Cove: A System for Oblivious Community Search on Outsourced Streaming Graphs

Songlei Wang
Shenzhen University
songlei.wang@szu.edu.cn

Yifeng Zheng
The Hong Kong Polytechnic University
yifeng.zheng@polyu.edu.hk

Jie Chen
Shenzhen University
chenjie@szu.edu.cn

Linlin Shen
Shenzhen University
llshen@szu.edu.cn

Jin Liu
Qi An Xin Technology Group Inc.
liujin@qianxin.com

Haibo Hu
The Hong Kong Polytechnic University
haibo.hu@polyu.edu.hk

ABSTRACT

It is increasingly common to store and query graph databases at the cloud so as to harness well-understood benefits. This yet poses a threat to the privacy of information-rich graph data exposed to cloud providers, making it imperative to provide privacy assurance in search over outsourced graph databases. As a common and fundamental graph search task, community search plays a crucial role in applications such as identifying key opinion leaders in social networks, detecting fraudulent activities in financial systems, and tracing contact chains in healthcare settings. Specifically, given a graph and a query vertex, community search aims to find in the graph communities/subgraphs centered around the query vertex, inside which the vertices are densely connected. As graphs in many applications are generated in a streaming fashion (i.e., vertices and edges are continuously growing over time), there is a practical need for community search on streaming graphs to continuously identify communities for the query vertex. In this paper, we present Cove, the first system supporting oblivious community search over outsourced streaming graphs. Cove enables community search over an outsourced streaming graph, while protecting the streaming graph, the query, and the identified communities against cloud servers. At a high level, this is achieved through a tailored synergy of insights on lightweight cryptography, graph modeling, and data encoding. We implement Cove and conduct extensive experiments on real-world datasets for performance evaluation. The results demonstrate that, compared with a baseline directly built on generic secure multiparty computation, Cove achieves up to a 1963× speedup in search latency and reduces server-side communication cost by up to 415×.

1 INTRODUCTION

In real-world scenarios, various types of relationships among different entities can be naturally modeled as graphs, such as social networks, e-commerce networks, and financial transaction networks. With the proliferation of cloud computing which is widely known for benefits like scalability, flexibility, and cost-effectiveness [34], it has become increasingly common to store and query graph databases at the cloud [5, 20, 38]. Such trend of graph service outsourcing, however, raises critical privacy concerns as the outsourced graph databases could easily contain sensitive information in many application domains It is thus imperative to provide privacy assurance in search over outsourced graph databases.

A common graph search task is *community search*, which aims to find the *densely connected subgraph centered around a given query vertex* [30, 32, 49, 52]. Community search plays a vital role in applications such as identifying key opinion leaders in social networks, detecting fraudulent behaviors in financial systems, and tracing contacts in healthcare settings [17]. In practice, many of these applications involve *streaming graphs* [30], where directed edges are generated continuously, leading to graphs evolving over time. For example, in streaming financial networks, community search helps identify clusters of suspicious accounts for timely fraud detection; in streaming contact networks, it supports tracking high-risk individuals or emerging disease hotspots. With streaming graphs being outsourced to the cloud, how to perform community search with strong privacy assurance becomes a practical demand.

Related works. In the literature, extensive efforts have been devoted to privacy-preserving graph analytics, with most existing approaches focusing on static graphs, such as shortest-path queries [16, 19] and subgraph matching [23, 51]. More recently, a few works have examined secure search over streaming graphs, but they primarily address pattern detection [46] or attribute-based queries [50], rather than community search. A limited number of studies have considered privacy-preserving community discovery [10, 22, 43, 53]. However, these approaches either target different problem settings—such as graph publishing under differential privacy [10, 53]—or focus exclusively on community discovery over outsourced static graphs [22, 43]. None of them are capable of handling dynamic or streaming graph data. Specifically, they typically assume that each user holds a fixed and limited local view of the entire graph and must repeatedly interact with the servers using information derived from this static local view or by pre-building local search indexes. This assumption breaks down in streaming graphs: users cannot know the structure of the graph in advance, and the graph itself is continuously evolving as new edges arrive.

1.1 Overview of Our Techniques

In this paper, we present Cove, the *first* system for oblivious community search over streaming graphs. Cove enables the servers to obliviously maintain a continuously evolving streaming graph while supporting community search. It protects the confidentiality of the query while concealing both the structure of the streaming graph and the communities returned from the servers.

Cove builds on an emerging distributed trust model [2, 7, 15, 29, 36, 44], where three servers operated by independent cloud service providers jointly empower the secure service. With such a distributed trust model and compatible lightweight secret sharing techniques, Cove offers a novel and tailored secure realization of the state-of-the-art plaintext method [30] for community search on streaming graphs, which builds on the D-truss model tailored for directed graphs. At a high level, truss community search aims to identify, within each snapshot of a streaming graph as per the time-based sliding window model, the maximal subgraph (referred to as a *truss community*) centered around a specified query vertex that satisfies the query's connectivity requirements. In the following, we describe the challenges encountered in designing Cove and outline the key ideas behind our proposed solutions.

Unlike secure outsourced computation on *static* graphs—where the graph owner can locally conceal vertex connections because it holds the complete graph structure—the dynamic nature of streaming graphs introduces additional challenges. Specifically, if updates are not handled carefully, they can inadvertently leak information about vertex connections. For instance, consider a scenario where two vertices, v_1 and v_2, are connected to a growing number of neighboring vertices. If the cloud servers observe that the number of neighboring vertices for both v_1 and v_2 increases by one simultaneously, they may deduce that a new edge has been formed between them. To address this challenge, we depart from the commonly used posting list structure [13] or adjacency matrix, opting instead for the data-augmented graph (DAG)-list structure [29] to model the streaming graph. With this structure, each vertex and edge is represented as an independent tuple, enabling secure graph updates on the server side.

