improvements over prior work since it is agnostic to the underlying MPC (see Table 2 for asymptotic improvements). Finally, note that to ensure a fair comparison with prior works, we instantiate all frameworks with the same MPC setting. The key benchmark results are as follows.

- Graphiti: We benchmark the application of contact tracing using BFS described earlier. We observe that when increasing N from 10^4 to 10^7, in comparison to [2], our improvements in run time increase from 585× to 1034× for the linear variant, and 18× to 106× for RO variant. This is because Graphiti's round complexity remains independent of N unlike [2]'s. Moreover, we observe that it takes under 2 minutes to perform the computation on a graph of size 10^7. This shows that our solution is highly scalable.
- Secure shuffle: We design a shuffle protocol secure against a semi-honest adversary in the 2-party with a helper setting. Note that the state-of-the-art 2-party shuffle protocol of [9] can be adapted to the 2-party with a helper setting by offloading the preprocessing computations to a helper party. In comparison to this adaptation of [9], we note that our shuffle protocol has a 2× factor improvement in the online rounds while being on par in terms of communication cost. This is also corroborated by our benchmarks, where we observe improvements of up to 1.83× in online run time. In fact, it takes less than 2 seconds to shuffle a vector of ten million 64-bit elements. A comparison of our shuffle protocol with the adaptation of [9] appears in Table 4. It is worthwhile to note that in frameworks such as GraphSC and Graphiti, multiple invocations of shuffle are required. In such cases, the additional communication of $N\ell$ bits in the preprocessing phase of our shuffle gets amortised across all the shuffle invocations, where the vector being shuffled is of size N and each entry is of size ℓ bits.

Organisation. Preliminaries such as system model, GraphSC [2, 22], etc., appear in §2. §3 provides our Graphiti framework, and §4 discusses our new shuffle protocol. These are followed by benchmarks in §5. Additional details of Graphiti, shuffle and benchmarks appear in §A, §B, and §C, respectively. §D provides a proof sketch for the security of the designed protocols.

2 Preliminaries

2.1 MPC and threat model

Graphiti can be instantiated with any MPC protocol of choice and the security offered by the MPC protocol will be carried forward for Graphiti. For benchmarking reasons, we instantiate it in the 2-party computation (2PC) setting, with at most one semi-honest corruption. Further, since we operate in the preprocessing paradigm, as done in several works [5, 6, 25, 27, 28], we assume the presence of an additional trusted (incorruptible) helper party in the preprocessing phase to generate the required data. Thus, the set of computing parties is denoted as $\mathcal{P} = \{P_0, P_1\}$, while P_2 denotes the trusted helper party which does not collude with parties in $\mathcal{P}$. Without loss of generality, we also use the notation P_i, P_{i-1} for $i \in \{0, 1\}$ to denote parties P_0 and P_1. During the run of the protocol, the helper party P_2 generates and distributes shared randomness to the online parties P_0 and P_1 in an input-independent preprocessing phase. Following this, P_0 and P_1 perform computation in the online phase. The helper party is inactive during the online phase and unaware of the computation the two parties intend to execute. Like in any MPC setting, the parties $\{P_0, P_1, P_2\}$ are connected via pairwise private and authenticated channels over a synchronous network. We let $\mathcal{A}$ denote a static, probabilistic, polynomial time, semi-honest adversary that can corrupt one party amongst $\{P_0, P_1\}$. Our protocols in this setting are proven secure in the standard real-world/ideal-world simulation paradigm.

Parameter	Reference	Scatter*	Gather	ApplyV	Total†
Rounds	GraphSC [2]	$O(\mathsf{N}\cdot \mathsf{r_{mul}}\cdot \mathsf{r_{AE}})$	$O(\mathsf{N}\cdot \mathsf{r_{mul}})$	$O(\mathsf{r_{AV}})$	$O(\mathsf{N}\cdot \mathsf{r_{mul}}\cdot \mathsf{r_{AE}}+\mathsf{r_{AV}}+\mathsf{r_{shfl}})$
	GraphSC [2] (RO) ‡	$O(\log(\lvert V\rvert)\cdot \mathsf{r_{mul}}\cdot \mathsf{r_{AE}})$	$O(\log(\lvert V\rvert)\cdot \mathsf{r_{mul}})$		$O(\log(\lvert V\rvert)\cdot \mathsf{r_{mul}}\cdot \mathsf{r_{AE}}+\mathsf{r_{AV}}+\mathsf{r_{shfl}})$
	Graphiti	$O(\mathsf{r_{shfl}})+O(\mathsf{r_{AE}})$	$O(\mathsf{r_{shfl}})$		$O(\mathsf{r_{AE}}+\mathsf{r_{AV}}+\mathsf{r_{shfl}})$
Communcation	GraphSC [2]	$O(\mathsf{N}\cdot \mathsf{c_{mul}}\cdot \mathsf{c_{AE}})$	$O(\mathsf{N}\cdot \mathsf{c_{mul}})$	$O(\mathsf{N}\cdot \mathsf{c_{AV}})$	$O(\mathsf{N}\cdot \mathsf{c_{mul}}\cdot \mathsf{c_{AE}}+\mathsf{N}\cdot \mathsf{c_{AV}}+\mathsf{c_{shfl}})$
	GraphSC [2] (RO) ‡	$O(\mathsf{N}\cdot\log(\lvert V\rvert)\cdot \mathsf{c_{mul}}\cdot \mathsf{c_{AE}})$	$O(\mathsf{N}\cdot\log(\lvert V\rvert)\cdot \mathsf{c_{mul}})$		$O(\mathsf{N}\cdot\log(\lvert V\rvert)\cdot \mathsf{c_{mul}}\cdot \mathsf{c_{AE}}+\mathsf{N}\cdot \mathsf{c_{AV}}+\mathsf{c_{shfl}})$
	Graphiti	$O(\mathsf{c_{shfl}})+O(\mathsf{N}\cdot \mathsf{c_{AE}})$	$O(\mathsf{c_{shfl}})$		$O(\mathsf{N}\cdot \mathsf{c_{AE}}+\mathsf{N}\cdot \mathsf{c_{AV}}+\mathsf{c_{shfl}})$
Computation	GraphSC [2]	$O(\mathsf{N}\cdot \mathsf{p_{mul}}\cdot \mathsf{p_{AE}})$	$O(\mathsf{N}\cdot \mathsf{p_{mul}})$	$O(\mathsf{N}\cdot \mathsf{p_{AV}})$	$O(\mathsf{N}\cdot \mathsf{p_{mul}}\cdot \mathsf{p_{AE}}+\mathsf{N}\cdot \mathsf{p_{AV}}+\mathsf{p_{shfl}})$
	GraphSC [2] (RO) ‡	$O(\mathsf{N}\cdot\log(\lvert V\rvert)\cdot \mathsf{p_{mul}}\cdot \mathsf{p_{AE}})$	$O(\mathsf{N}\cdot\log(\lvert V\rvert)\cdot \mathsf{p_{mul}})$		$O(\mathsf{N}\cdot\log(\lvert V\rvert)\cdot \mathsf{p_{mul}}\cdot \mathsf{p_{AE}}+\mathsf{N}\cdot \mathsf{p_{AV}}+\mathsf{p_{shfl}})$
	Graphiti	$O(\mathsf{p_{shfl}})+O(\mathsf{N}\cdot \mathsf{p_{AE}})$	$O(\mathsf{p_{shfl}})$		$O(\mathsf{N}\cdot \mathsf{p_{AE}}+\mathsf{N}\cdot \mathsf{p_{AV}}+\mathsf{p_{shfl}})$

* Note that our cost for Scatter accounts for the cost for Propagate + ApplyE, and is hence reported with the split.
† The total cost additionally accounts for the transition from source order to destination order, which is realised using shuffle and is common to both [2] and Graphiti.
‡ The round optimised variant of [2] additionally assumes a bound on the maximum degree of the vertex. For a fair comparison, we set the maximum degree to be V.
$\mathsf{r}_{(\cdot)}$, $\mathsf{c}_{(\cdot)}$ and $\mathsf{p}_{(\cdot)}$ denote the round, communication and computation complexity respectively of function $f_{(\cdot)}$ when securely realised via MPC.
When reporting the cost of [2] we *highlight* the additional overhead in [2] in comparison to Graphiti.

Table 2: Comparison of the cost of primitives in one iteration of the message-passing phase where $\mathsf{N} = |\mathsf{V}| + |\mathsf{E}|$.

Framework	Rounds	Communication	Computation
GraphSC [2]	$O(\mathsf{N})$	$O(\mathsf{N})$	$O(\mathsf{N})$
GraphSC [2] (RO)	$O(\log(\lvert V\rvert))$	$O(\mathsf{N}\log(\lvert V\rvert))$	$O(\mathsf{N}\log(\lvert V\rvert))$
Graphiti	$O(1)$	$O(\mathsf{N})$	$O(1)$

Table 3: Comparison of how the complexity of one iteration of the message-passing phase of BFS scales with the graph size N.

Protocol	Rounds	Communication (bits)	
		Online	Preprocessing
[9]	k	$k(2\mathsf{N}\ell)$	$k(2\mathsf{N}\ell)$
Ours	$2k$	$k(2\mathsf{N}\ell)$	$k(2\mathsf{N}\ell)+\mathsf{N}\ell$

Table 4: Comparison of cost for k shuffle invocations on a vector of N elements where each element is of size ℓ.

The protocols operate over the ring algebraic structure, with $\mathbb{Z}_{2^\ell}$ denoting the ring of ℓ-bit elements. Our protocols work over additive secret sharing denoted by $\langle\cdot\rangle$. We say a value $\mathsf{x}\in\mathbb{Z}_{2^\ell}$ is $\langle\cdot\rangle$-shared or additively shared over $\mathbb{Z}_{2^\ell}$ if P_i for $i\in\{0,1\}$ holds $\langle\mathsf{x}\rangle_i$ such that $\mathsf{x}=\langle\mathsf{x}\rangle_0+\langle\mathsf{x}\rangle_1$. Note that 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 $\mathsf{c}_1\mathsf{x}+\mathsf{c}_2\mathsf{y}$. Parties use a one-time key setup [7, 10, 17, 21, 23] to establish common random keys for a pseudo-random function (PRF) between them. This enables each subset of parties to non-interactively sample a common random ℓ-bit string $\mathsf{v}\in\mathbb{Z}_{2^\ell}$.

Primitives. We rely on the following primitives to securely realise Graphiti. Other primitives as required for the specific graph applications considered are described in place.

– Multiplication: This primitive takes as input two $\langle\cdot\rangle$-shared values $\mathsf{x},\mathsf{y}\in\mathbb{Z}_{2^\ell}$ and outputs the $\langle\cdot\rangle$-shares of $\mathsf{z}=\mathsf{x}\cdot\mathsf{y}$. We let $\mathcal{F}_{\mathsf{mult}}$ denote the ideal functionality for the same.

– Shuffle: This primitive takes as input a $\langle\cdot\rangle$-shared vector $\mathsf{T}\in\mathbb{Z}_{2^\ell}^{\mathsf{N}}$ of size N, where each element in T, i.e., $\mathsf{T}[i],\forall i\in\{1,\ldots,\mathsf{N}\}$, is $\langle\cdot\rangle$-shared. It outputs $\langle\cdot\rangle$-shares of a vector $\mathsf{T_O}\in\mathbb{Z}_{2^\ell}^{\mathsf{N}}$ where $\mathsf{T_O}$ denotes the vector T shuffled under some random secret permutation π. We let $\mathcal{F}_{\mathsf{Shuffle}}$ denote the ideal functionality for this primitive. Note that a permutation of size N, denoted as π, is a bijective function that maps elements from the set $\{1,2,\ldots,\mathsf{N}\}$ to itself. We use the notation $\pi(\mathsf{T})$ to refer to the operation of applying the permutation π on T, which results in reordering the elements in T. Specifically, the i^{th} element in the vector $\mathsf{T}[i]$ is moved to position $\pi(i)$. Since permutation is a bijective mapping, it can be inverted and we use π^{-1} to denote the inverse of the permutation. We use $\pi_0\circ\pi_1$ to denote the composition of permutations i.e. $\pi_0\circ\pi_1(\mathsf{T})=\pi_0(\pi_1(\mathsf{T}))$. Further, we note that a party can sample a random permutation locally using the Fisher-Yates algorithm [12].

– Sort: This primitive takes as input a $\langle\cdot\rangle$-shared vector $\mathsf{T}\in\mathbb{Z}_{2^\ell}^{\mathsf{N}}$, where each element in T is $\langle\cdot\rangle$-shared. It outputs $\langle\cdot\rangle$-shares of a vector $\mathsf{T_O}\in\mathbb{Z}_{2^\ell}^{\mathsf{N}}$ where $\mathsf{T_O}$ denotes the vector T that is sorted. We let $\mathcal{F}_{\mathsf{Sort}}$ denote the ideal functionality for this primitive.

– Insecure sort: This primitive is similar to the secure sort primitive except that during the run of the secure sort protocol, the output of the intermediate secure comparisons is not kept hidden. In this way, the permutation that maps the input vector T to the output $\mathsf{T_O}$ is learned on clear. Note, however, that it is only this mapping from the input to output that is not hidden, the elements of the input and output continue to be $\langle\cdot\rangle$-shared throughout the protocol. We let $\mathcal{F}_{\mathsf{insecSort}}$ denote the ideal functionality for this primitive.

– The underlying MPC: Note that Graphiti may require various other MPC primitives such as comparison, equality etc. depending on the considered graph algorithm. We abstract these out via the $\mathcal{F}_{\mathsf{MPC}}$ functionality that takes inputs from the parties and returns as output the function computed on the inputs. In the 2PC with the helper setting, we instantiate $\mathcal{F}_{\mathsf{MPC}}$ using the 2PC protocols of [11] where the preprocessing data is generated by a helper party.

2.2 GraphSC framework

GraphSC framework of [22]. It takes as input a directed data augmented graph $G(\mathsf{V},\mathsf{E},\mathsf{data})$ represented in the form of a DAG-list G. Each entry in DAG-list G is represented by a tuple which contains the following components—source (src), destination (dst), is_Vertex (isV) to denote if the entry corresponds to a node or an edge, and data to store the various data items that can be associated with each tuple. A node u is encoded as $(\mathsf{u},\mathsf{u},1,\mathsf{data})$ and a directed edge $\mathsf{e}(\mathsf{u},\mathsf{v})$ is encoded as $(\mathsf{u},\mathsf{v},0,\mathsf{data})$. The DAG-list provides an effective representation of the graph. An undirected graph can also be represented in the DAG-list by converting it into a directed graph where each edge is accounted for twice in both directions.

To enable secure computation of the graph algorithm the framework relies on expressing graph algorithms as message-passing algorithms that operate in multiple iterations. In each iteration, the nodes in the graph—(i) use their state information to send messages over their incident edges; (ii) receive messages along their incident edges and aggregate these messages; (iii) use these messages to update their state. The sending, receiving and updating of state information is realised via the primitives of Scatter, Gather and Apply. Formally, the primitives are defined as follows:

- Scatter: It takes as input a function $f_{\mathsf{AE}} : \{0,1\}^* \rightarrow \{0,1\}^*$ and updates each directed edge $\mathsf{e}(\mathsf{u},\mathsf{v})$ as $\mathsf{e.data} = f_{\mathsf{AE}}(\mathsf{e.data} \,||\, \mathsf{u.data})$.
- Gather: It takes as input a binary aggregation operation $\oplus : \{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^*$ and updates the node data as $\mathsf{v.data} = \mathsf{v.data} \,||\, \oplus_{\forall \mathsf{e}(\mathsf{u},\mathsf{v}) \in \mathsf{E}} \mathsf{e.data}$. Here, $\oplus$ is the iterated binary operation. GraphSC requires $\oplus$ to be commutative and associative. We additionally require that the aggregation operation is linear.
- Apply: It takes as input a user defined function $f_{\mathsf{AV}} : \{0,1\}^* \rightarrow \{0,1\}^*$ and updates each node as $\mathsf{v.data} = f_{\mathsf{AV}}(\mathsf{v.data})$.