Challenge 1: Obliviously searching for a proximity subgraph around the query vertex. Intuitively, community search is user-centered, where the returned community should be centered around the query vertex, within the target radius specified by the query [32]. However, vertices that are distant from the query vertex may still be included in the resulting subgraph, leading to the so-called "free-rider effect" [49]. To address this, our idea is to first identify a subgraph around the query vertex within the radius, referred to as the *proximity subgraph*. The key challenge here is how to obliviously determine the vertices (named as *proximity vertices*) that are reachable from the query vertex within this radius, along with the edges (named as *adjacent edges*) connecting these vertices. Privacy-preserving reachability queries on *static* graphs is relatively straightforward and has been widely studied [26, 42, 48]. However, it becomes challenging on streaming graphs due to their *dynamic* and *unbounded* nature.

To address this challenge, our idea is to formulate the oblivious reachability query problem as an oblivious message passing problem. In particular, given the secret-shared input graph represented in the DAG-list structure, each tuple is attached with a flag. The flag of the tuple corresponding to the query vertex is initialized to 1, while the flags of all other tuples are set to 0. Subsequently, each vertex's flag is iteratively propagated to its adjacent edges and neighboring vertices. In each iteration, the flag of each tuple is obliviously updated by aggregating the flags it receives. The structure of the input graph and the flags passed through each tuple remain concealed during propagation. In the end, all tuples with nonzero flags collectively form the proximity subgraph.

To support the above oblivious message passing process, we observe that the state-of-the-art technique in [29] can serve as a good starting point, which also works in the secret sharing domain. However, directly applying this technique does not facilitate a working solution over *streaming* graphs. Firstly, it requires the input secret-shared graph (i.e., a DAG-list) to follow a strict vertex order, where all vertices appear sequentially before all edges. Although an initialization technique is introduced in [29] to yield vertex-ordered DAG-lists, it assumes that the total number of vertices is known in advance. This assumption is not feasible in streaming graphs, where the number of vertices is not fixed and may change over time. Moreover, if the initialization method were applied in such settings, the entire graph would need to be reshared with the servers whenever it is updated, which is impractical due to the frequent updates in streaming graphs. Secondly, the technique can only propagate the query vertex's flag to the proximity vertices. It does not support propagation to the adjacent edges.

In light of these challenges, we adapt the oblivious message passing technique [29] and propose an oblivious proximity subgraph search method. Firstly, through an in-depth analysis of the technique, we observe that the requirement for the input graph to follow a strict vertex order is not necessary. It suffices for all vertices to be ordered before the edges in the input DAG-list. This is because, in oblivious message passing, each vertex receives messages through edges from other vertices, rather than directly from the vertices themselves, making the vertex order irrelevant. Therefore, we relax the requirements for the input graph and adapt the oblivious message passing technique to handle streaming graphs. Secondly, we observe that each adjacent edge connects two proximity vertices, while all other edges connect at least one non-proximity vertex. Based on this observation, we design an oblivious dual-passing method, which allows the oblivious propagation of each vertex's flag to both its outgoing and incoming edges. Since the proximity vertices have a non-zero flag and all other vertices have a zero flag, the secret-shared multiplication of the flags each edge receives from its source and destination ensures that only the adjacent edges hold a non-zero flag, while all other edges have a zero flag. In this way, the adjacent edges can be obliviously determined.

Challenge 2: Obliviously filtering out weakly connected vertices and their adjacent edges from the proximity subgraph. Truss community search requires that the returned community be a densely connected subgraph. However, within the proximity subgraph, some vertices may exhibit weak connectivity and therefore fail to satisfy the query's requirements. The connectivity is quantified by the number of triangles, as this metric captures the mutual reinforcement among vertices [32]. Therefore, oblivious triangle counting techniques are required, which output the secret-shared triangle counts while ensuring that the structure of the graph remains hidden throughout the counting process. Privacy-preserving triangle counting has been widely studied in the literature. However, most existing methods are built on differential privacy [24, 25], which typically incurs accuracy loss. To overcome this limitation, we propose an efficient and accurate private triangle counting method based on the concept of oblivious message passing. We observe that the core of triangle counting is to determine the cardinality of the intersection between the neighboring vertex sets of each pair of vertices. To compute this efficiently, our idea is to first encode each vertex's identifier as a one-hot vector, and then obliviously pass each vertex's identifier to its neighbors, which are subsequently collected into a vector associated with the corresponding vertex. After the passing, the problem of triangle counting is simplified to the problem of calculating the secret-shared dot product of the vectors.

Contributions. We highlight our contributions below:

• We initiate the *first* study on secure community search over outsourced streaming graphs, building on a new primitive of oblivious message passing.

• We propose a secure proximity subgraph search method via tailoring oblivious message passing for streaming graphs, and introduce an oblivious dual-passing technique to address the challenge of oblivious adjacent edge identification.

• By simplifying the problem of oblivious triangle counting to the oblivious dot product of vectors using the concept of oblivious message passing, we propose an oblivious vertex connectivity computation method. This facilitates the filtering out weakly connected proximity vertices and their adjacent edges to produce the resulting community.

• We provide formal security analysis and conduct extensive experiments on three real-world graph datasets. The results show that Cove is able to process each snapshot graph within a few minutes. Compared to a baseline built on the widely adopted generic secure multiparty computation (MPC) framework MP-SPDZ [28], Cove achieves up to a 1963× speedup in search latency and up to a 415× reduction in server-side communication cost.

2 PRELIMINARIES

2.1 D-Truss Community Search

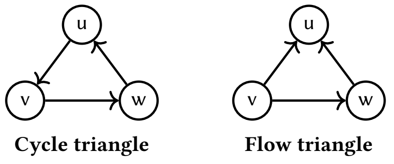
Following the latest advance in the plaintext domain [30], we consider a directed and unweighted streaming graph.

Definition 1. ***Streaming Graph.*** *A streaming directed graph is a continually growing sequence of edges, denoted as* $G = \{e_{u,v}, \dots\}$, *where* $e_{u,v} = (u, v, t)$ *represents a directed edge from vertex* u *to vertex* v *that arrives at time point* t.