Note that for simplicity, the above definition considers Scatter occurring via outgoing edges, and Gather occurring via incoming edges. However, these definitions can be made generic by accounting for a bit b that specifies whether the update occurs over the incoming or outgoing edges during Scatter-Gather. We refer to [22] for the detailed definitions. Secure realisations of these primitives enables secure realisation of a message-passing algorithm that is expressed as a composition of these primitives.

Observe that, naively realising Scatter-Gather while ensuring the topology of the graph remains hidden would require linearly scanning over the DAG-list $|\mathsf{V}|$ many times where the data of edges corresponding to a single node is updated or aggregated in each scan. GraphSC framework realises this efficiently by relying on specific orderings of the DAG-list. A pictorial representation of how Scatter and Gather can be achieved in a single linear scan by leveraging specific orderings of the DAG-list appears in Fig. 1[6]. Observe that by relying on the source order, Scatter can be realised for all edges in a single linear scan of the DAG-list as described in Algorithm 1. Similarly, by relying on the destination order, Gather can be realised for all nodes in a single linear scan of the DAG-list as described in Algorithm 2. Note that the approach of [22] for Scatter-Gather requires performing different operations for entries corresponding to nodes from those corresponding to edges. However, since the information of whether the scanned entry is a node or an edge should be hidden, it necessitates additional interactive multiplication operations when realised via MPC (see algorithm 1 and 2). The GraphSC framework of [22] relies on sort to switch between the source order and destination order each time a Scatter or Gather is applied. Unlike Scatter-Gather, Apply primitive can be computed by processing each entry of the DAG-list in parallel and applying the user-defined function if the element is a vertex. Thus, Apply neither requires a linear scan nor requires any special ordering of the DAG-list. Realising Scatter-Gather-Apply, as described above, securely using MPC enables secure computation of the message-passing algorithms. To summarise, message-passing graph algorithms can be computed iteratively by—(i) sorting based on source order, (ii) Scatter, (iii) sorting based on destination order, (iv) Gather, and (v) Apply. Towards this [22] relies on garbled circuits (GC) to securely realise the primitives. However, note that the GC-based solution, despite an efficient round complexity, has a high overhead due to local computations and communication that dominates the runtime. Thus, to facilitate an improved runtime, [22] describes an optimised solution that aims at reducing the computational overhead. Specifically, [22] assumes a multiprocessor setting and designs a depth optimised circuit for Scatter – Gather primitives that has depth logarithmic in the number of processors. Thus, the efficiency improvements are witnessed by leveraging the presence of multiprocessors that facilitate computations that can be done in parallel.

[6]Recall that Scatter updates edge data as a function of both node data and edge data. However, for simplicity, we consider the function to be an identity function where the edge updates its data to be equal to the data of the source node itself. Similarly, for Gather we consider the aggregation operation to be an addition operation (see §1.1).

Algorithm 1: Scatter of [22]

Input: DAG-list G in source order

```
tmp = 0;
for i = 1 to N do
    G[i].data = tmp + (G[i].data − tmp) · G[i].isV ;
    tmp = tmp + (G[i].x − tmp) · G[i].isV
end
```

Algorithm 2: Gather of [22]

Input: DAG-list G in destination order

```
agg = 0;
for i = 1 to N do
    G[i].data = agg · G[i].isV ;
    agg = (agg + G[i].data) · (1 − G[i].isV)
end
```

Improved GraphSC of [2]. To ensure secure computation over the graph the work of [2], relies on secret-sharing based MPC. In this setting, the input graph represented as the DAG-list is secret-shared among the computing parties of MPC. This ensures that parties cannot distinguish between shares of a tuple corresponding to a vertex and those of an edge. Scatter can be realised in a sequential scan of the DAG-list sorted in source order as described earlier. Similarly, Gather can be realised in a sequential scan of the DAG-list sorted in destination order as described earlier. To transition between different orderings of the DAG-list, a secure sort protocol is required. However, [2] observes that the mapping that takes one ordering to another remains the same across multiple iterations of message-passing phase. Thus, the mappings can be generated in a one-time initialisation phase. Further, instead of relying on a secure sort protocol for generating the mappings, [2] observes that it can be efficiently achieved by replacing the secure sort protocol with a secure shuffle followed by (partially) insecure sort. This approach of [2] is more efficient for transitioning between the different orderings of the DAG-list given the initialisation phase.

Since [2] continues to rely on the approach of [22] for Scatter and Gather, it also requires additional interactive multiplication

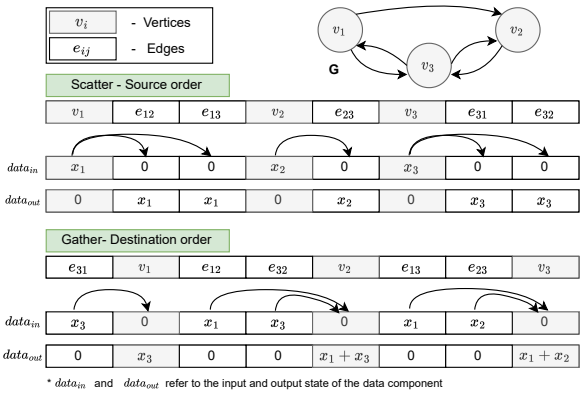


Figure 1: Example of Scatter **and** Gather **in GraphSC.**

operations (see Algorithm 1 and 2). This incurs a round complexity of $O(\mathsf{N})$ when realised using secret-sharing based MPC. Towards addressing this [2] attempts to provide a round optimized (RO) variant of Scatter-Gather by assuming a bound (B) on the maximum degree of the node. The resulting solution continues to have a round complexity $O(\log(\mathsf{B}))$ and a communication cost of $O(\mathsf{N} \cdot \log(\mathsf{B}))$. We would like to note that [2] considers a possibly unrealistic bound of 1023 for a graph of size 10^7. However, we observe that several real world graphs [26] have maximum degree of $O(|\mathsf{V}|)$.

Note that, unlike in [22], [2] has rounds as the bottleneck rather than the computations. Thus, rather than relying on a multiprocessor setting, [2] proposes moving to the secret-sharing based approach, replacing invocations of the secure sort protocol to secure shuffle followed by partially insecure sort and also proposes a round optimised variant for Scatter − Gather towards attaining an improved solution over [22]. We take this one step ahead and improve [2] by designing an improved solution that has round complexity independent of the graph size, where neither rounds nor computations are the bottleneck.

3 Graphiti

Our framework allows secure evaluation of graph algorithms expressed as message-passing algorithms. Here, the message passing algorithm is evaluated iteratively where each iteration consists of the following primitives—(i) Propagate: Propagating information from a source node onto its outgoing edges. (ii) ApplyE: Updating edge data using the information propagated in Propagate. (iii) Gather: Aggregating information from its incoming edges into the destination node. (iv) ApplyV: Updating node data using the information aggregated in Gather. We next formally define these primitives. Following this, we provide the cleartext algorithms for our novel approach of performing Scatter (that consists of Propagate and ApplyE) and Gather. Finally, we showcase how these primitives can be realised securely via MPC. Moreover, we also describe the end-to-end secure Graphiti framework and discuss how the round complexity of the message-passing phase is independent of $|\mathsf{V}|+|\mathsf{E}|$.

3.1 Graphiti **primitives**

Like GraphSC, Graphiti also relies on the DAG-list representation of a data-augmented directed graph $G(\mathsf{V}, \mathsf{E}, \mathsf{data})$. The four primitive operations of Graphiti are as defined below. We note that our primitives are generic as defined in [22] and can also cater to the bit b = '*in/out*' that specifies whether the update occurs over the incoming or outgoing edges of each node. However, as done in [22] for simplicity, we assume that Scatter always occurs via outgoing edges, and Gather always occurs via incoming edges.

1. Propagate: A node propagates data to its outgoing edges. Through this operation, each directed edge is updated as
$$\mathsf{e.data} = \mathsf{e.data} \,||\, \mathsf{u.data} \quad \forall\, \mathsf{e}(\mathsf{u},\mathsf{v}) \in \mathsf{E}$$
2. ApplyE: Through this operation, all edges locally update their data. It takes as input a user-defined function $f_{\mathsf{AE}} : \{0,1\}^* \to \{0,1\}^*$ and updates each edge as
$$\mathsf{e.data} = f_{\mathsf{AE}}(\mathsf{e.data}) \quad \forall\, \mathsf{e}(\mathsf{u},\mathsf{v}) \in \mathsf{E}$$
Observe that the Propagate operation now solely involves the propagation of data in the nodes to its edges. Further, it is the operation ApplyE that allows the edges to locally update the data by applying a user-defined function. Thus, Propagate and ApplyE together encapsulate the original operation of Scatter as described in GraphSC [22]. Further, below definitions of Gather and ApplyV remain same as [22].
3. Gather: A node aggregates the data from incoming edges and updates its data. It takes as input a linear aggregation (see §1.1) operation $\oplus : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ and updates the data on node as
$$\mathsf{v.data} = \mathsf{v.data} \,||\, \oplus_{\forall \mathsf{e}(\mathsf{u},\mathsf{v}) \in \mathsf{E}} \mathsf{e.data} \quad \forall\, \mathsf{e}(\mathsf{u},\mathsf{v}) \in \mathsf{E}$$
4. ApplyV: Through this operation, all nodes locally update their data. It takes as input a user-defined function $f_{\mathsf{AV}} : \{0,1\}^* \to \{0,1\}^*$ and updates each node as
$$\mathsf{v.data} = f_{\mathsf{AV}}(\mathsf{v.data}) \; \forall\, \mathsf{v} \in \mathsf{V}$$

3.2 Scatter

Recall from §2.2 that each entry in DAG-list G is represented by a tuple (src, dst, isV, data). We note that data, the state information stored at each node/edge, can itself can have multiple components, some specific to nodes and some specific to edges. However, to ensure that the topology of the graph is hidden, every entry in the G is associated with all the components. However, only the data component relevant to an entry has valid information, while the other components have 0 or dummy values. Let $\mathsf{G}[i]$ be the i^{th} entry/tuple in the DAG-list. We use $\mathsf{G}[i].\mathsf{data_s}$ to denote the data component that has to be sent/propagated. Similarly, we use $\mathsf{G}[i].\mathsf{data_r}$ to denote the data component that receives the information propagated. Note that at the beginning of Propagate, $\mathsf{G}[i].\mathsf{data_r}$ is initialised to 0 for all entries and $\mathsf{G}[i].\mathsf{data_s} = 0$ if $\mathsf{G}[i]$ corresponds to an edge entry. After Propagate, the $\mathsf{data_s}$ component of every node should be propagated and stored on $\mathsf{data_r}$ component of its outgoing edges, while the $\mathsf{data_r}$ component remains 0 for all nodes.

At a high level, Propagate works as follows. Similar to the approach of [22], consider the DAG-list sorted in source order, where outgoing edges of a node are located immediately after the node. Recall that to realise Scatter, the approach in [22] was to ensure that during a linear scan of the DAG-list, each edge is updated as a function of the data component present on the node preceding it in the source order of the DAG-list. A similar approach can be taken to realise Propagate where the edge is updated as the data component

present on the node preceding it without applying a function. However, to realise this, the Scatter algorithm required differentiating between operations carried out at a node (from where data had to be picked up) and operations carried out at an edge (where the picked-up data had to be dropped off). Instead, we take a different approach where the data dropped off at an edge comprises data present at all the nodes preceding this edge in the source order sorted DAG-list. Observe that this approach can be realised by simply performing a cumulative sum of the data values 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 $\mathsf{G}[i].\mathsf{data_s}$ for an edge is 0, the cumulative sum of all entries preceding an edge correctly realises the above operation. Moreover, cumulative sum being a linear operation, can be realised non-interactively via MPC.

Observe, however, that each edge now possesses sum of the data to be propagated from its source node and the data present on the nodes that appear before it in the source order of the DAG-list. The latter part contributing to the sum needs to be removed. For this, we adjust the $\mathsf{data_s}$ at each node so that the cumulative sum computes the intended data to be propagated. To achieve this, we introduce a new ordering of the DAG-list, called the vertex ordering, which is used to update the data to be propagated by each. A vertex ordering of the DAG-list is an ordering such that all the nodes are sorted as per their indices and appear first, followed by all the edges. We define the vertex order in this way to ensure that the ordering among the nodes remains consistent with their ordering in the source order. This allows us to correctly adjust the values to be propagated by the nodes. Elaborately, given the vertex order of the DAG-list, the value to be propagated by a node 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 $|\mathsf{V}|$[7]. Since this step comprises entirely linear operations, it can also be performed non-interactively within MPC. Having computed the updated data to be propagated in the vertex order, the computations in the source order to propagate the correct data involve computing $\mathsf{G}[i].\mathsf{data_r} = \sum_{j=1}^{i-1} \mathsf{G}[j].\mathsf{data_s}'$ for $i = 1$ to N. Finally, note that transitioning between a vertex order and source order of DAG-list requires one invocation of $\mathcal{F}_{\mathsf{Shuffle}}$, akin to transition between source and destination order in [2] as elaborated in §3.4. An example of the computations performed on the vertex order and source order of the DAG-list appears in Fig. 2.

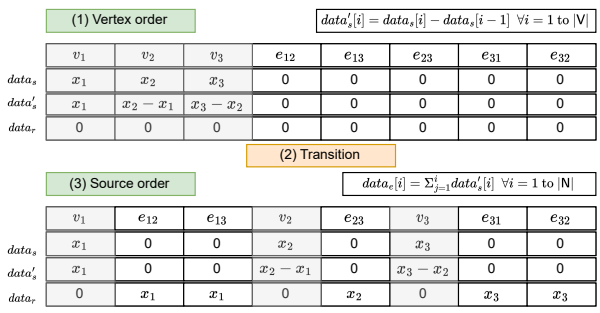


Figure 2: Example for Propagate **in** Graphiti **for the graph** G **considered in Fig. 1.**

[7]Although it appears that knowledge of $|\mathsf{V}|$ is required, it can be avoided (see §A.1).

The formal steps for Propagate appear in Algorithm 3. The steps in the algorithm account for optimisations such as—(1) Not requiring to maintain an explicit $\mathsf{G}[i].\mathsf{data_s}'$ component, which can instead be computed as part of $\mathsf{G}[i].\mathsf{data_r}$. (2) Since $\mathsf{G}[i].\mathsf{data_r}$ is used for propagating data as well as accumulating the propagated data, it necessitates performing a reverse linear scan of the DAG-list in the source order. (3) To ensure that the $\mathsf{G}[i].\mathsf{data_r}$ component of a node becomes 0 after Propagate, the computation to be performed on the source ordered list is $\mathsf{G}[i].\mathsf{data_r} = \sum_{j=1}^{i} \mathsf{G}[j].\mathsf{data_r} - \mathsf{G}[i].\mathsf{data_s}$ instead of just $\mathsf{G}[i].\mathsf{data_r} = \sum_{j=1}^{i} \mathsf{G}[j].\mathsf{data_r}$.