Given an edge $e_{u,v}$, vertex u is referred to as v's in-neighbor, and vertex v is referred to as u's out-neighbor. We use e to denote an edge when the context is clear. N_u^- denotes the set of in-neighbors of vertex u, and N_u^+ denotes the set of out-neighbors of vertex u. Given the continually growing size of the streaming graph, and building on prior work in the plaintext domain [30], we focus on the most recent edges using a time-based sliding window model.

Definition 2. ***Time-based Sliding Window Model.*** *A time-based window with length W includes edges whose timestamps fall within the interval $(t - W, t]$, where t denotes the current time. A time-based sliding window with a slide interval of β is one that moves forward by β time units at each step. The subgraph induced by all the edges within the time-based window at time point t is referred to as the snapshot graph.*

This paper focuses on the recently developed and widely adopted community search model for directed graphs: the D-truss community search [32]. It identifies communities based on the count of two types of non-isomorphic triangles: *cycle triangles* and *flow triangles*, which are structured as follows.



The count of cycle triangles and flow triangles formed by an edge is referred to as its *cycle support* and *flow support*, respectively. Their definitions are as follows.

Definition 3. ***Cycle Support.*** *Edge* e*'s cycle support is the number of vertices that form cycle triangles with it, denoted as* $\text{sup}_c(e)$.

Definition 4. ***Flow Support.*** *Edge* e*'s flow support is the number of vertices that form flow triangles with it, denoted as* $\text{sup}_f(e)$.

Since community search is user-centered and personalized, the returned community should be relevant to the given query vertex [32]. Therefore, we consider a radius-constrained D-truss community search on streaming graphs as follows.

Definition 5. ***D-Truss Community Search over Streaming Graphs.*** *Given a streaming graph G and a query $Q = \{q, r, (k_c, k_f), W, \beta\}$, where,* q *is the query vertex, r is the target radius, and k_c and k_f are two non-negative integers, the radius-constraint D-truss community search aims to find a subgraph g of each snapshot of G such that:*

• *The shortest path length from* q *to* $\forall v \in g$ *is at most r.*

• $\forall e \in g, \text{sup}_c(e) \geq k_c$ *and* $\text{sup}_f(e) \geq k_f$.

• *g is maximal, i.e., g should contain the most vertices among all subgraphs of the snapshot that satisfy the above two conditions.*

2.2 Cryptographic Techniques

Cove makes use of replicated secret sharing (RSS) [3] to support lightweight secure computation. Given a private value x in the ring $\mathbb{Z}_{2^l}$, RSS splits it into three additive shares $\langle x\rangle_1, \langle x\rangle_2, \langle x\rangle_3 \in \mathbb{Z}_{2^l}$, such that $x = \langle x\rangle_1 + \langle x\rangle_2 + \langle x\rangle_3$. These shares are distributed to three parties in pairs: $(\langle x\rangle_1, \langle x\rangle_2)$, $(\langle x\rangle_2, \langle x\rangle_3)$, and $(\langle x\rangle_3, \langle x\rangle_1)$, ensuring that no single party can reconstruct the secret alone. Throughout this paper, we use $[\![\cdot]\!]$ to denote secret-shared values or sets.

In addition, we leverage a suite of RSS-based secure building blocks in the design of Cove.

• Secure comparison (secCmp) [8]: Given two secret-shared values $[\![a]\!]$ and $[\![b]\!]$, it outputs $[\![1]\!]$ if $a \geq b$, and $[\![0]\!]$ otherwise.

• Secure equality test (secTest) [8]: Given $[\![a]\!]$ and $[\![b]\!]$, this outputs $[\![1]\!]$ if $a = b$, and $[\![0]\!]$ otherwise.

• Secure shuffle (secShuffle) [4]: Given a secret-shared vector $[\![T]\!]$, it outputs $[\![\pi(T)]\!]$, where π is a random secret permutation unknown to the parties. We use π^{-1} to denote the inverse of π, and $\pi_1 \circ \pi_0$ to represent permutation composition, i.e., $\pi_1 \circ \pi_0(T) = \pi_1(\pi_0(T))$.

• Secure sort (secSort) [6]: Given a list of secret-shared key-value pairs, this primitive outputs the secret-shared values sorted in descending order based on their keys. The sort is stable, preserving the order of elements with equal keys.

• Insecure sort (insecSort) [29]: Similar to secSort, but during execution, the permutation mapping the input value list to the output

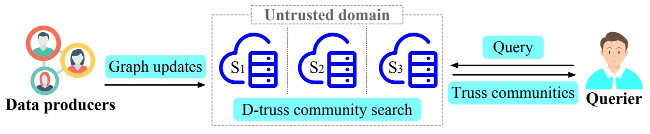


Figure 1: The system architecture of Cove.

is revealed in the clear. Note that both the values and the keys remain secret-shared throughout the protocol.

3 SYSTEM OVERVIEW

3.1 Architecture

As shown in Fig. 1, there are three types of entities in Cove: data producers, the querier, and the servers. The querier aims to continuously manage the constantly growing streaming graph contributed by data producers and to detect specific truss communities over the streaming graph. Each producer can be viewed as a single vertex that continuously generates incident edges, which collectively form the streaming graph. The querier is motivated by the advantages of outsourcing computation, which enable it to offload continuous streaming graph analytics to servers and thereby reduce local infrastructure costs [27, 33]. Hence, it leverages the outsourcing paradigm to manage the evolving streaming graph and continuously perform community search, alerting the querier when the specified truss communities emerge.

A running example of Cove is in public health monitoring, where public health authorities rely on the cloud to continuously manage the growing contact tracing data contributed by users and to detect specific disease hotspots over the streaming graph. In practice, such an outsourcing paradigm has been widely adopted in dynamic graph search systems (e.g., [1, 5, 39]). However, due to concerns about the private information-rich streaming graph, it is essential to embed privacy assurance in the outsourced service.