Algorithm 3: Propagate

Input: DAG-list G in vertex order
1 **for** $i = |\mathsf{V}|$ *to* 2 **do**
2 $\quad \mathsf{G}[i].\mathsf{data_r} = \mathsf{G}[i].\mathsf{data_s} - \mathsf{G}[i-1].\mathsf{data_s}$;
3 **end**
4 Transition to the source order of the DAG-list ;
5 **for** $i = \mathsf{N}$ *to* 1 **do**
6 $\quad \mathsf{G}[i].\mathsf{data_r} = \sum_{j=1}^{i} \mathsf{G}[j].\mathsf{data_r} - \mathsf{G}[i].\mathsf{data_s}$;
7 **end**

Looking ahead, note that in the secure realisation of Algorithm 3 using MPC, step 2 and step 6, which consist of only linear operations, can be realised non-interactively. Thus, the only cost incurred is for the transition from vertex order of the DAG-list to source order, which can be achieved using a single invocation $\mathcal{F}_{\mathsf{Shuffle}}$ as described in §3.4. Finally, note that to achieve the effect of Scatter as per the original definition of [22], the operations described above are followed by the ApplyE primitive. Having scattered the data from nodes onto the outgoing edges, the next step is to gather this scattered information into the respective nodes. Details of how this can be achieved are described next.

3.3 Gather

After Propagate and ApplyE, every edge has the scattered data stored in the $\mathsf{data_r}$ component, while the same is empty for a node. During Gather, each node aggregates $\mathsf{data_r}$ from its incoming edges using an aggregation operation denoted as $\oplus$. For ease of explanation, we consider this operation to be an addition operation. However, note that this can be any linear aggregation function. We use $\mathsf{G}[i].\mathsf{data_g}$ to denote the data component of an entry that stores information aggregated as a part of Gather. Note that at the beginning of Gather, $\mathsf{G}[i].\mathsf{data_g}$ is initialised to 0 for all the entries. After Gather, the $\mathsf{data_g}$ component of a node stores the aggregation of the $\mathsf{data_r}$ component of its incoming edges while the $\mathsf{data_g}$ component of an edge contains dummy values.

Our algorithm for Gather begins similar to the algorithm of [22], wherein the DAG-list is present in the destination order. In this order, entries corresponding to all the incoming edges of each node precede the node entry in the DAG-list. To aggregate information into a node, a linear scan of this 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. That is, the linear scan computes $\mathsf{G}[i].\mathsf{data_g}' = \sum_{j=1}^{i-1} \mathsf{G}[j].\mathsf{data_r}$ across all entries i in the DAG-list.

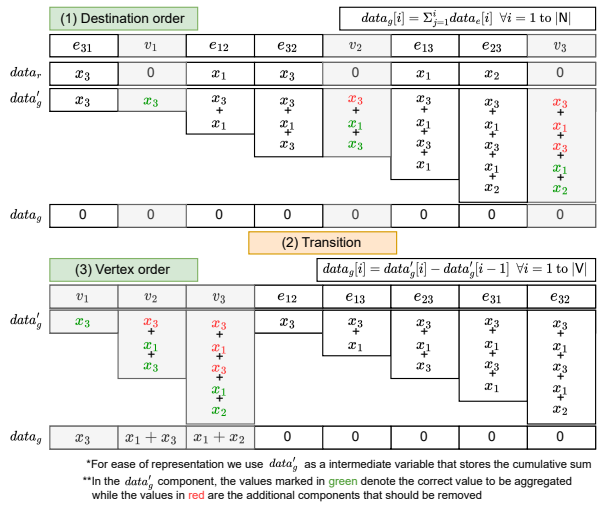


Figure 3: Example for Gather **in** Graphiti **for the same graph** G **given in Fig. 1.**

Like in Propagate, the cumulative sum only requires linear operations, which facilitates realising it non-interactively within MPC. Next, due to the cumulative sum, observe that each node aggregates not only the data_r component from its incoming edges but also the data_r component accumulated by the preceding entries in DAG-list. To ensure that the correct information is aggregated, it is required to remove this additional information that is gathered by each node. For this, observe that this additional information aggregated by a node is indeed the data aggregated by the preceding node in the DAG-list. Thus, transitioning to the vertex order of the DAG-list allows each node to remove this additional information and compute the correct data_g component by computing $\text{G}[i].\text{data}_\text{g} = \text{G}[i].\text{data}_\text{g}' - \text{G}[i-1].\text{data}_\text{g}'$ for $i = 1$ to N. Observe that this also comprises entirely of linear operations, and the transition from destination ordering to vertex ordering can be done using $\mathcal{F}_{\mathsf{Shuffle}}$. A pictorial representation of gather is given in Fig. 3.

The formal steps for Gather appear in Algorithm 4. The steps in the algorithm account for the optimisations of not maintaining an explicit $\text{G}[i].\text{data}_\text{g}'$ component. Instead it can be computed as part of $\text{G}[i].\text{data}_\text{g}$. However, this necessitates performing a reverse linear scan of the DAG-list in the vertex order. Looking ahead, note that in the secure realisation of Algorithm 4 using MPC, step 2 and step 6, which consist of only linear operations, can be realised non-interactively. Thus, the only cost incurred is the transition from the destination order of the DAG-list to the vertex order, which can be achieved using a single invocation $\mathcal{F}_{\mathsf{Shuffle}}$ as described in §3.4.

Finally, Gather is followed by the ApplyV primitive, which marks the completion of one round of the message-passing phase. Unlike [2], where ApplyV is realised by applying the function f_{AV} on all entries of DAG-list and obliviously updating only the vertices, Graphiti can leverage the knowledge of |V| to realise ApplyV more efficiently. 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} can be applied only to the first |V| entries of the DAG-list that corresponds to the vertices.

Algorithm 4: Gather

Input: DAG-list G in destination order
1 **for** $i = 1$ *to* N **do**
2 $\quad \text{G}[i].\text{data}_\text{g} = \sum_{j=1}^{i} \text{G}[j].\text{data}_\text{r}$;
3 **end**
4 Transition to the vertex order of the DAG-list;
5 **for** $i = \text{N}$ *to* 2 **do**
6 $\quad \text{G}[i].\text{data}_\text{g} = \text{G}[i].\text{data}_\text{g} - \text{G}[i-1].\text{data}_\text{g}$;
7 **end**

3.4 The complete Graphiti framework

Graphiti securely evaluates a graph algorithm by invoking multiple iterations of the message-passing phase. Each iteration starts with performing a Propagate as described in Algorithm 3, which entails performing computations on the DAG-list in the vertex order, followed by a transition to the source order. Once information is propagated, an invocation of ApplyE 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. Following this, we perform a Gather as defined in Algorithm 4, which entails transitioning from the source order to the destination order, performing computations on the destination order, and transitioning to the vertex order. Once information is gathered, an invocation of ApplyV ensures that data on each node in the DAG-list is updated under the function $f_{\mathsf{AV}}(\cdot)$. This constitutes one iteration of the message-passing phase.

Observe that unlike [22], our solution additionally requires a vertex order of the DAG-list. However, each iteration of the message-passing phase in our solution starts and ends on the vertex order. Hence, when evaluating multiple message-passing rounds, we do not require additional transitions to switch between any of the orderings, as required in [22] to move from the destination order to the source order. Further, since the mapping used in the transitions remain the same across multiple message-passing rounds, as done in the work of [2], we generate these mappings in a one-time initialisation phase. We next provide details of this initialisation phase, followed by the message-passing phase.

3.4.1 Initialisation phase. The initialisation of GraphSC [22] comprises generating secret shares of mappings that allow transitioning between different orderings of the DAG-list. While these mappings can be generated naively by relying on a secure sort, the work of [2] showcased efficiency improvements that can be achieved by instead relying on a secure shuffle followed by an insecure sort (§2.2). Our solution continues to use this shuffle-then-sort paradigm.

The initialisation phase of [2] begins with a secret-sharing of the DAG-list that is randomly ordered. However, it is unclear how such a randomly ordered DAG-list can be generated, even as a part of input sharing. Instead, we observe that the vertex ordering of DAG-list is a more naturally occurring input state, and hence, we begin our initialisation phase with vertex ordering of the DAG-list. To this end, we also design an input sharing phase in Graphiti (see §A.2), which enables users who only have a partial view of a global graph to secret share their view of the graph such that all the computing parties hold secret shares of the DAG-list in the vertex

order. Given this, steps in our initialisation follow on similar lines as that of [2] except that we additionally require the generation of mappings that allow transitioning to/from vertex order.

Steps involved in initialisation. Given the DAG-list in the vertex order, the parties invoke $\mathcal{F}_{\mathsf{Shuffle}}$ to apply a random permutation π_A on the DAG-list to generate a random ordering of the DAG-list, denoted as Shuffle-A. The parties then invoke $\mathcal{F}_{\mathsf{Shuffle}}$ to apply another random permutation π_B on Shuffle-A to generate Shuffle-B. The parties invoke $\mathcal{F}_{\mathsf{insecSort}}$ to compute a mapping from the Shuffle-A to the source order. We denote this mapping as π_S. Observe that, since this mapping merely maps a random permutation of the DAG-list to a sorted list, it leaks no information about the relation between the vertex order and the source order. Hence, the mapping π_S can be made public. Similarly, the parties invoke $\mathcal{F}_{\mathsf{insecSort}}$ on Shuffle-B to compute a public mapping from the Shuffle-B to the destination order. This mapping is denoted as π_D. Note that the steps for initialisation are similar to those described in [2] except that we begin with a vertex ordering of the DAG-list instead of a random ordering. We refer to [2] for additional details such as the need for Shuffle-A, Shuffle-B, etc. A pictorial illustration of initialisation phase appears in Fig. 7 in §A.3.

Transition between different orderings. Given secret shared permutations π_A, π_B and public permutations π_S and π_D that are generated during the initialisation phase, the transition between different orderings during the message-passing phase can be achieved as illustrated in Fig. 4. Formal details appear in §A.3.1.

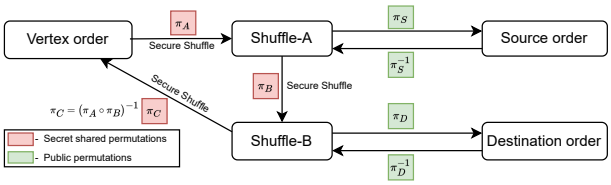


Figure 4: Transition between orderings of DAG-list **in** Graphiti**.**

Complexity. Starting with vertex ordered DAG-list, two sequential invocations of $\mathcal{F}_{\mathsf{Shuffle}}$ are required to generate Shuffle-A and Shuffle-B, along with sharing of permutations π_A and π_B. Then, two parallel invocations of $\mathcal{F}_{\mathsf{insecSort}}$ are required to generate mappings corresponding to source and destination order. Finally, a single invocation of $\mathcal{F}_{\mathsf{Shuffle}}$ in parallel is required to generate sharing of π_C. Thus, initialisation requires two invocations of shuffle followed by parallel invocations of two $\mathcal{F}_{\mathsf{insecSort}}$ and $\mathcal{F}_{\mathsf{Shuffle}}$.

3.4.2 Message-passing phase. Each iteration of the message-passing phase consists of a secure evaluation of Propagate (Algorithm 3), ApplyE, Gather (Algorithm 4) and ApplyV. Securely realising Propagate and Gather involves transitioning between orderings of the DAG-list, which can be done using an invocation to $\mathcal{F}_{\mathsf{Shuffle}}$, followed by local operations. The secure realisation of ApplyE and ApplyV can be performed using MPC. A pictorial illustration of steps in one iteration of message-passing phase appears in Fig. 5. Observe that securely realising a graph algorithm boils down to designing the above mentioned primitives with respect to the graph algorithm under consideration. These further rely on $\mathcal{F}_{\mathsf{Shuffle}}$, $\mathcal{F}_{\mathsf{insecSort}}$ and the underlying MPC protocol as required for realising the functions in ApplyE and ApplyV. Hence, given the cleartext algorithm for the aforementioned primitives, one can rely on an appropriate MPC protocol to attain the desired level of security. In this way, Graphiti is a generic framework and can be instantiated with an MPC protocol of choice. The formal details, along with their security proof, appear in §D. Additionally, details of securely evaluating contact tracing via Graphiti as an example is described in §A.4.

Complexity. The protocol begins with DAG-list sorted in vertex order. Propagate, involves secure evaluation of Algorithm 3 using MPC. Observe that in the secure evaluation, steps 1-3 consist of only linear operations and hence can be evaluated non-interactively. Following this, the parties invoke $\mathcal{F}_{\mathsf{Shuffle}}$ to apply π_A, then apply the public permutation π_S on their local shares of DAG-list to generate the source order. Finally, steps 5-7 also consist of only linear operations and hence can be evaluated non-interactively. Parties evaluate ApplyE in parallel by securely computing the function f_{AE} on the appropriate data components of each entry in DAG-list. To transition from the source order to the destination order, the parties locally apply the public permutation π_S^{-1} to generate Shuffle-B. The parties invoke one instance of $\mathcal{F}_{\mathsf{Shuffle}}$, which applies π_B on Shuffle-A and generate Shuffle-B. The parties locally apply the public permutation π_D to get the destination order. During Gather, parties run the MPC protocol to evaluate Gather described in algorithm 4. In the secure evaluation of Algorithm 4, observe that steps 1-3 consist of only linear operations and hence can be evaluated non-interactively. Following this, parties apply the public permutation π_D^{-1} on their local shares, followed by one invocation of $\mathcal{F}_{\mathsf{Shuffle}}$ to apply π_C to generate the vertex order. Finally, steps 5-7 also consist of only linear operations and hence can be evaluated non-interactively. Parties evaluate ApplyV in parallel by securely computing the function f_{AV} on the appropriate data components of the first $|\mathsf{V}|$ entries in the DAG-list. In this way, the round complexity of the message-passing phase is independent of the length of the DAG-list and is constant in the $\mathcal{F}_{\mathsf{Shuffle}}$-hybrid model.

4 (2+1)-Shuffle

We design a secure shuffle protocol in the 2PC setting with a helper that is secure against a semi-honest adversary. The 2PC shuffle protocol of [9] when instantiated with a helper forms the state-of-the-art and has an online complexity of 2 rounds and 2N elements of communication. Our shuffle protocol brings down the round complexity to just 1. An elaborate discussion on shuffle protocols in other settings is deferred to §B. We begin by describing the ideal functionality for our shuffle protocol, followed by the secure protocol for the same.

Let $\mathsf{T} \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$ be a vector of N elements, where each element is drawn from $\mathbb{Z}_{2^\ell}$. Let T be additively shared among the parties P_0, P_1, i.e., every element of T is additively shared among P_0, P_1 such that $P_i \in \{P_0, P_1\}$ holds $\langle \mathsf{T} \rangle_i \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$ and $\mathsf{T} = \langle \mathsf{T} \rangle_0 + \langle \mathsf{T} \rangle_1$. Secure shuffle takes as input $\langle \cdot \rangle$-shares of T and generates $\langle \cdot \rangle$-shares of $\mathsf{T}_\mathsf{O} \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$ where T_O represents the shuffled vector T under a random secret permutation π, i.e., $\mathsf{T}_\mathsf{O}[i] = \mathsf{T}[\pi(i)]$. We use $\mathsf{T}_\mathsf{O} = \pi(\mathsf{T})$ to denote the operation of shuffling T under a random permutation π to obtain T_O. Ideal functionality for secure shuffle appears in Fig. 9[8] in §B.

Observe that to perform a secure shuffle operation, the permutation, π, used for shuffling should be hidden from the parties. Hence,

[8] Graphiti requires shuffling the DAG-list which can be viewed as a vector, where each element is now a tuple of elements instead of being a single element from $\mathbb{Z}_{2^\ell}$.

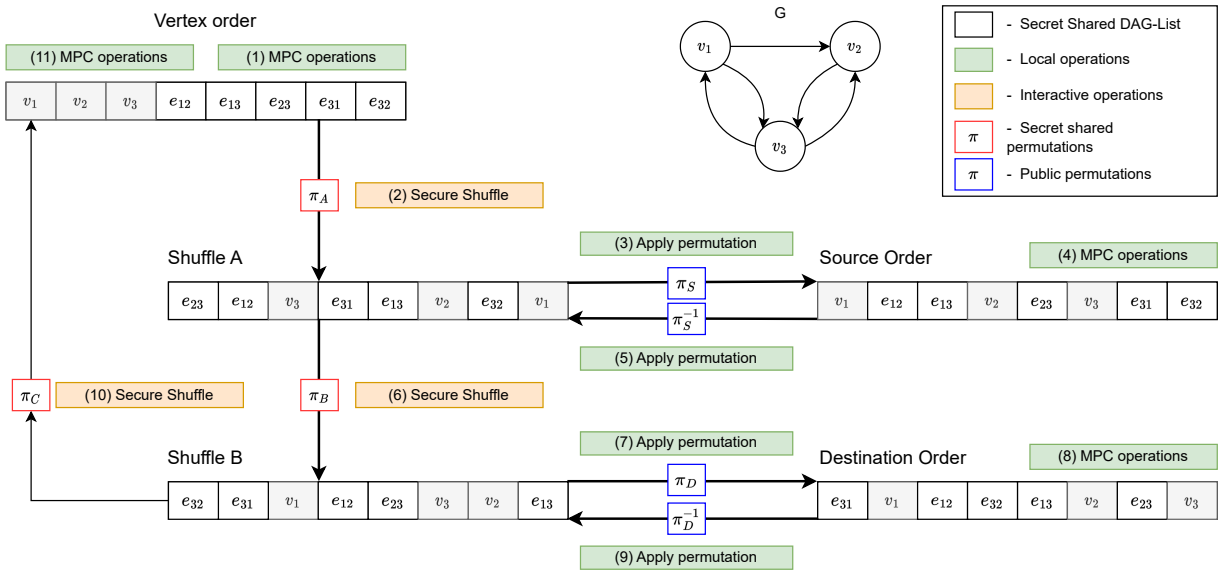


Figure 5: Example for message-passing round in Graphiti.

we define π to be a composition of two random permutations π_0 and π_1, i.e., $\pi = \pi_0 \circ \pi_1$, such that $P_i \in \{P_0, P_1\}$ only holds π_i. Note that P_i can generate this random permutation π_i non-interactively (see §2.1). Given this, to compute $\langle\cdot\rangle$-shares of $\mathsf{T_O} = \pi(\mathsf{T})$, we make the following observation.

$$\mathsf{T_O} = \pi(\mathsf{T}) = \pi\left(\langle\mathsf{T}\rangle_0 + \langle\mathsf{T}\rangle_1\right) = \pi\left(\langle\mathsf{T}\rangle_0\right) + \pi\left(\langle\mathsf{T}\rangle_1\right)$$

Thus, one can define the $\langle\cdot\rangle$-shares of $\mathsf{T_O}$ as $\langle\mathsf{T_O}\rangle_0 = \pi\left(\langle\mathsf{T}\rangle_1\right)$ (to be held by P_0) and $\langle\mathsf{T_O}\rangle_1 = \pi\left(\langle\mathsf{T}\rangle_0\right)$ (to be held by P_1). However, observe that defining the shares in this way allows P_1 to learn $\pi\left(\langle\mathsf{T}\rangle_0\right)$ which is the permuted share of the input held by P_0 and similarly for P_0. Hence, to ensure that no information about the other party's input share is leaked, we define the $\langle\cdot\rangle$-shares of $\mathsf{T_O}$ by masking both $\pi\left(\langle\mathsf{T}\rangle_0\right)$ and $\pi\left(\langle\mathsf{T}\rangle_1\right)$ with a vector of random values $\mathsf{R} \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$. That is,

$$\mathsf{T_O} = \pi\left(\langle\mathsf{T}\rangle_0\right) + \pi\left(\langle\mathsf{T}\rangle_1\right) = \underbrace{\left(\pi\left(\langle\mathsf{T}\rangle_0\right) - \mathsf{R}\right)}_{\langle\mathsf{T_O}\rangle_1} + \underbrace{\left(\pi\left(\langle\mathsf{T}\rangle_1\right) + \mathsf{R}\right)}_{\langle\mathsf{T_O}\rangle_0}$$

Thus, our goal boils down to generating $\langle\mathsf{T_O}\rangle_0 = \pi\left(\langle\mathsf{T}\rangle_1\right) + \mathsf{R}$ towards P_0 and generating $\langle\mathsf{T_O}\rangle_1 = \pi\left(\langle\mathsf{T}\rangle_0\right) - \mathsf{R}$ towards P_1. At a high-level, to generate $\langle\mathsf{T_O}\rangle_0$ towards P_0, the approach is to first generate $\langle\mathsf{T_O}\rangle_0' = \pi\left(\langle\mathsf{T}\rangle_1 + \mathsf{R}_1\right)$ towards P_0, where $\mathsf{R}_1 \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$ is not known to P_0. Similarly, $\langle\mathsf{T_O}\rangle_1' = \pi\left(\langle\mathsf{T}\rangle_0 + \mathsf{R}_0\right)$ can be generated towards P_1. To ensure that the same randomness is used as a mask in each $\langle\mathsf{T_O}\rangle_i$ for $i \in \{0,1\}$, P_i then non-interactively computes $\langle\mathsf{T_O}\rangle_i = \langle\mathsf{T_O}\rangle_i' - \mathsf{B}_i$, where $\mathsf{B}_0 = \pi(\mathsf{R}_1) - \mathsf{R}$ and $\mathsf{B}_1 = \pi(\mathsf{R}_0) + \mathsf{R}$. Observe that since B_i comprises of values that are independent of the input, it can be computed and made available to P_i for $i \in \{0,1\}$ in the preprocessing phase. We next elaborate on these steps, where we begin by discussing how $\langle\mathsf{T_O}\rangle_0$ can be generated towards P_0, followed by discussing generation of $\langle\mathsf{T_O}\rangle_1$ towards P_1.

Generating $\langle\mathsf{T_O}\rangle_0$ towards P_0. Recall that $\langle\mathsf{T_O}\rangle_0 = \pi\left(\langle\mathsf{T}\rangle_1\right) + \mathsf{R}$. Consider the term $\pi\left(\langle\mathsf{T}\rangle_1\right) = \pi_0\left(\pi_1\left(\langle\mathsf{T}\rangle_1\right)\right)$. Observe that to compute $\pi\left(\langle\mathsf{T}\rangle_1\right)$, P_0 misses π_1 and $\langle\mathsf{T}\rangle_1$. Both these are held by P_1, who can compute and send $\mathsf{A}_0 = \pi_1(\langle\mathsf{T}\rangle_1 + \mathsf{R}_1)$ to P_0. Here, $\mathsf{R}_1 \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$ is a random vector sampled by P_1 to mask $\pi_1, \langle\mathsf{T}\rangle_1$ from P_0. On receiving A_0, P_0 can compute $\pi_0(\mathsf{A}_0)$ to obtain $\langle\mathsf{T_O}\rangle_0' = \pi\left(\langle\mathsf{T}\rangle_1 + \mathsf{R}_1\right)$. Next, to generate $\langle\mathsf{T_O}\rangle_0 = \pi\left(\langle\mathsf{T}\rangle_1\right) + \mathsf{R}$, the approach is to make available towards P_0 the vector $\mathsf{B}_0 = \pi(\mathsf{R}_1) - \mathsf{R}$. While the process for generation of B_0 is described later, observe here that computing $\langle\mathsf{T_O}\rangle_0' - \mathsf{B}_0 = \pi\left(\langle\mathsf{T}\rangle_1\right) + \mathsf{R}$ allows P_0 to obtain $\langle\mathsf{T_O}\rangle_0$.

Generating $\langle\mathsf{T_O}\rangle_1$ towards P_1. An analogous approach, as described above, does not help in generating $\langle\mathsf{T_O}\rangle_1 = \pi\left(\langle\mathsf{T}\rangle_0\right) - \mathsf{R}$ towards P_1. This is because unlike the value $\pi\left(\langle\mathsf{T}\rangle_1\right) = \pi_0\left(\pi_1\left(\langle\mathsf{T}\rangle_1\right)\right)$ in $\langle\mathsf{T_O}\rangle_0$, where P_0 was missing both the inner terms π_1 and $\langle\mathsf{T}\rangle_1$, the term $\pi\left(\langle\mathsf{T}\rangle_0\right) = \pi_0\left(\pi_1\left(\langle\mathsf{T}\rangle_0\right)\right)$ in $\langle\mathsf{T_O}\rangle_0$ does not have this structure. Elaborately, although P_0 holds π_0 and $\langle\mathsf{T}\rangle_0$, note that the application of π_0 on $\langle\mathsf{T}\rangle_0$ is preceded by the application of π_1. Since the composition of permutations is not commutative, generating $\langle\mathsf{T_O}\rangle_1'$ towards P_1 in a single round is challenging. To enable the generation of $\pi\left(\langle\mathsf{T}\rangle_0\right)$ (masked with randomness) towards P_1 in a single round, we let the parties generate another pair of permutations π_0', π_1' such that $\pi = \pi_0 \circ \pi_1 = \pi_1' \circ \pi_0'$. Ensuring $\pi = \pi_1' \circ \pi_0'$ allows to perform analogous steps towards P_1 to generate $\mathsf{A}_1 = \pi_0'\left(\langle\mathsf{T}\rangle_0 + \mathsf{R}_1\right)$ and thereby $\langle\mathsf{T_O}\rangle_1$ using π_1', π_0', as done to generate $\langle\mathsf{T_O}\rangle_0$ towards P_0 using π_0, π_1. Elaborately, P_0 computes and sends $\mathsf{A}_1 = \pi_0'(\langle\mathsf{T}\rangle_0 + \mathsf{R}_0)$ to P_1, where $\mathsf{R}_1 \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$ serves as a vector of random masks to hide $\pi_0', \langle\mathsf{T}\rangle_0$ from P_1. Party P_1 can then compute $\langle\mathsf{T_O}\rangle_1' = \pi_1'(\mathsf{A}_0) = \pi\left(\langle\mathsf{T}\rangle_0\right) + \pi(\mathsf{R}_0)$. To generate $\langle\mathsf{T_O}\rangle_1$ we make available towards P_1 the vector $\mathsf{B}_1 = \pi_0'(\mathsf{R}_0) + \mathsf{R}$. This allows P_1 to compute $\langle\mathsf{T_O}\rangle_1' - \mathsf{B}_1 = \pi\left(\langle\mathsf{T}\rangle_0\right) - \mathsf{R}$ and obtain $\langle\mathsf{T_O}\rangle_1$.

Generating input-independent data. We now discuss how the various terms such as $\pi_i, \pi_i', \mathsf{R}_i$ for $i \in \{0,1\}$ and $\mathsf{B}_0 = \pi(\mathsf{R}_1) - \mathsf{R}$, $\mathsf{B}_1 = \pi'(\mathsf{R}_0) + \mathsf{R}$ can be generated with the help of a trusted party P_2 in the preprocessing phase. First, parties P_i, P_2 for $i \in \{0,1\}$ sample a random permutation π_i over $\{1, \ldots, \mathsf{N}\}$, non-interactively. This defines $\pi = \pi_0 \circ \pi_1$. However, the second pair of permutations π_0', π_1' should be defined such that $\pi = \pi_1' \circ \pi_0' = \pi_0 \circ \pi_1$. To realize

this, parties P_0 and P_2 randomly sample π'_0 non-interactively. Given π'_0, observe that $\pi'_1 = \pi \circ \pi'^{-1}_0$ where π and π'_0 are both known to P_2. Thus P_2 computes π'_1 and sends it to P_1. Observe that P_2 learns the entire π on clear. Hence, to hide the permutation from P_2, after generating $\langle \mathsf{T_O} \rangle$, P_0, P_1 sample a random permutation π_2 that is not known to P_2 and locally permute the shares of $\mathsf{T_O}$. Finally, note that R_i can be sampled non-interactively by parties P_i, P_2 for $i \in \{0, 1\}$. Following this, P_2 computes and sends $\mathsf{B}_0 = \pi(\mathsf{R}_1) - \mathsf{R}$ to P_0 and $\mathsf{B}_1 = \pi'_0(\mathsf{R}_0) + \mathsf{R}$ to P_1. The formal protocol for secure shuffle appears in Fig. 10 in §B.

Complexity. We next discuss the round and communication complexity of our protocol and also compare it with the semi-honest shuffle protocol of [9]. Observe that the online phase involves a single round of interaction where each party communicates one message of $\mathsf{N}\ell$ bits. The preprocessing phase also involves a single round of interaction where P_2 communicates three messages of size $\mathsf{N}\ell$. Thus, the overall cost of the $\Pi_{(2+1)\text{-Shuffle}}$ is two rounds and $5\mathsf{N}\ell$ bits of communication. A detailed comparison of our shuffle protocol with that of [9] appears in §B.

5 Benchmarks

We empirically evaluate the performance of Graphiti framework and compare it with the state-of-the-art GraphSC framework of [2]. We instantiate the underlying MPC using the MPC framework of [11]. Here, we offload input-independent computations to a preprocessing phase where the helper party carries out the computations. We implement all the protocols from scratch in C++ using the code base of [30]. We note that our code[9] is developed for benchmarking, is not optimised for industry-grade use. We perform the benchmarks over LAN on Ubuntu servers equipped with AMD Ryzen Threadripper PRO 5965WX and 256GB RAM. Each party is run as a process on the same machine. We simulate the network connection using the Linux tc command. We consider a bandwidth of 1 Gbps and 0.5 ms of latency. We note that our code is not multithreaded, and all experiments are run on a single thread. For all our experiments we consider the online run time and online communication as the parameters for benchmark. For completeness, we report the preprocessing cost in §C. We first provide a performance comparison of Graphiti with the framework of [2], followed by benchmarking the performance of our shuffle protocol against that of [9]. With respect to GraphSC, note that although [2] relied on a 3-party honest majority setting for showcasing performance, the techniques therein are generic and can be instantiated with any MPC. Hence, to draw a fair comparison with Graphiti and showcase the performance improvement brought in by our techniques to realise Scatter-Gather, we instantiate Graphiti as well as the techniques of [2] in the above mentioned 2-party setting with a helper. Similarly, with respect to shuffle, to draw a fair comparison, we adapt the protocol of [9] in the 2-party with a helper setting.