Cove leverages a distributed trust model, where three servers (denoted as S_1, S_2, S_3) from separate trust domains jointly empower the oblivious community search service. This means that the servers could be hosted by different cloud services providers. Such a distributed trust model has seen increasing adoption in recent security designs (e.g. [7, 15, 29, 44]) as well as in industry [18, 35, 37].

3.2 Threat Model and Security Guarantees

Threat model. Similar to prior works that adopt the multi-server setting for security designs [7, 29, 44], we consider a semi-honest and non-colluding adversary model, where each server honestly follows our protocol but may *individually* attempt to deduce private information while providing community search services. In practice, such non-colluding servers could be cloud servers hosted by different competitive commercial cloud providers (e.g., Amazon, Microsoft, and Google). The competitive nature of these providers creates strong incentives for them to maintain independent and non-collusive operations. Such non-colluding multi-server model has also been widely adopted for building secure database applications (e.g., [40, 45, 47, 54]) and is also utilized in industry (e.g., Safeheron Wallet [41]). Additionally, with a focus on protecting the privacy of the streaming graph and the resulting truss communities against the servers, we consider the data producers and the querier to be trustworthy parties, as the data producers only contribute their local updates to the servers, and the querier only receives the resulting truss communities.

Security guarantees. Under the aforementioned threat model, Cove guarantees that each server only learns the following information and nothing more. (1) Truss community occurrence in the streaming graph. This is essential for the servers to deliver the secure service (i.e., securely detecting truss communities and alerting the querier). Without this, the service would be ineffective. (2) Timestamp of each edge. They are also considered in most security designs for time-series data (e.g., [15]). (3) Query radius. We consider the query radius to be public, as it is independent of the private streaming graph. (4) Count information. This includes the count of edges in the current streaming graph and the number of edges and vertices filtered out during the community search process. We discuss how to mitigate the leakage in Appendix A.

4 FRAMEWORK OF COVE

System inputs. We adopt the widely used edge-list representation [21, 30] to model the streaming graph, as formalized in Definition 1, where only edges are explicitly represented and vertices are implicitly encoded within these edges. To enable efficient and secure community search over streaming graphs, our system employs the lightweight cryptographic primitive of RSS. Concretely, for each locally generated edge $\mathrm{e}_{\mathrm{u,v}} = (\mathrm{u}, \mathrm{v}, t)$, the data producer splits the private values u and v into replicated secret shares and distributes these shares, together with the public timestamp t, to the servers. In addition, the querier splits the query vertex q and the required support thresholds (k_c, k_f) into RSS form and distributes the corresponding shares—along with the public radius, time-window length, and slide interval—to the servers.

Secure proximity subgraph search. Once the streaming graph is secret-shared across the servers, they privately perform truss community search over each secret-shared snapshot graph according to the querier's requirements. As recalled in Definition 5, community search is user-centered and personalized: the returned community must be relevant to a given query vertex [32]. Therefore, we first propose a secure proximity subgraph search method (Section 5), which enables the servers to obliviously identify a subgraph centered at the query vertex within the radius specified by the querier.

The key challenge is how to obliviously determine the proximity vertices that are reachable from the query vertex within the given radius, together with the adjacent edges connecting these vertices. To address this challenge, we formulate the oblivious reachability problem as an *oblivious message passing* process following a "Scatter-Gather" framework. Specifically, given the secret-shared input graph, each tuple (vertex or edge) is associated with a secret-shared flag. The flag corresponding to the query vertex is obliviously initialized to 1, while the flags of all other tuples are initialized to 0. Subsequently, the servers iteratively propagate each vertex's flag along its incident edges to neighboring vertices (i.e., Scatter), and then obliviously aggregate the secret-shared flags received by each vertex in each iteration to update its flag (i.e., Gather). Throughout this process, both the structure of the input graph and the propagated flags remain concealed. At the end of propagation, all vertices with nonzero flags are identified as proximity vertices.

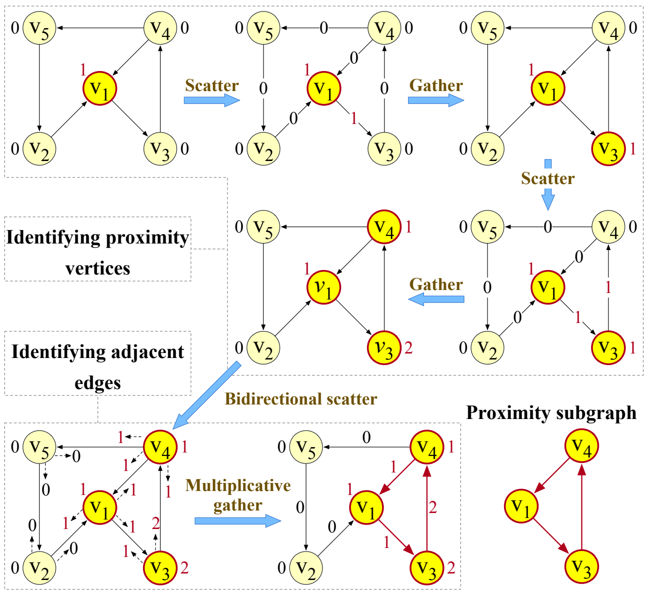


Figure 2: Example of the underlying framework for our proximity subgraph search, where vertex v_1 is the query vertex and the query radius $r = 2$.

We then consider how to enable the servers to obliviously identify the adjacent edges. We observe that an adjacent edge connects two proximity vertices, whereas any other edge is incident to at least one non-proximity vertex. Based on this observation, we design an *oblivious dual message passing* method that allows each vertex's flag to be obliviously propagated to both its outgoing and incoming edges. Since proximity vertices carry nonzero flags while all other vertices have zero flags, each edge receives flags from both its source and destination vertices; by computing the secret-shared product of these two flags, only edges connecting two proximity vertices obtain a nonzero value, while all other edges remain zero. In this way, the servers can identify the adjacent edges, which, together with the proximity vertices, form the proximity subgraph.