5.1 Graphiti

We consider the two variants of GraphSC provided by [2] for comparison– (i) linear variant that has round and communication complexity of $O(\mathsf{N})$ for performing Scatter/Gather and (ii) round optimised (RO) variant that has better round complexity of $O(\log(|\mathsf{V}|))$ albeit requiring a higher communication complexity of $O(\mathsf{N}\log(|\mathsf{V}|))$ for Scatter/Gather[10]. Recall that protocols in Graphiti and GraphSC requires $\mathcal{F}_{\mathsf{Shuffle}}$ and $\mathcal{F}_{\mathsf{insecSort}}$. For this, we instantiate the $\mathcal{F}_{\mathsf{Shuffle}}$ with $\Pi_{(2+1)\text{-Shuffle}}$ and $\mathcal{F}_{\mathsf{insecSort}}$ with the secure protocol for quicksort as described in [2]. We note that the run time of our protocol is independent of the graph topology and depends only on N and |V|. We set 10% of N as nodes in our experiments. We use similar graph sizes (N) as considered in [2] for all the comparisons. In what follows, we first discuss the performance of Scatter-Gather, followed by benchmarking two applications of contact tracing via BFS, using both Graphiti as well as [2].

5.1.1 Improvements in Scatter/Gather. We begin by comparing the cost of the primitives in Graphiti. Fig. 6 and Table 8 report the comparison of Scatter where $\mathsf{N} = |\mathsf{V}| + |\mathsf{E}|$ is varied from 10^4 to 10^7. Recall that Scatter updates the edge data as a function of node data. For simplicity, we consider this function to be an identity function where the edge data is updated to be equal to the data component of the source node. We make the following observation:

- Our protocol for Scatter clearly outperforms both the linear and the RO variant of [2], with respect to run time. Concretely, we see improvements of up to 1418× and 116× in run time for linear and RO variants, respectively. The improvements in run time can be attributed to the improved round, communication, as well as computation cost of our Scatter. With only one invocation of $\Pi_{(2+1)\text{-Shuffle}}$, our protocol requires just a single round, as opposed to the N and log(|V|) number of rounds for the linear and RO variant, respectively. Further, our protocol has 4× and up to 160× less communication with respect to the linear variant and RO variant, respectively. This is because our protocol for Scatter has a communication of $2\mathsf{N}\ell$. In comparison, the linear variant has 2N invocations of secure multiplication where each multiplication has a cost of 4ℓ bits, making the online communication $8\mathsf{N}\ell$. The RO variant has the highest communication wherein it requires $4\mathsf{N}\log(|\mathsf{V}|)$ invocations of secure multiplication and hence has a communication of $16\mathsf{N}\log(|\mathsf{V}|)$. Additionally, our local computations involve just additions/subtractions as opposed to 4N and $8\mathsf{N}\log(|\mathsf{V}|)$ local multiplications in the linear and RO variant of [2]. All these factors contribute to the improved run time of our solution.
- With respect to scalability with N, we observe that the run time of Scatter for both Graphiti, as well as the the two variants of [2] increases with N. However, the rate of increase for our protocol is much smaller in comparison to that of [2]. Specifically, as N increases from 10^4 to 10^7, we observe that our run time increases by a factor of 435× while for both the linear variant and the RO variant of [2] if increases by a factor of approximately 1002× and 2575×. The increase in the run time of our Scatter can be attributed to the increase in the communication cost, which scales linearly with N. However, as N increases, along with communication, the round complexity also increases for both the linear and the RO variant of [2]. Hence, they both have a higher factor of increase in run time. In fact, the RO variant has the highest factor of increase

[9] https://github.com/Bhavishrg/Graphiti

[10] Recall that the RO variant of [2] has round and communication complexity of $O(\log \mathsf{B})$ and $O(\mathsf{N}\log \mathsf{B})$, respectively. Here, B denotes the bound on the maximum degree. For experiments, [2] considers as low a bound as B = 1023 even for graphs of size 10^7. However, we observe that several real-world graphs have a maximum degree as $O(|\mathsf{V}|)$ [26]. Hence, we set the bound B = |V|.

as its communication scales at an increased factor of $O(N \log(|V|))$. Consequently, our improvements in run time with respect to both the linear and the RO variant of [2] increase with increasing N. Concretely, we see the improvements increase from 263× to 1418× for the linear variant and from 17× to 116× for the RO variant. In this way, our solution is much more scalable and takes less than a second to process the DAG-list of 1 million entries.

• Fig. 11 in §C.1 captures how all the protocols compare with each other. The improvements in the run time of the RO variant with respect to the linear variant decrease from 30× to 10× when increasing N from 10^4 to 10^7. This can be attributed to the higher communication complexity of the RO variant that starts becoming a bottleneck as N increases. This clearly shows that the RO variant trades off communication to achieve better run time. In contrast, note that our protocol reduces both the round and communication complexity, making it a clear-cut winner, as evident from Fig. 11.

The comparison for Gather is appears in Table 9 in §C.1. Since the trend for Gather follows similarly, we omit explicit analysis for the same.

5.1.2 BFS for Contact tracing. To showcase the practicality of Graphiti for real life applications, we benchmark the two variants of contact tracing described in §A.4 via Graphiti and compare it with the GraphSC framework of [2].

The application of simple contact tracing requires τ iterations of message-passing phase, where each iteration comprises the following components— (i) Scatter, (ii) transition from source ordering to destination ordering, (iii) Gather. Additionally, only the last iteration of message-passing phase comprises an ApplyV, which in turn relies on a secure comparison protocol [11]. Table 5 and Table 10 report the run time and communication split, respectively. The reported costs account for the various components in one iteration of the message-passing phase. The total time for simple contact tracing is computed as $\tau \times$ (time for Scatter + time for Transition + time for Gather) + time for ApplyV.

Observe that our framework clearly outperforms [2]. Specifically, we see improvements of up to 1034× and 85× in run time, and up to 3× and 106× in the communication cost, with respect to the linear and RO variant of [2], respectively. This can be attributed to the improvements witnessed in Scatter – Gather and the ApplyV. We do not explicitly report the improvements with respect to the Scatter and Gather components since they follow similar trends, as reported earlier. Further, we note that the cost for transition remains the same across all protocols. Finally, we see an improvement of up to 11× in run time and up to 10× in the communication of ApplyV. This is because, as discussed in §3.3, Graphiti leverages the vertex order and the knowledge of |V| to improve the communication of ApplyV wherein the function f_{AV} is applied only to first |V| entries of the DAG-list. Unlike this, for both the linear and RO variants of [2], the function f_{AV} is applied to all the entries of the DAG-list. We observe that it takes under 2 minutes to perform simple contact tracing on a graph of size 10^7. This shows that our solution is sufficiently practical. For completeness, we also report the cost of the initialisation phase for Graphiti in Table 11. Note that the cost of initialisation for Graphiti is the same as that of [2].

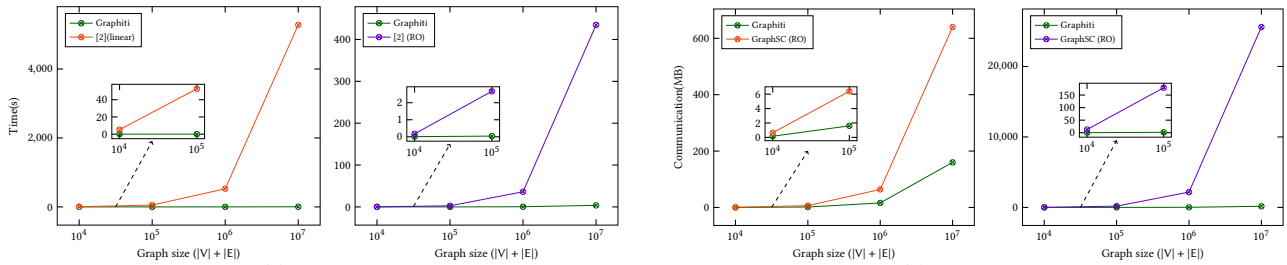
[11] We rely on circuit-based comparison protocol[21] which can be realised using MPC.

Ref	N	Scatter	Transition	Gather	ApplyV	Total
[2] (linear)		5.27		5.27	0.01	105.45
[2] (RO)	10^4	0.16	0.01	0.17	0.01	3.35
Graphiti		0.01		0.01	0.01	0.18
[2] (linear)		52.65		52.66	0.07	1053.43
[2] (RO)	10^5	2.73	0.02	2.77	0.07	55.34
Graphiti		0.04		0.04	0.01	1.13
[2] (linear)		526.02		526.68	0.71	10529.79
[2] (RO)	10^6	36.25	0.18	36.46	0.71	730.54
Graphiti		0.40		0.42	0.07	11.00
[2] (linear)		5271.83		5271.84	8.17	105464.60
[2] (RO)	10^7	434.78	1.71	433.27	8.17	8716.64
Graphiti		3.66		3.67	0.69	101.90

Table 5: Comparison of run time(s) of simple contact tracing for threshold $\tau = 10$ and varying $N = |V| + |E|$.

Effect of decoupling Scatter. Recall that the simple contact tracing considered above does not have the ApplyE component. However, several other graph algorithms require a specific function to be applied during Scatter. In such scenarios, recall that our approach of decoupling Scatter into Propagate and ApplyE aids in improving the round complexity. We benchmark the application of probabilistic contact tracing to showcase these improvements. Each iteration of the message passing phase here additionally requires a function f_{AE} to be applied during Scatter (see Equation (1)). The same can be realised in Graphiti via Propagate and ApplyE where propagate remains the same as in the case of simple contact tracing, while ApplyE is now defined as $e.data_r = f_{\mathsf{AE}}(e.data_r) \forall e(u, v) \in E$. The rest of the message-passing phase proceeds in the same way as in simple contact tracing. Note that in the secure evaluation, f_{AE} can be realised by sampling a random value r and computing $f_{\mathsf{AE}}(data) = data \cdot \mathbf{1}\{r < p\}$ where $\mathbf{1}\{\cdot\}$ denotes the indicator function. Observe that securely computing f_{AE} requires one invocation of secure comparison followed by one invocation of multiplication. Thus, securely computing f_{AE} incurs a round complexity of $\log(\ell) + 1$ when relying on circuit-based comparison protocol and MPC of [11]. Here ℓ denotes the input length of r.

As stated in §1.1, recall that one can attain an intermediate solution that immediately improves over [2] by decoupling Scatter as Propagate and ApplyE. Specifically, for the application of probabilistic contact tracing, the intermediate solution improves the round complexity of Scatter in the linear variant of [2] from $O(N \cdot (\log(\ell)))$ to $O(N + \log(\ell))$ and from $O(\log(|V|) \cdot (\log(\ell)))$ to $O(\log(|V|) + \log(\ell))$ for the RO variant. Graphiti not only decouples Scatter but also designs a novel approach that further improves the round complexity to $O(\log(\ell))$. Table 6 reports the cost of probabilistic contact tracing for the linear and RO variants of [2], Graphiti, as well as the intermediate solutions with decoupled Scatter. Concretely, the intermediate solutions witness an improvement up to 6× in comparison to the linear variant of [2] and up to 8.5× improvement in comparison to the RO variant of [2]. Observe that decoupling of Scatter has a higher impact on the RO variant than the linear variant since decoupling aids in improving not only the rounds but also the communication complexity in the case of RO variant. Specifically, the communication complexity drops from $O(N \cdot \log(|V|) \cdot \ell)$ to $O(N \cdot \log(|V|) + N \cdot \ell)$ for the RO variant. Finally, to capture the improvements brought in by the novel approach to Scatter, we compare Graphiti with the intermediate solution. Here, Graphiti



(a) Run time (b) Communication

Figure 6: Comparison of run time and communication of Scatter for varying graph size.

witnesses an improvement of 1000× and 27× over the decoupled Scatter solution for linear and RO.

Ref	Scatter		Total
	Propagate	ApplyE	
[2] (linear)	62.92		682.01
[2] (RO)	2.48		26.55
[2] (linear with decoupled Scatter)	10.47	0.02	157.42
[2] (RO with decoupled Scatter)	0.27		4.57
Graphiti	0.01	0.02	0.325

Table 6: Comparison of the run time (s) for the application of contact tracing for probabilistic infection spread for $N = 10^4$.

5.1.3 Other applications: As done in [22], we note that Graphiti can be used to securely realise other applications such as matrix factorization, Pagerank, histogram, etc. Regarding the concrete run time of Graphiti for these applications, we note that the computations involved in these are very similar to that of BFS, described in §5.1.2. Specifically, steps within Propagate and Gather follow along similar lines as in the case of BFS. Further, ApplyE and ApplyV functions are linear and can be performed locally within MPC. Thus, the run times for these applications are dominated by the run times for Propagate and Gather, where the run times for the latter will follow similar trends as in Table 8 and Table 9. Hence, we do not benchmark these additional applications explicitly.

5.2 Shuffle

We compare the performance of our newly designed shuffle protocol with the state-of-the-art protocol of [9]. Since [9] works in the 2PC setting, for a fair comparison, we adapt and implement the protocol in the 2PC with a helper setting.

We begin by comparing our shuffle protocol for the case of multiple invocations of shuffle where the same random permutation is applied, as required when considering multiple iterations of message-passing phase in Graphiti. Table 7 reports the comparison of the online phase of shuffle where we vary the number of shuffle invocations from 1 to 100. Observe that our shuffle protocol outperforms [9] in terms of run time. Specifically, we see improvements of up to 1.83× in the run time while having the same communication cost. The improvements in run time can be attributed to the improvements in the round complexity of our protocol wherein the protocol of [9] requires one additional round compared to our protocol. For completeness, we also compare the total run time and communication of our protocol and report the cost in appendix §C (see Table 16). Despite having slightly higher communication in the preprocessing our protocol outperforms the protocol of [9] in terms of run time. This shows that the effect of the lower number of rounds dominates the increase in communication. Further, we observe that as the number of sequential shuffle invocations increases, our improvements in the total time increase as our communication cost gets amortised. This shows that our shuffle protocol is apt for scenarios where the same random permutation has to be applied multiple times. Next, to showcase scalability, we benchmark the performance of our shuffle protocol for varying input vector sizes. Table 17 in §C.2 reports the costs when varying the input vector size from 10^4 to 10^7. We observe that the run time of our shuffle increases linearly with the increase in vector size. Further, it takes less than 2 seconds to shuffle a vector of ten million elements of length 64 bits each. This shows that our shuffle protocol is highly scalable, and this greatly impacts the performance of Graphiti.

Ref	#Shuffle	Run time (ms)	Comm. (MB)
[9]	1	21.37	1.6
Ours		17.93	
[9]	10	124.68	16.00
Ours		77.20	
[9]	50	644.95	80.00
Ours		346.01	
[9]	100	1253.75	160.00
Ours		681.63	

Table 7: Comparison of shuffle for multiple shuffle invocations with $|T| = 10^5$ when same permutation is applied.

6 Acknowledgements

Arpita Patra would like to acknowledge financial support from Sony Faculty Innovation Award, Google India Faculty Award and JPM Faculty Research Award. The project also received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850990 PSOTI). It was co-funded by the Deutsche Forschungsgemeinschaft (DFG) within SFB 1119 CROSSING/236615297.

References

[1] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. 2016. High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority. Cryptology ePrint Archive, Paper 2016/768. https://eprint.iacr.org/2016/768

[2] Toshinori Araki, Jun Furukawa, Kazuma Ohara, Benny Pinkas, Hanan Rosemarin, and Hikaru Tsuchida. 2021. Secure Graph Analysis at Scale. In *ACM CCS*.

[3] Gilad Asharov, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Ariel Nof, Benny Pinkas, Katsumi Takahashi, and Junichi Tomida. 2022. Efficient secure three-party sorting with applications to data analysis and heavy hitters. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*.
[4] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. 1988. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract). In *STOC*.
[5] Elette Boyle, Nishanth Chandran, Niv Gilboa, Divya Gupta, Yuval Ishai, Nishant Kumar, and Mayank Rathee. 2021. Function secret sharing for mixed-mode and fixed-point secure computation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer.
[6] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2019. Secure computation with preprocessing via function secret sharing. In *Theory of Cryptography: 17th International Conference, TCC 2019, Nuremberg, Germany, December 1–5, 2019, Proceedings, Part I 17*. Springer.
[7] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. 2019. Practical Fully Secure Three-Party Computation via Sublinear Distributed Zero-Knowledge Proofs. In *ACM CCS*.
[8] Andreas Brüggemann, Thomas Schneider, Ajith Suresh, and Hossein Yalame. 2022. Poster: Efficient Three-Party Shuffling Using Precomputation. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 3331–3333.
[9] Melissa Chase, Esha Ghosh, and Oxana Poburinnaya. 2020. Secret-shared shuffle. In *ASIACRYPT*.
[10] Harsh Chaudhari, Ashish Choudhury, Arpita Patra, and Ajith Suresh. 2019. ASTRA: High Throughput 3PC over Rings with Application to Secure Prediction. In *ACM CCSW@CCS*.
[11] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY-A framework for efficient mixed-protocol secure two-party computation.. In *NDSS*.
[12] Ronald Aylmer Fisher and Frank Yates. 1953. *Statistical tables for biological, agricultural, and medical research*. Hafner Publishing Company.
[13] Pranav Jangir, Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, Bhavish Raj Gopal, and Somya Sangal. 2023. Vogue: Faster computation of private heavy hitters. *IEEE Transactions on Dependable and Secure Computing* (2023).
[14] Banashri Karmakar, Nishat Koti, Arpita Patra, Sikhar Patranabis, Protik Paul, and Divya Ravi. 2023. Asterisk: Super-fast MPC with a Friend. *Cryptology ePrint Archive* (2023).
[15] Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, and Bhavish Raj Gopal. 2023. Entrada to Secure Graph Convolutional Networks. *Cryptology ePrint Archive* (2023).
[16] Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, Bhavish Raj Gopal, et al. 2023. Find thy neighbourhood: Privacy-preserving local clustering. *PoPETs* (2023).
[17] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. 2021. SWIFT: Super-fast and Robust Privacy-Preserving Machine Learning. In *USENIX Security*.
[18] Donghang Lu and Aniket Kate. 2022. RPM: Robust Anonymity at Scale. *PETS* (2022).
[19] Sahar Mazloom and S Dov Gordon. 2018. Secure computation with differentially private access patterns. In *CCS*.
[20] Sahar Mazloom, Phi Hung Le, Samuel Ranellucci, and S Dov Gordon. 2020. Secure parallel computation on national scale volumes of data. In *USENIX Security*.
[21] Payman Mohassel and Peter Rindal. 2018. ABY^3: A Mixed Protocol Framework for Machine Learning. In *ACM CCS*.
[22] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. 2015. GraphSC: Parallel secure computation made easy. In *IEEE S&P*.
[23] Arpita Patra and Ajith Suresh. 2020. BLAZE: Blazing Fast Privacy-Preserving Machine Learning. In *NDSS*.
[24] A Pranav Shriram, Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, Bhavish Raj Gopal, and Somya Sangal. 2023. Ruffle: Rapid 3-party shuffle protocols. *PoPETS* (2023).
[25] M Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M Songhori, Thomas Schneider, and Farinaz Koushanfar. 2018. Chameleon: A hybrid secure computation framework for machine learning applications. In *Proceedings of the 2018 on Asia conference on computer and communications security*.
[26] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. https://networkrepository.com
[27] Théo Ryffel, Pierre Tholoniat, David Pointcheval, and Francis Bach. 2020. Ariann: Low-interaction privacy-preserving deep learning via function secret sharing. *arXiv preprint arXiv:2006.04593* (2020).
[28] Sameer Wagh, Divya Gupta, and Nishanth Chandran. 2019. SecureNN: 3-Party Secure Computation for Neural Network Training. *Proc. Priv. Enhancing Technol.* 2019 (2019).
[29] Hanzhi Wang, Mingguo He, Zhewei Wei, Sibo Wang, Ye Yuan, Xiaoyong Du, and Ji-Rong Wen. 2021. Approximate graph propagation. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*.
[30] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. 2016. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit.
[31] Andrew Chi-Chih Yao. 1982. Protocols for Secure Computations (Extended Abstract). In *FOCS*.

A Graphiti

A.1 Scatter

Knowedge of |V|. Recall that our Scatter algorithm relies on knowing |V| when processing the DAG-list in vertex order. Note that this is required to ensure that the $\mathsf{data_r}$ component of edges are 0, which is required to ensure correctness. This is in contrast to the GraphSC framework of [22], which only assumes the knowledge of |V| + |E|. However, we highlight that our algorithm can be adapted to function without knowing |V|, while still maintaining a round complexity independent of |V| + |E|. Consider the current algorithm for Scatter (Algorithm 3), where knowledge of |V| is required in steps 1-3. The algorithm can be adjusted to run the for loop in steps 1-3 for i from |V| + |E| to 2. Subsequently, edges can locally update their $\mathsf{data_r}$ component to 0. This can be achieved by multiplying the $\mathsf{data_r}$ component with the isV component. The secure realisation of this using MPC requires one invocation of secure multiplication at each entry of the DAG-list. Note that this operation is akin to ApplyV, where operations are local to the entries of the DAG-list and hence can be executed in parallel. Further, since secure multiplication via MPC can be realised in constant rounds, the round complexity of the algorithm remains independent of |V| + |E|. The rest of the protocol can then proceed as previously described. The modified algorithm that does not require the knowledge of |V| appears in Algorithm 5.

Algorithm 5: Scatter

Input: DAG-listG in vertex order
for $i = |V| + |E|$ *to* 1 **do**
  $\mathrm{G}[i].\mathsf{data_r} = \mathrm{G}[i].\mathsf{data_s} - \mathrm{G}[i-1].\mathsf{data_s}$;
end
for $i = |V| + |E|$ *to* 1 *in parallel* **do**
  $\mathrm{G}[i].\mathsf{data_r} = \mathrm{G}[i].\mathsf{data_r} \cdot \mathrm{G}[i].\mathsf{isV}$;
end
Transition to the source order of the DAG-list ;
for $i = |V| + |E|$ *to* 1 **do**
  $\mathrm{G}[i].\mathsf{data_r} = \sum_{j=1}^{i} \mathrm{G}[j].\mathsf{data_r} - \mathrm{G}[i].\mathsf{data_s}$;
end

The secure implementation of Algorithm 5 now additionally requires |V| + |E| parallel invocations of $\mathcal{F}_{\mathsf{mult}}$. Thus the round complexity of Scatter is reduces to the round complexity of one invocation of $\mathcal{F}_{\mathsf{mult}}$ followed by $\mathcal{F}_{\mathsf{Shuffle}}$.

A.2 Input sharing

Recall that the input to the initialisation phase of the Graphiti framework consists of the graph represented as a DAG-list G in the vertex order. In this section, we discuss the input-sharing phase, which details the inputs shared by the clients (who distributively hold the graph) and the subsequent generation of the $\langle\cdot\rangle$-shares of the DAG-list G by the computing parties.

Recall that in applications such as contact tracing, the input graph can be distributed between multiple clients who only have a

partial view of the graph. The partial view of a client consists of a subset of vertices, the data associated with them and the edges corresponding to those vertices. Thus each client secret shares the following to the computing parties- (i) an edge list of size |V| with respect to each vertex u in its veiw, where the i^{th} entry in the list corresponds to a 1 if the the client has the view of the edge $e(u, i)$ and (ii) a list of data values corresponding to each vertex in the view of the clients. Having received the shares of input, assuming the mapping between clients and ordering of the vertices is known, the computing parties do the following. The parties first generate the adjacency matrix, A, of the graph by stacking up the edge list corresponding to each vertex. Further, they also generate a data list, X, corresponding to the vertices by concatenating the list of data values received from the clients.

Following this, the parties perform a series of consistency checks to ensure that they have received valid shares that indeed correspond to an adjacency matrix. This involves checking the consistency of the input shares generated by the client towards the parties, checking if each entry in A is a 0/1, checking if A is symmetric (in case the input is an undirected graph) etc. Further details of these consistency checks can be found in [15].

Given $\langle\cdot\rangle$-shares of A, X, the parties proceed to generate DAG-list in vertex order. This entails generating $\langle\cdot\rangle$-shares of—(i) G$[i]$.src, G$[i]$.dst (ii) G$[i]$.isV to denote if the i^{th} tuple is a vertex or an edge, (iii) G$[i]$.data to store the data elements, where the data components of vertices are present in X and data components of edges are initialised to 0. The parties first initialize G and set the first |V| entries of the G as vertices ordered from 1 to |V|. Thus the parties set the $\langle$G$[i]$.isV$\rangle$ to $\langle 1\rangle$, $\langle$G$[i]$.src$\rangle$ = $\langle$G$[i]$.dst$\rangle$ = $\langle i\rangle$ and $\langle$G$[i].data\rangle$ = $\langle X[i]\rangle$ for i = 1 to |V|. Next to generate the entries of G corresponding to valid edges, the parties first generate $\langle\cdot\rangle$-sharing of a list G′ comprising of all possible edges (i.e. every element in A) where $\langle$G′$[i]$.isV$\rangle$ is set $\langle \mathsf{A}_{ij}\rangle$, and data component is set to $\langle 0\rangle$. Next, to extract the valid |E| edges from $|\mathsf{V}|^2$ entries in G′, [15] relies on sorting G′ based on the values in isV, and extracts the first |E| entries, and appends these to G. However, we observe that instead of sorting, one can rely on a compaction protocol [3, 13] to efficiently extract the E valid edges. Finally, isV should be 1 only for vertices thus, values of isV are set to 1 − isV for the valid edges in G′ before appending them to G.

A.3 The complete framework

A.3.1 Initialisation. The transitions during initialisation can be achieved as follows given the mappings that are generated during the initialisation phase as discussed in §3.4.1.

- *Vertex order → source order:* The parties invoke $\mathcal{F}_{\mathsf{Shuffle}}$ to apply the secret shared permutation π_A to move from vertex order to Shuffle-A. Then, the parties locally apply the public permutation π_S on their share of Shuffle-A to get the source ordered DAG-list. Thus, a single invocation of $\mathcal{F}_{\mathsf{Shuffle}}$ is required to transition from vertex order to source order.
- *Source order → destination order:* The parties locally apply the public permutation π_S^{-1} move from source order to Shuffle-A. Note that since π_S is a public permutation, π_S^{-1} can be computed locally by the computing parties. Following this, the parties invoke $\mathcal{F}_{\mathsf{Shuffle}}$ to apply the permutation π_B to move to Shuffle-B. Finally, the parties locally apply the public permutation π_D on their local shares of Shuffle-B to get the DAG-list in destination order. Thus, a single invocation of $\mathcal{F}_{\mathsf{Shuffle}}$ is required to transition from source order to destination order.
- *Destination order → vertex order:* The parties locally apply the public permutation π_D^{-1} on their shares to get to Shuffle-B from destination order. From here, observe that two invocations of $\mathcal{F}_{\mathsf{Shuffle}}$ are required to get to vertex ordering, i.e., apply π_B^{-1} to get Shuffle-A followed by π_A^{-1} to get vertex order from Shuffle-A. However, the sharing of $\pi_C = \pi_B^{-1} \circ \pi_A^{-1}$ among the parties can be computed once using one call to $\mathcal{F}_{\mathsf{Shuffle}}$ during the initialisation and can be used later in the message-passing phases. Thus, applying the public permutation π_D^{-1} followed by a single invocation to $\mathcal{F}_{\mathsf{Shuffle}}$ to apply π_C generates the vertex order of the DAG-list from the destination order.

A pictorial representation of the initialisation phase of Graphiti appears in Fig. 7.

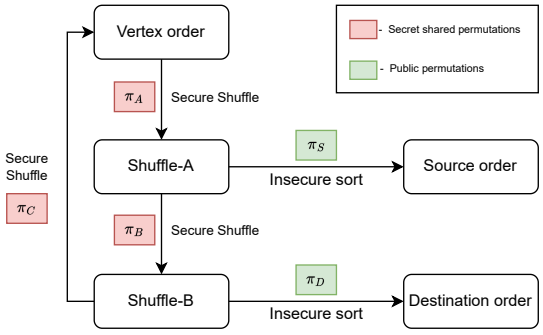


Figure 7: Initialisation in Graphiti**.**

A.4 Contact tracing using Graphiti

In this section we discuss the two variants of contact tracing that and how it an be realised using the framework of Graphiti.

Simple contact tracing. Recall the application of contact tracing described earlier, where the goal was to identify all persons at a distance smaller than some threshold τ from an infected person in the contact graph. A simple way of achieving this is to run a BFS algorithm. We demonstrate how our framework can be used to realise the BFS algorithm, which is also a basic building block for many other applications. Specifically, given a graph, we are interested in identifying all nodes that are reachable within a given distance τ from a source node via BFS. We assume the input graph is represented as the DAG-list, sorted in vertex order. This DAG-list consists of nodes and edges, encoded as a tuple (src, dst, isV, data_s, data_r, data_g) (see §2.2).

At a high level, the message-passing phase can be realised as follows. Following the initialisation phase described in §3.4.1, the parties evaluate the BFS algorithm in τ message-passing rounds. Initially, the source node, representing an infected person, has the data_s component set to 1, and all other entries in the DAG-list have have data_s set to 0. data_r, data_g components for all entries in the DAG-list are initialized to 0 as well. In the i^{th} message-passing round, our goal is to identify nodes that are reachable within a distance of i hops from the source node. To realize this, in each of

the τ message-passing rounds, for each edge $e(u, v)$, if the $data_s$ component of u is a 1, then we set the $data_s$ component of v also to 1. To achieve this, Propagate is defined to propagate the $data_s$ component of a node onto its outgoing edges. The propagated data is stored in the $data_r$ component of the edges. An explicit call to ApplyE is not required. Then, Gather is defined to compute a cumulative sum of the $data_r$ values of all incoming edges, and is stored in the $data_g$ component of the node. Observe that if $data_g > 0$ for a v, then its $data_s$ component should be set to 1. This can done in the ApplyV phase where each vertex updates its $data_s$ to 1 if $data_g > 0$. In this way, observe that after τ message-passing rounds, $data_s = 1$ for all nodes that are within a distance of at most τ from the source node. Further, note that as an optimisation, we can avoid checking if $data_g > 0$ in each message-passing round and perform this check only once in the last round. That is, instead of checking if $data_g > 0$ in each message-passing round, $data_g$ can be simply be added to the $data_s$ component. In this case, at the end of τ rounds, $data_s \geq 1$ for all nodes that are within a distance at most τ from the source node. Thus, in the ApplyV phase of the last message-passing round, $data_s$ can be updated to 1 if $data_s > 0$. The formal details of performing simple contact tracing appears in Fig. 8.