Example 1. *Fig. 2 illustrates the underlying framework of our proximity subgraph search method. In the first round of flag propagation (i.e., one iteration of* Scatter-Gather*), vertex v_3 receives a flag of 1; in the second round, vertex v_4 receives a flag of 1. Accordingly, the proximity vertices are v_3 and v_4. The vertex flags are then propagated to their adjacent edges, and only edges $e_{1,3}$, $e_{3,4}$, and $e_{4,1}$ receive two nonzero flags, identifying them as adjacent edges. As a result, the output proximity subgraph consists of vertices v_1, v_3, and v_4, together with edges $e_{1,3}$, $e_{3,4}$, and $e_{4,1}$.*

Secure low-support edge filtering. Given the proximity subgraph, the next step is to compute the cycle support and flow support of each edge and to filter out edges whose supports fall below the thresholds specified by the querier. Since an edge's cycle and flow supports equal the numbers of cycle and flow triangles it induces, respectively, the core challenge is to securely count triangles over the secret-shared proximity subgraph. However, directly counting triangles in this setting is nontrivial, as the computation requires access to each vertex's neighbors, while the proximity subgraph is represented solely as independent edges, with vertex information implicitly encoded in these edges. To address this challenge, we decompose secure triangle counting into two steps: (i) securely aggregating each vertex's neighboring vertices to produce secret-shared neighbor vectors, and (ii) securely computing the secret-shared number of triangles induced by each edge based on these neighbor vectors.

For the first step, we propose a secure neighbor aggregation method following the Scatter-Gather framework. Specifically, the servers obliviously propagate each vertex's ID—encoded in *one-hot vector form*—along its incident edges to neighboring vertices (i.e., Scatter). For in-neighbor aggregation, vertex IDs are propagated along the forward direction of edges, whereas for out-neighbor aggregation, vertex IDs are propagated along the backward direction of edges. The servers then obliviously aggregate all secret-shared IDs received by each vertex to construct its neighbor vector (i.e., Gather). In these secret-shared neighbor vectors, the position of each entry equal to 1 corresponds to the ID of a neighboring vertex.

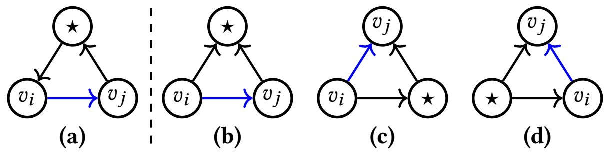


Figure 3: Triangles induced by edge $e_{i,j}$ under different roles: (a) cycle triangle; (b)—(d) flow triangles.

For the second step, we first distinguish the triangles induced by an edge $e_{i,j}$ under different roles, as illustrated in Fig. 3. As shown in the figure, the number of cycle triangles induced by edge $e_{i,j}$ equals the cardinality of the intersection between the in-neighbors of vertex v_i and the out-neighbors of vertex v_j. This quantity can be computed as the dot product (denoted by "·") between the corresponding neighbor vectors $N^-_{v_i}$ and $N^+_{v_i}$:

$$N^-_{v_i} \cdot N^+_{v_j}. \tag{1}$$

In contrast, the *flow triangles* induced by edge $e_{i,j}$ are more involved and can be categorized into three types. Accordingly, the number of flow triangles induced by $e_{i,j}$ equals the cardinality of the union of the following three intersections: (i) the intersection between the in-neighbors of v_i and the in-neighbors of v_j; (ii) the intersection between the out-neighbors of v_i and the in-neighbors of v_j; and (iii) the intersection between the out-neighbors of v_i and the out-neighbors of v_j. This can be formulated as

$$\text{sum}\left[(N^-_{v_i} \odot N^-_{v_j}) \vee (N^+_{v_i} \odot N^-_{v_j}) \vee (N^+_{v_i} \odot N^+_{v_j})\right]. \tag{2}$$

Here, "$\odot$" denotes element-wise multiplication, "$\vee$" denotes the element-wise OR operation, and sum$(\cdot)$ denotes the summation of all vector entries. Based on Eq. 1 and Eq. 2, triangle counting reduces to element-wise vector operations. While addition and multiplication are naturally supported in the secret sharing domain, the element-wise OR operation ($\vee$) is not natively supported. To address this limitation, we reformulate the element-wise OR between two vectors $\mathbf{a}$ and $\mathbf{b}$ as $\mathbf{a} \vee \mathbf{b} = \mathbf{a} + \mathbf{b} - \mathbf{a} \odot \mathbf{b}$.

Next, the servers filter out edges whose cycle and flow supports are below the thresholds specified by the querier. They then filter out vertices that become unreachable from the query vertex after edge filtering, using oblivious message passing. If any vertices are removed, some edges' cycle or flow supports may further decrease; thus, the servers iteratively recompute these supports and repeat the filtering. Moreover, once a vertex is filtered, all edges incident to it become irrelevant and must also be removed before the next recomputation. Conversely, if no vertices are filtered, all remaining vertices are reachable from the query vertex and all remaining edges satisfy the cycle and flow support requirements. The servers then output the remaining subgraph as the truss community.

5 SECURE PROXIMITY SUBGRAPH SEARCH

Preprocessing the secret-shared streaming graph. Recall Definition 2, where community search on a streaming graph is performed over the current snapshot graph. Given the secret-shared streaming graph, the current time point t, and the time window size W, the servers first derive the current secret-shared snapshot graph $[\![G_t]\!] = \{[\![\mathsf{e}_{i,j}]\!], \dots\}$ by checking whether each edge's timestamp falls within the interval $(t - W, t]$.

Our secure proximity subgraph search is built upon oblivious message passing, which requires representing the input graph as a data-augmented graph (DAG) list. In this representation, each vertex or edge is encoded as a tuple with three components: "src" (source vertex ID), "dst" (destination vertex ID), and "isV" (a flag indicating whether the tuple corresponds to a vertex or an edge). For example, a vertex v_i is encoded as $(\mathsf{v}_i, \mathsf{v}_i, 1)$, while a directed edge $\mathsf{e}_{i,j}$ is encoded as $(\mathsf{v}_i, \mathsf{v}_j, 0)$. Accordingly, for each $[\![\mathsf{e}_{i,j}]\!] \in [\![G_t]\!]$, the servers extract $[\![\mathsf{v}_i]\!]$ and $[\![\mathsf{v}_j]\!]$ and generate the following DAG-list tuples: $([\![\mathsf{v}_i]\!], [\![\mathsf{v}_i]\!], [\![1]\!])$, $([\![\mathsf{v}_j]\!], [\![\mathsf{v}_j]\!], [\![1]\!])$, $([\![\mathsf{v}_i]\!], [\![\mathsf{v}_j]\!], [\![0]\!])$. We use N to denote the total number of tuples in the list.

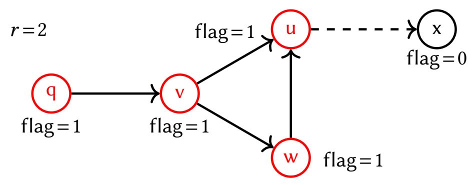
Starting point. To achieve secure proximity subgraph search, our starting point is the oblivious message passing technique [29]. It enables the parties to obliviously pass messages along the edges between vertices in a secret-shared graph within the DAG-list structure. To obliviously identify the proximity vertices and adjacent edges, Cove first attaches a flag of $[\![0]\!]$ to each tuple in the snapshot graph, and then obliviously sets the flag of the tuple corresponding to the query vertex to $[\![1]\!]$. The servers then obliviously propagate the query vertex's flag (i.e., message) to the proximity vertices and adjacent edges. At the end, the tuples with non-zero flags constitute the proximity subgraph. However, we make two key observations that make Graphiti [29] not directly applicable to our problem and drive us to develop a new oblivious message passing-based secure proximity subgraph search method tailored for streaming graphs.

• Observation-1: The input of Graphiti [29] is a secret-shared graph within the DAG-list structure. However, it requires the input DAG-list to follow a strict vertex order by the ID, where all vertices appear sequentially before all edges (i.e., $\mathsf{v}_1, \mathsf{v}_2, \dots, \mathsf{e}_{1,2}, \mathsf{e}_{2,3}, \dots$). Although [29] proposes an initialization technique to achieve vertex-ordered DAG-lists, it assumes that the total number of vertices is known in advance. This assumption is impractical, especially for streaming graphs, where the number of vertices changes over time. Moreover, if the initialization method were applied in such settings, the entire graph would need to be reshared with the servers whenever it is updated, which is impractical due to the frequent updates in streaming graphs.

• Observation-2: The original oblivious message passing method can only identify the proximity vertices, but it cannot identify the adjacent edges.

For the first issue, we observe that a strict vertex order is unnecessary; it suffices for all vertices to be placed before the edges. The order of the vertices is irrelevant, as long as the original vertex order is maintained during the oblivious message passing. This is because, in oblivious message passing, each vertex receives messages through edges from other vertices, rather than directly from the vertices themselves, making the vertex order irrelevant. Therefore, we adapt Graphiti [29] to relax the requirements for the input secret-shared DAG-list, making the oblivious message-passing method applicable to streaming graphs. For the second issue, we design an oblivious dual-passing method that allows the servers to obliviously propagate the non-zero flags of the proximity vertices to their adjacent edges.

Overview of our method. Initially, the servers attach a secret-shared flag $[\![0]\!]$ to each tuple, and obliviously set the flag of the vertex corresponding to the query vertex to $[\![1]\!]$. Subsequently, the servers obliviously propagate the query vertex flag of $[\![1]\!]$ to its proximity vertices. Next, the servers should obliviously propagate the flags (with non-zero values) of the proximity vertices to the adjacent edges. Note that during the propagation of the query vertex's flag to its proximity vertices, the edges can store the flag as it passes through them. However, this does not ensure that all adjacent edges will have a non-zero flag. The following example shows this case:



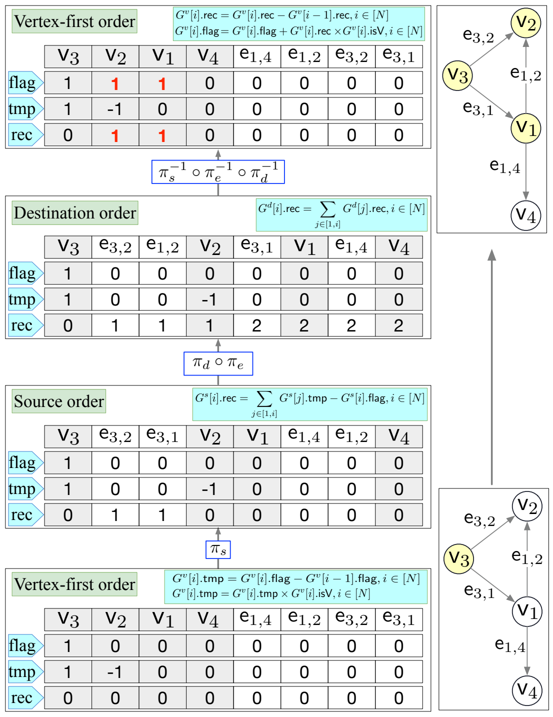
In the example with $r = 2$, the query vertex q's flag of 1 will only propagate for two hops, and the vertices v, w, u are the proximity vertices. We observe that during the flag propagation process, no flag $= 1$ is passed through the edge $\mathsf{e}_{\mathsf{w},\mathsf{u}}$. However, it connects two proximity vertices w and u and is an adjacent edge. Thus, merely storing the flag as it passes through edges is not sufficient.

To address this issue, we design an oblivious dual-passing method. Specifically, the servers first obliviously propagate the flags (with nonzero values) of all proximity vertices to their outgoing and incoming edges. Then, the servers multiply both secret-shared flags of each edge, obtained from its source and destination vertices, ensuring that only the adjacent edges retain a nonzero flag while other edges have a flag value of 0.