Inputs: The parties hold sharing of DAG-list G in vertex order. Each tuple in the DAG-list consists of the following components— (src, dst, isV, $data_s$, $data_r$, $data_g$). The $data_s$, $data_r$, $data_g$ component is initialised to 0 except for the source node which has $data_s$ set to 1. A public threshold τ

Output: The parties hold sharing of DAG-list G in vertex order such that the $data_s$ component of all the infected nodes is set to 1.

Protocol:

- The parties run the initialisation steps described in §3.4 to generate sharing of permutations π_A, π_B, π_C and public permutations π_S, π_D.
- for $i = 0$ to τ the parties do the following:
  - **Propagate:** The parties invoke $\mathcal{F}_{\mathsf{MPC}}$ to evaluate steps 1-3 of Propagate described in algorithm 3. Following this, the parties invoke $\mathcal{F}_{\mathsf{Shuffle}}$ to apply π_A followed by applying the public permutation π_S on their local shares of DAG-list to generate the source order. Finally, the parties invoke $\mathcal{F}_{\mathsf{MPC}}$ to evaluate steps 5-7 of algorithm 3.
  - An explicit call to ApplyE is not required.
  - **source order to destination order:** The parties locally apply the public permutation π_A^{-1} to generate sharing of shuffle-B. The parties invoke one instance of $\mathcal{F}_{\mathsf{Shuffle}}$ which applies π_B on shares of shuffle-A and generate sharing of shuffle-B. The parties locally apply the public permutation π_D to get the destination order.
  - **Gather:** The parties run the MPC protocol to evaluate Gather described in algorithm 4. In the secure evaluation of Algorithm 4, observe that steps 1-3 consist of only linear operations and hence can be evaluated non-interactively. Following this, the parties apply the public permutation π_D^{-1} on their local shares followed by one invocation of $\mathcal{F}_{\mathsf{Shuffle}}$ to apply π_C to generate the vertex order. Finally, steps 5-7, also consist of only linear operations and hence can be evaluated non-interactively.
  - Set $G[j].data_s = G[j].data_g$ for $j = 0$ to V.
  - An explicit call to ApplyV is not required.
- **ApplyV:** The parties invoke $\mathcal{F}_{\mathsf{MPC}}$ on the circuit computing the function f_{AV} on the sharing of $data_s$ component of the first $|V|$ entries in G. Here, f_{AV} is defined as $f_{\mathsf{AV}}(\text{data}) = 1\{\text{data} > 0\}$.

Figure 8: Secure evaluation of simple contact tracing using Graphiti**.**

Probabilistic contact tracing. A more fine grained analysis of contact tracing is also possible, when we associate a probability with which the infection can spread. In this scenario, an infected node spreads the infection to its neighbor with a probability p. The goal is to trace the spread of infection at a distance τ from a source node. The message-passing round for the same can be realised in a similar manner to that of simple contact tracing with Scatter modified to update the edge data as a function of the source node data, i.e., $e.data_r = f_{\mathsf{AE}}(u.data_s) \forall e(u, v) \in E$ where:

$$f_{\mathsf{AE}}(\text{data}) = \begin{cases} f_{\mathsf{AE}}(\text{data}) & \text{with probability } p \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

The same can be realised in Graphiti via Propagate and ApplyE where propagate is defined as propagating the $data_s$ component of a node onto its outgoing edges, while ApplyE is now defined as $e.data_r = f_{\mathsf{AE}}(e.data_r) \forall e(u, v) \in E$. The rest of the message-passing round proceeds in the same way as BFS.

B (2+1)-Shuffle

Shuffle related works. We would like to note that the work of [8] also designs a shuffle protocol in the setting of 2PC with a helper, and requires 1 round and 2N elements of communication in the online phase. However, it requires a function-dependent preprocessing phase, unlike our shuffle protocol whose preprocessing phase is function-independent. Moreover, it relies on permutation matrices, which makes the computation cost quadratic in the number of nodes in the graph. This defeats the goal of Graphiti, which is to reduce the quadratic complexity. [1] also designs a shuffle protocol in the 3PC (all-online) setting, which requires 3 rounds of interaction and 6N elements of communication. When adapted to a 2PC setting, it requires 2 rounds of interaction and 4N elements of communication. Finally, the 2PC shuffle protocol of [9] is designed in the function-independent preprocessing setting. However, its online complexity is 2 rounds and 2N elements of communication. We design a new shuffle protocol that can bring down the round complexity to just 1.

The ideal functionality of shuffle appears in Fig. 9 and the protocol box appears in Fig. 10.

Comparison with shuffle protocol of [9]. The protocol of [9], originally designed for 2PC setting, can be adapted to our setting by offloading the preprocessing computations to the helper. The approach taken in [9] is to compute $\langle\cdot\rangle$-shares of $\mathsf{T_O} = \pi(\mathsf{T}) = \pi_0(\pi_1(\mathsf{T}))$ in two rounds, where the first round involves generating $\langle\cdot\rangle$-shares of $\mathsf{T}' = \pi_1(\mathsf{T})$, followed by the generation of $\langle\cdot\rangle$-shares of $\mathsf{T_O} = \pi_0(\mathsf{T}')$. Each round in this protocol involves communicating one message of size $N\ell$. In comparison, our protocol follows a completely different approach by relying on two pairs of permutation π_i, π_i' for $i \in \{0, 1\}$ to compute $\pi(\mathsf{T})$ in only one round.

Comparing the preprocessing phases, the protocol from [9] requires two messages of size $N\ell$ communicated by the helper in one

round, while our protocol requires the communication of three messages of size $N\ell$. However, we would like to note that in applications such as Graphiti, where there are multiple invocations of shuffle and the same permutation is reused across these shuffle instances, the cost of generating the permutations π'_0, π'_1 gets amortized. This makes the overall cost of $\Pi_{(2+1)\text{-Shuffle}}$ almost the same as that of [9].

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

Without loss of generality, let $P_c \in \mathcal{P}$ denote the party corrupted by adversary $\mathcal{S}$. $\mathcal{F}_{\text{Shuffle}}$ interacts with parties in $\mathcal{P}$ and $\mathcal{S}$. It receives as input $\langle\cdot\rangle$-shares of the input table T from P_0, P_1.
$\mathcal{F}_{\text{Shuffle}}$ proceeds as follows.

- Reconstruct input T using $\langle\cdot\rangle$-shares of P_0, P_1.
- Sample a random permutation π from the space of all permutations, S_N and generate $\mathsf{T_O} = \pi(\mathsf{T})$.
- Generate random $\langle\cdot\rangle$-sharing of $\mathsf{T_O}$.
- Send (Output, $\langle \mathsf{T_O} \rangle_i$) to P_i for $i \in \{0, 1\}$.

Figure 9: Ideal functionality for shuffle.

Protocol $\Pi_{(2+1)\text{-Shuffle}}$

Inputs: Additive shares, $\langle \mathsf{T} \rangle_0, \langle \mathsf{T} \rangle_1$ of the table T with N rows to be shuffled shared between parties P_0, P_1.
Outputs: Additive shares, $\langle \mathsf{T_O} \rangle_0, \langle \mathsf{T_O} \rangle_1$ of the table $\mathsf{T_O} = \pi(\mathsf{T})$ shared between parties P_0, P_1, where T is shuffled under a random secret permutation π.

Preprocessing:

- Parties P_i, P_2 for $i \in \{0, 1\}$ non-interactively samples $\mathsf{R}_i \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$.
- Parties P_i, P_2 for $i \in \{0, 1\}$ non-interactively sample a random permutation π_i over $\{1, \ldots, \mathsf{N}\}$. Define $\pi = \pi_0 \circ \pi_1$.
- Parties P_0, P_2 non-interactively sample a random permutation π'_0 over $\{1, \ldots, \mathsf{N}\}$.
- P_2 computes and sends $\pi'_1 = \pi \circ {\pi'}_0^{-1}$ to P_1.
- P_2 randomly samples $\mathsf{R} \in \mathbb{Z}_{2^\ell}^{\mathsf{N}}$ and sends $\mathsf{B}_0 = \pi(\mathsf{R}_1) - \mathsf{R}$ to P_0 and $\mathsf{B}_1 = \pi'_0(\mathsf{R}_0) + \mathsf{R}$ to P_1.

Online:

Round 1

- P_1 computes and sends $\mathsf{A}_0 = \pi_1(\langle \mathsf{T} \rangle_1 + \mathsf{R}_1)$ to P_0.
- P_0 computes and sends $\mathsf{A}_1 = \pi'_0(\langle \mathsf{T} \rangle_0 + \mathsf{R}_0)$ to P_1.

Local computation

- P_0 computes $\langle \mathsf{T_O} \rangle'_0 = \pi_0(\mathsf{A}_0)$ and $\langle \mathsf{T_O} \rangle_0 = \langle \mathsf{T_O} \rangle'_0 - \mathsf{B}_0$.
- P_1 computes $\langle \mathsf{T_O} \rangle'_1 = \pi'_1(\mathsf{A}_1)$ and and $\langle \mathsf{T_O} \rangle_1 = \langle \mathsf{T_O} \rangle'_1 - \mathsf{B}_1$.
- P_i for $i \in \{0, 1\}$ locally sample a random permutation π_2 and set $\langle \mathsf{T_O} \rangle_i = \pi_2(\langle \mathsf{T_O} \rangle_i)$.

Figure 10: Secure shuffle protocol for two-party setting.

C Benchmarks

C.1 Graphiti

Optimisation: Recall that each entry in the DAG-list has multiple components namely, (src, dst, isV, data). Further recall that the message passing phase of Graphiti involves transitions between different orderings of the DAG-list. Therefore, all the components of the DAG-list must be reordered during these transitions. However, observe that the src, dst and isV components of an entry in the DAG-list remains unchanged throughout the message passing phase. Thus we generate copies of these components of the DAG-list in different orderings as a part of initialisation and maintain them separately. During the message passing phase, only the necessary data components are reordered to transition between different ordering of the DAG-list as required for Scatter and Gather. This helps in reducing the communication cost as the number of components in the DAG-list that has to be reordered is reduced.

Table 8 and Table 9 reports the comparison of Scatter and Gather respectively for varying N. Fig. 11 reports the comparison of Scatter in Graphiti with both the variants of [2] in log-log scale.

Ref	N	Runtime (s)	Comm. (MB)
[2] (linear)		5.26	0.64
[2] (RO)	10^4	0.17	12.8
Graphiti		0.01	0.16
[2] (linear)		52.66	6.4
[2] (RO)	10^5	2.67	179.2
Graphiti		0.04	1.6
[2] (linear)		526.25	64
[2] (RO)	10^6	36.16	2176
Graphiti		0.39	16
[2] (linear)		5278.41	640
[2] (RO)	10^7	434.78	25600
Graphiti		3.72	160

Table 8: Comparison of Scatter **for varying N**= |V| + |E|.

Ref	N	Runtime (s)	Comm. (MB)
[2] (linear)		5.27	0.64
[2] (RO)	10^4	0.17	12.8
Graphiti		0.01	0.16
[2] (linear)		52.61	6.4
[2] (RO)	10^5	2.77	179.2
Graphiti		0.04	1.6
[2] (linear)		526.68	64
[2] (RO)	10^6	36.46	2176
Graphiti		0.42	16
[2] (linear)		5271.84	640
[2] (RO)	10^7	433.27	25600
Graphiti		3.67	160

Table 9: Comparison of Gather **for varying N**= |V| + |E|.

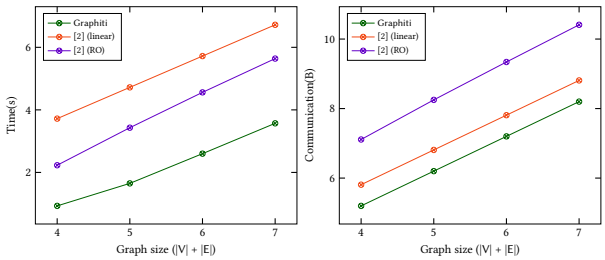


Figure 11: Run time(left) and communication (right) of Scatter **for varying graph size. Plots are log-log plots with both x-axis and y-axis logarithmic in base 10.**

Ref	N	Scatter	Transition	Gather	ApplyV	Total
[2] (linear)		0.64		0.64	0.32	14.72
[2] (RO)	10^4	12.8	0.16	12.8	0.32	257.92
Graphiti		0.16		0.16	0.03	4.83
[2] (linear)		6.4		6.4	3.15	147.15
[2] (RO)	10^5	179.2	1.6	179.2	3.15	3603.15
Graphiti		1.6		1.6	0.32	48.32
[2] (linear)		64		64	31.50	1471.50
[2] (RO)	10^6	2176	16	2176	31.50	43711.50
Graphiti		16		16	3.15	483.15
[2] (linear)		640		640	315.00	14715.00
[2] (RO)	10^7	25600	160	25600	315.00	513915.00
Graphiti		160		160	31.50	4831.50

Table 10: Comparison of communication (MB) of simple contact tracing for threshold $\tau = 10$ for varying N= |V| + |E|.

Initialisation: Table 11 reports the cost of the initialisation phase for Graphiti. Recall that the cost of the initialisation phase consists of two invocations of $\mathcal{F}_{\mathsf{Shuffle}}$ and two invocations of $\mathcal{F}_{\mathsf{insecSort}}$ to generate the sharing of permutations π_A, π_B and public permutations π_S, π_D (see §3.4). Additionally, the initialisation involves one invocation of shuffle to generate the sharing of the permutation, $\pi_C = (\pi_A \circ \pi_B)^{-1}$. However, we note that in the consider setting of 2PC with a helper, the helper can generate the sharing of permutation π_C in the preprocessing phase. Thus, the online cost of initialisation for Graphiti remains the same as that of [2].

|V+E|	Runtime (s)		Communication (GB)	
	Online	Preprocessing	Online	Preprocessing
10^4	0.54	0.19	0.04	0.01
10^5	9.53	4.24	0.43	0.17
10^6	115.03	48.56	5.11	2.00
10^7	1727.07	624.09	61.20	23.91

Table 11: Cost of initialisation for GraphSC and Graphiti **varying N= |V| + |E|.**

Preprocessing cost: For completeness we report the preprocessing cost of Graphiti and GraphSC in the 2PC with helper setting. 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. Table 12 reports the preprocessing cost of Scatter and Gather. We note that unlike the online phase, where the local computations involved in Scatter and Gather are different, the rounds, communication and computation are exactly the same in the preprocessing phase. Table 13 and Table 14 reports the preprocessing cost of simple contact tracing for threshold $\tau = 10$ and varying N = |V| + |E|. Table 15 reports the preprocessing cost of probabilistic contact tracing for threshold $\tau = 10$ and varying N = |V| + |E|.