Next, we detail our secure proximity subgraph search method. It consists of two phases. In the first phase (Section 5.1), the servers obliviously propagate the query vertex's flag to the proximity vertices. In the second phase (Section 5.2), the servers obliviously propagate the proximity vertices' flags to the adjacent edges. Finally, a naive method is to directly recover the flags of all tuples to identify the proximity vertices and adjacent edges. However, this could leak the access pattern [14] and potentially expose the structure of the graph. To prevent this, Cove lets the servers obliviously shuffle all tuples before recovering their flags, thereby identifying

Algorithm 1 Secure Propagation of Flags to Proximity Vertices

```
Input: Snapshot graph $[\![G]\!]$ and the target radius $r$.
Output: Snapshot graph $[\![G^v]\!]$ after propagation of flags.
1:  $[\![G]\!] = \text{secShuffle}([\![G]\!])$.
2:  $[\![G^v]\!] = \text{secSort}([\![G]\!], [\![\text{isV}]\!])$. % Vertex-first order.
3:  $\pi_s = \text{insecSort}([\![G^v]\!])$. % Permutation for source order.
4:  $[\![G^s]\!] = \pi_s([\![G^v]\!])$. % Source order.
5:  $\pi_e = \text{insecSort}([\![G^s]\!])$.%Perm. for edge-first order.
6:  $[\![G^e]\!] = \pi_e([\![G^s]\!])$. % Edge-first order.
7:  $\pi_d = \text{insecSort}([\![G^e]\!])$. % Permutati. for destination order.
8:  for all $\tau \in [1, r]$ do
9:      $[\![G^v[i].\text{tmp}]\!] = [\![G^v[i].\text{flag}]\!] - [\![G^v[i-1].\text{flag}]\!], i \in [N]$.
10:     $[\![G^v[i].\text{tmp}]\!] = [\![G^v[i].\text{tmp}]\!] \times [\![G^v[i].\text{isV}]\!], i \in [N]$.
11:     $[\![G^s]\!] = \pi_s([\![G^v]\!])$. % Source order.
12:     $[\![G^s[i].\text{rec}]\!] = \sum_{j\in[1,i]} [\![G^s[j].\text{tmp}]\!] - [\![G^s[i].\text{flag}]\!], i \in [N]$.
13:     $[\![G^d]\!] = \pi_d(\pi_e([\![G^s]\!]))$. % Destination order.
14:     $[\![G^d[i].\text{rec}]\!] = \sum_{j\in[1,i]} [\![G^d[j].\text{rec}]\!], i \in [N]$.
15:     $[\![G^v]\!] = \pi_s^{-1}(\pi_e^{-1}(\pi_d^{-1}([\![G^d]\!])))$. % Vertex-first order.
16:     $[\![G^v[i].\text{rec}]\!] = [\![G^v[i].\text{rec}]\!] - [\![G^v[i-1].\text{rec}]\!], i \in [N]$.
17:     $[\![G^v[i].\text{flag}]\!] += [\![G^v[i].\text{rec}]\!] \times [\![G^v[i].\text{isV}]\!], i \in [N]$.
18:     $[\![G^v[i].\text{tmp}]\!] = [\![G^v[i].\text{rec}]\!] = [\![0]\!], i \in [N]$. % Reset.
19: end for
```


Vertex-first order
$G^v[i].\text{rec} = G^v[i].\text{rec} - G^v[i-1].\text{rec}, i \in [N]$
$G^v[i].\text{flag} = G^v[i].\text{flag} + G^v[i].\text{rec} \times G^v[i].\text{isV}, i \in [N]$
$\pi_s^{-1} \circ \pi_e^{-1} \circ \pi_d^{-1}$
Destination order
$G^d[i].\text{rec} = \sum_{j\in[1,i]} G^d[j].\text{rec}, i \in [N]$
$\pi_d \circ \pi_e$
Source order
$G^s[i].\text{rec} = \sum_{j\in[1,i]} G^s[j].\text{tmp} - G^s[i].\text{flag}, i \in [N]$
π_s
Vertex-first order
$G^v[i].\text{tmp} = G^v[i].\text{flag} - G^v[i-1].\text{flag}, i \in [N]$
$G^v[i].\text{tmp} = G^v[i].\text{tmp} \times G^v[i].\text{isV}, i \in [N]$
v_2
v_3
v_1
v_4
$e_{3,2}$
$e_{1,2}$
$e_{3,1}$
$e_{1,4}$

Vertex-first order

	v_3	v_2	v_1	v_4	$e_{1,4}$	$e_{1,2}$	$e_{3,2}$	$e_{3,1}$
flag	1	1	1	0	0	0	0	0
tmp	1	-1	0	0	0	0	0	0
rec	0	1	1	0	0	0	0	0

Destination order

	v_3	$e_{3,2}$	$e_{1,2}$	v_2	$e_{3,1}$	v_1	$e_{1,4}$	v_4
flag	1	0	0	0	0	0	0	0
tmp	1	0	0	-1	0	0	0	0
rec	0	1	1	1	2	2	2	2

Source order

	v_3	$e_{3,2}$	$e_{3,1}$	v_2	v_1	$e_{1,4}$	$e_{1,2}$	v_4
flag	1	0	0	0	0	0	0	0
tmp	1	0	0	-1	0	0	0	0
rec	0	1	1	0	0	0	0	0

Vertex-first order

	v_3	v_2	v_1	v_4	$e_{1,4}$	$e_{1,2}$	$e_{3,2}$	$e_{3,1}$
flag	1	0	0	0	0	0	0	0
tmp	1	-1	0	0	0	0	0	0
rec	0	0	0	0	0	0	0	0

Figure 4: Example for our oblivious message passing round.

the proximity vertices and adjacent edges while preserving the confidentiality of the access pattern.