Ref	N	Runtime (s)	Comm. (MB)
[2] (linear)		0.02	0.32
[2] (RO)	10^4	0.25	6.40
Graphiti		0.01	0.16
[2] (linear)		0.13	3.20
[2] (RO)	10^5	3.42	89.60
Graphiti		0.07	1.60
[2] (linear)		1.25	32.00
[2] (RO)	10^6	42.33	1088.00
Graphiti		0.33	16.00
[2] (linear)		12.46	320.00
[2] (RO)	10^7	522.73	12800.00
Graphiti		3.23	160.00

Table 12: Comparison of the preprocessing phase for Scatter **and** Gather **for varying N= |V| + |E|.**

Ref	N	Scatter	Transition	Gather	ApplyV	Total
[2] (linear)		0.02		0.02	0.01	0.43
[2] (RO)	10^4	0.25	0.01	0.25		5.06
Graphiti		0.01		0.01	0.01	0.27
[2] (linear)		0.13		0.13	0.04	2.85
[2] (RO)	10^5	3.42	0.03	3.42		68.79
Graphiti		0.07		0.07	0.01	1.72
[2] (linear)		1.25		1.25	0.41	28.17
[2] (RO)	10^6	42.33	0.28	42.33		849.74
Graphiti		0.33		0.33	0.04	9.40
[2] (linear)		12.46		12.46	4.11	281.26
[2] (RO)	10^7	522.73	2.80	522.73		10486.68
Graphiti		3.23		3.23	0.41	93.01

Table 13: Comparison of the preprocessing run time(s) for the application of simple contact tracing for threshold $\tau = 10$ and varying N = |V| + |E|.

C.2 Shuffle

Table 16 reports the comparison of total run time and communication of shuffle while varying the number of shuffle sequential shuffle invocations. Table 17 reports the cost of our shuffle for varying vector size.

D Security Proofs

In this section, we discuss the security of our designed protocols. We rely on the standard real-world/ideal-world simulation paradigm to prove the security of our protocol. Let $\mathcal{A}$ denote the real-world adversary and $\mathcal{S}$ denote the ideal-world adversary. Without loss of generality, we provide the simulator proof for the case where $\mathcal{A}$ corrupts P_1. The simulation for the case of corrupt P_0 follows similarly. We prove the security of the protocol in the $\mathcal{F}_{\mathsf{setup}}$-hybrid model

Ref	N	Scatter	Transition	Gather	ApplyV	Total
[2] (linear)	10^4	0.32	0.16	0.32	0.16	8.16
[2] (RO)		6.40		6.40		129.76
Graphiti		0.16		0.16	0.02	4.81
[2] (linear)	10^5	3.20	1.6	3.20	1.58	81.58
[2] (RO)		89.60		89.60		1809.58
Graphiti		1.60		1.60	0.16	48.16
[2] (linear)	10^6	32.00	16	32.00	15.75	815.75
[2] (RO)		1088.00		1088.00		21935.75
Graphiti		16.00		16.00	1.58	481.58
[2] (linear)	10^7	320.00	160	320.00	157.50	8157.50
[2] (RO)		12800.00		12800.00		257757.50
Graphiti		160.00		160.00	15.75	4815.75

Table 14: Comparison of the preprocessing communication (MB) of simple contact tracing for threshold $\tau = 10$ for varying N= |V| + |E|.

Ref	Scatter		Total
	Propagate	ApplyE	
[2] (linear)	0.04		0.74
[2] (RO)	0.29		5.83
[2] (linear with decoupled Scatter)	0.02	0.02	0.41
[2] (RO with decoupled Scatter)	0.25		5.14
Graphiti	0.01	0.01	0.34

Table 15: Comparison of the preprocessing run time (s) for the application of contact tracing for probabilistic infection spread for $N = 10^4$.

Ref	#Shuffle	Time (ms)	Comm (MB)
[9]	1	44.00	3.20
Ours		43.68	4.00
[9]	10	263.41	32.00
Ours		219.51	32.80
[9]	50	1290.71	160.00
Ours		995.35	160.80
[9]	100	2306.88	320.00
Ours		1978.60	320.80

Table 16: Comparison of total run time and communication of shuffle while varying the number of sequential shuffle invocations where the same permutation is applied with $|T| = 10^5$.

$\|T\|$	Runtime (ms)		Communication (MB)	
	Online	Preprocessing	Online	Preprocessing
10^4	1.84	2.43	1.60	1.76
10^5	19.25	24.34	16.00	17.6
10^6	175.33	243.82	160.00	176.00
10^7	1710.27	2418.10	1600.00	1760.00

Table 17: Cost of our shuffle for varying vector size ($|T|$).

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.

In what follows, we first prove the security of our shuffle protocol followed by the security of Graphiti. Note that although we prove the security of Graphiti in the 2PC with a helper setting, our framework is generic and can instantiated with an appropriate MPC protocol to attain the desired level of security. This is because the security of Graphiti essentially boils down to the security of the underlying MPC, $\mathcal{F}_{\mathsf{Shuffle}}$, and $\mathcal{F}_{\mathsf{insecSort}}$.

D.1 Shuffle

Lemma D.1: The protocol, $\Pi_{(2+1)\text{-}\mathsf{Shuffle}}$ (Fig. 10) securely realizes the functionality $\mathcal{F}_{\mathsf{Shuffle}}$ (Fig. 9) against a semi-honest adversary that corrupts at most one party in $\mathcal{P}$.

Simulator $\mathcal{S}^{P_1}_{\Pi_{(2+1)\text{-}\mathsf{Shuffle}}}$

$\mathcal{S}^{P_1}_{\Pi_{(2+1)\text{-}\mathsf{Shuffle}}}$ proceeds as follows.

Preprocessing:

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

– Sample and send a random permutation $\pi'_1 \in S_N$ on behalf of the honest party.

– Sample and send a random vector $\mathsf{B}_0 \in \mathbb{Z}^{\mathsf{N}}_{2^\ell}$ on behalf of the honest party.

Online:

– Sample and send a random vector $\mathsf{A}_1 \in \mathbb{Z}^{\mathsf{N}}_{2^\ell}$ on behalf of the honest party.

Figure 12: Simulator $\mathcal{S}^{P_1}_{\Pi_{(2+1)\text{-}\mathsf{Shuffle}}}$ for corrupt P_1.

Proof. Let $\mathcal{A}$ denote the real-world adversary and $\mathcal{S}_{\Pi_{(2+1)\text{-}\mathsf{Shuffle}}}$ denote the corresponding ideal-world adversary. The simulator begins by first emulating $\mathcal{F}_{\mathsf{setup}}$ during which common keys are established with $\mathcal{A}$ that are used to sample the common randomness required throughout the protocol. Following this, it simulates the steps of the shuffle protocol. The simulation steps for a corrupt P_1 appear in Fig. 12, where the corresponding simulator is denoted as $\mathcal{S}^{P_1}_{\Pi_{(2+1)\text{-}\mathsf{Shuffle}}}$. Analogously the corruption of P_0 can also be simulated.

Observe that in the real world, during the preprocessing phase, $\mathcal{A}$ receives messages π'_1 and B_0 from P_2, which are randomly chosen from a uniform distribution. In the simulation, too, observe that $\mathcal{A}$ receives messages that are sampled randomly from the uniform distribution from the simulator. During the online phase in the real world, $\mathcal{A}$ receives A_1 from P_0, which is randomized using the random mask R_0. In the simulation too, $\mathcal{S}^{P_1}_{\Pi_{(2+1)\text{-}\mathsf{Shuffle}}}$ samples a random $\mathsf{A}_1 \in \mathbb{Z}^{\mathsf{N}}_{2^\ell}$ and sends it to $\mathcal{A}$ on behalf of the honest parties. Observe that here A_1 is random and hence is indistinguishable from the real world. In this way, real-world and ideal-world executions are indistinguishable. □

D.2 Graphiti

Initialisation: The ideal functionality for the initialisation phase of Graphiti appears in Fig. 13 and the secure protocol that realizes it, Π_{Init}, appears in Fig. 14.

Lemma D.2: The protocol, Π_{MPR} (Fig. 14) securely realizes the functionality $\mathcal{F}_{\mathsf{Init}}$ (Fig. 13) against a semi-honest adversary that corrupts at most one party in $\mathcal{P}$.

Proof. Observe that the protocol relies on invoking $\mathcal{F}_{\mathsf{Shuffle}}$ and $\mathcal{F}_{\mathsf{insecSort}}$. Hence, the security follows from the security of the underlying protocols for $\mathcal{F}_{\mathsf{Shuffle}}$ and $\mathcal{F}_{\mathsf{insecSort}}$. Apart from this, the only communication is π_{C_1}, which is sent by P_2 towards P_1. Observe that π_{C_1} is a random permutation, and hence the simulator for P_1 can simulate it by sampling a random permutation and sending it to P_1 on behalf of P_2. □

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

Without loss of generality, let $P_c \in \mathcal{P}$ denote the party corrupted by adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{Init}}$ interacts with parties in $\mathcal{P}$ and $\mathcal{S}$. It receives as input $\langle\cdot\rangle$-shares of the DAG-list G sorted in vertex order from P_0, P_1. $\mathcal{F}_{\mathsf{Init}}$ proceeds as follows.

- Reconstruct input G using $\langle\cdot\rangle$-shares of P_0, P_1.
- Sample a random permutation π_A and apply it on G to generate Shuffle-A.
- Sample a random permutation π_B and apply it on Shuffle-A to generate Shuffle-B.
- Set $\pi_C = \pi_A \circ \pi_B$
- Determine the permutation π_S that sorts Shuffle-A in the source order. Determine the permutation π_D that sorts Shuffle-B in the destination order.
- Sample a random permutation π_{A_0} and set $\pi_{A_1} = \Pi_{A_0}^{-1} \circ \pi_A$. Sample a random permutation π_{B_1} and set $\pi_{B_1} = \Pi_{B_0}^{-1} \circ \pi_B$. Sample a random permutation π_{C_1} and set $\pi_{C_1} = \Pi_{C_0}^{-1} \circ \pi_C$.
- Send $(\mathsf{Output}, \pi_{A_i}, \pi_{B_i}, \pi_{C_i}, \pi_S, \pi_D)$ to P_i for $i \in \{0, 1\}$.

Figure 13: Ideal functionality for initialisation phase of Graphiti**.**

Protocol Π_{Init}

Inputs: $\langle\cdot\rangle$-sharing of DAG-list G, sorted in vertex order, shared between parties P_0, P_1.
Outputs: Sharing of a random permutations $\pi_A = \pi_{A_1} \circ \pi_{A_0}, \pi_B = \pi_{B_1} \circ \pi_{B_0}, \pi_C = \pi_{C_1} \circ \pi_{C_0}$ such that $\pi_{A_i}, \pi_{B_i}, \pi_{C_i}$ lies with P_i for $i \in \{0, 1\}$ and public permutations π_S and π_D known to P_0 and P_1.

- Invoke $\mathcal{F}_{\mathsf{Shuffle}}$ to apply a random permutation π_A on the $\langle \mathsf{G} \rangle$ to generate a $\langle\cdot\rangle$-shares of a random ordering of the DAG-list, denoted as Shuffle-A.
- Invoke $\mathcal{F}_{\mathsf{Shuffle}}$ to apply a random permutation π_B on the $\langle\cdot\rangle$-shares of Shuffle-A to generate $\langle\cdot\rangle$-shares a random ordering of the DAG-list, denoted as Shuffle-B.
- Party P_2 computes $\pi_C = \pi_A \circ \pi_B$. Parties P_0, P_2 sample a random permutation π_{C_0} using the common random key.
- Party P_2 computes and sends $\pi_{C_1} = \pi_{C_0}^{-1} \circ \pi_C$ to P_1.
- Invoke $\mathcal{F}_{\mathsf{insecSort}}$ on $\langle\cdot\rangle$-shares of Shuffle-A to generate a public permutation π_S that sorts Shuffle-A in source order.
- Invoke $\mathcal{F}_{\mathsf{insecSort}}$ on $\langle\cdot\rangle$-shares of Shuffle-B to generate a public permutation π_D that sorts Shuffle-B in destination order.

Figure 14: Secure protocol for initialisation of Graphiti **in the two-party setting.**

Message Passing Round: The ideal functionality for the message-passing phase of Graphiti appears in Fig. 13.

Lemma D.3: The protocol, Π_{MPR} (Fig. 16) securely realizes the functionality $\mathcal{F}_{\mathsf{Shuffle}}$ (Fig. 15) against an semi-honest adversary that corrupts at most one party in $\mathcal{P}$.

Proof. Observe that the protocol relies on invoking $\mathcal{F}_{\mathsf{Shuffle}}$ and $\mathcal{F}_{\mathsf{MPC}}$. Hence, the security follows from security of the underlying protocol for $\mathcal{F}_{\mathsf{Shuffle}}$ and $\mathcal{F}_{\mathsf{MPC}}$. □

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

Without loss of generality, let $P_c \in \mathcal{P}$ denote the party corrupted by adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{MPR}}$ interacts with parties in $\mathcal{P}$ and $\mathcal{S}$. It receives as input $\langle\cdot\rangle$-shares of the the DAG-list G sorted in vertex order from P_0, P_1. $\mathcal{F}_{\mathsf{MPR}}$ proceeds as follows.

- Reconstruct input G using $\langle\cdot\rangle$-shares of P_1, P_2.
- Perform Propagate on G as defined in §3.1
- Perform ApplyE on G as defined in §3.1
- Perform Gather on G as defined in §3.1
- Perform ApplyV on G as defined in §3.1
- Generate random $\langle\cdot\rangle$-sharing of G.
- Send $(\mathsf{Output}, \langle \mathsf{G} \rangle_i)$ to P_i for $i \in \{0, 1\}$.

Figure 15: Ideal functionality for message passing round of Graphiti**.**

Protocol Π_{MPR}

Inputs: $\langle\cdot\rangle$-sharing of DAG-list G, sorted in vertex order, shared between parties P_0, P_1.
Outputs: $\langle\cdot\rangle$-sharing of DAG-list G, sorted in vertex order shared between parties P_0, P_1, where the data components are updated according to the underlying algorithm.

- **Propagate:** The parties invoke $\mathcal{F}_{\mathsf{MPC}}$ to evaluate steps 1-3 of Propagate described in algorithm 3. Following this, the parties invoke $\mathcal{F}_{\mathsf{Shuffle}}$ to apply π_A followed by applying the public permutation π_S on their local shares of DAG-list to generate the source order. Finally, the parties invoke $\mathcal{F}_{\mathsf{MPC}}$ to evaluate steps 5-7 of algorithm 3.
- **Apply-Edges:** The parties invoke $\mathcal{F}_{\mathsf{MPC}}$ on the circuit computing the function f_E to compute apply the function on the sharing of data component at each entry in the DAG-list.
- **source order to destination order:** The parties locally apply the public permutation π_A^{-1} to generate sharing of shuffle-B. The parties invoke one instance of $\mathcal{F}_{\mathsf{Shuffle}}$ which applies π_B on shares of shuffle-A and generate sharing of shuffle-B. The parties locally apply the public permutation π_D to get the destination order.

- **Gather:** The parties run the MPC protocol to evaluate Gather described in algorithm 4. In the secure evaluation of Algorithm 4, observe that steps 1-3 consist of only linear operations and hence can be evaluated non-interactively. Following this, the parties apply the public permutation π_D^{-1} on their local shares followed by one invocation of $\mathcal{F}_{\mathsf{Shuffle}}$ to apply π_C to generate the vertex order. Finally, steps 5-7, also consist of only linear operations and hence can be evaluated non-interactively.
- **Apply-Vertices:** The parties invoke $\mathcal{F}_{\mathsf{MPC}}$ on the circuit computing the function f_V to apply the function on the sharing of data component at each entry in the DAG-list.

Figure 16: Secure protocol for message passing round of Graphiti**.**