5.1 Secure Vertex-to-Vertex Flag Propagation

The method follows the framework of [29], consisting of two phases: (1) Scatter: propagating information from a source vertex to its outgoing edges; and (2) Gather: aggregating information from incoming edges at the destination vertex. By invoking Scatter − Gather once, the query vertex's flag is propagated one hop forward. Therefore, by invoking them r times, the query vertex's flag will propagate to all proximity vertices within the given radius r. Algorithm 1 (named as secVertex) shows the process, with the design rationale presented as follows.

Preparation (lines 1-7). Each tuple in the snapshot graph is represented as ($[\![\text{src}]\!]$, $[\![\text{dst}]\!]$, $[\![\text{isV}]\!]$, $[\![\text{flag}]\!]$). First, the servers obliviously set the query vertex's flag to $[\![1]\!]$ by evaluating:

$$[\![G[i].\text{flag}]\!] = [\![G[i].\text{isV}]\!] \times \text{secTest}([\![q]\!], [\![G[i].\text{src}]\!]), \forall i \in [N]. \quad (3)$$

To store the propagated information, we associate each tuple with two components, $[\![\text{tmp}]\!]$ and $[\![\text{rec}]\!]$, both initially set to $[\![0]\!]$. After initialization, to protect the permutations in the subsequent insecure sorting process, the servers first invoke the secure shuffle (secShuffle) operation [4] to obliviously permute the DAG-list (line 1). Then, the servers obliviously sort $[\![G]\!]$ into *vertex-first order* (denoted as $[\![G^v]\!]$), where all vertices appear before edges (e.g., $v_3, v_1, v_2, e_{1,2}, \ldots$). This is achieved using secSort(·) with $[\![\text{isV}]\!]$ as the sort key (line 2). Since vertices have isV = 1 and edges have isV = 0, this operation ensures that all vertices are obliviously positioned before edges.

Next, the servers invoke insecSort(·) to compute a permutation π_s that transforms the snapshot graph from the vertex-first order into the *source order* (denoted as $[\![G^s]\!]$), where each vertex appears immediately before all its outgoing edges, e.g., $v_2, e_{2,1}, v_1, e_{1,2}, e_{1,3}, \cdots$. (line 3). Since insecSort(·) is stable, every vertex in $[\![G^s]\!]$ remains immediately before its outgoing edges. Notably, since $[\![G^v]\!]$ is obtained through a secure sorting protocol applied to the original snapshot subgraph, the mapping between $[\![G]\!]$, $[\![G^v]\!]$ appears random from the servers' view. Moreover, as the servers are unaware of the exact vertex count, π_s appears to them as merely mapping an arbitrary DAG-list to a sorted list, revealing no information about the relationship between $[\![G]\!]$, $[\![G^s]\!]$. Thus, π_s can be made public.

The servers then compute the permutation π_d to transform the snapshot graph from the source order into the *destination order* (denoted as $[\![G^d]\!]$), where all incoming edges of a vertex are placed immediately before that vertex (e.g., $e_{2,1}, v_1, e_{1,2}, e_{3,2}, v_2, \ldots$). To achieve this, the servers first convert $[\![G^s]\!]$ into *edge-first order* (denoted as $[\![G^e]\!]$, line 6), where all edges appear before vertices (e.g., $e_{2,1}, e_{1,2}, e_{3,2}, v_1, v_2, \ldots$). This is achieved by invoking insecSort(·) on $[\![G^s]\!]$ with $[\![\neg\text{isV}]\!]$ as the sort key to obtain the public permutation π_e (line 5). Here "¬" represents the NOT operator, which can be easily computed locally (one pair of shares is set to $\langle\text{isV}\rangle_i = 1 - \langle\text{isV}\rangle_i$ and the others are set to $\langle\text{isV}\rangle_i = -\langle\text{isV}\rangle_i$). Since all edges have $\neg\text{isV} = 1$ and all vertices have $\neg\text{isV} = 0$, this operation ensures that all edges are positioned before edges, i.e., edge-first order. In addition, since the insecure sorting protocol is stable, the vertex order is preserved. Then, the servers invoke insecSort(·) on $[\![G^e]\!]$ obtain the public permutation π_d (line 7). Since $[\![G^e]\!]$ is already in edge-first order and insecSort(·) is stable, all incoming edges of a vertex are placed before that vertex, i.e., destination order.

Scatter (lines 9-12). We observe that it is unnecessary for the input of Scatter to be strictly in vertex order; instead, it suffices for

Algorithm 2 Secure Propagation of Vertex Flags to Edges

```
Input: $[\![G^v]\!]$ in vertex-first order; perm. $\pi_s$ for source order.
Output: Snapshot graph $[\![G^{s'}]\!]$ after the propagation of flags.
1: $[\![G^v[i].\text{tmp}]\!] = [\![G^v[i].\text{flag}]\!] - [\![G^v[i-1].\text{flag}]\!], i \in [N]$.
2: $[\![G^v[i].\text{tmp}]\!] = [\![G^v[i].\text{tmp}]\!] \times [\![G^v[i].\text{isV}]\!], i \in [N]$.
3: $[\![G^s]\!] = \pi_s([\![G^v]\!])$. % Source order.
4: $[\![G^s[i].\text{rec}]\!] = \sum_{j\in[1,i]} [\![G^s[j].\text{tmp}]\!] - [\![G^s[i].\text{flag}]\!], i \in [N]$.
5: $[\![G^{s'}]\!] = \text{secSort}([\![G^s]\!])$. % Destination order (vertex-first).
6: $[\![G^{s'}[i].\text{rec}]\!] = [\![G^{s'}[i].\text{rec}]\!] \times (\sum_{j\in[1,i]} [\![G^{s'}[j].\text{tmp}]\!] - [\![G^{s'}[i].\text{flag}]\!]), i \in [N]$. % Multiply the received flags.
7: $[\![G^{s'}[i].\text{flag}]\!] += [\![G^{s'}[i].\text{rec}]\!], i \in [N]$. % Update the flags.
```

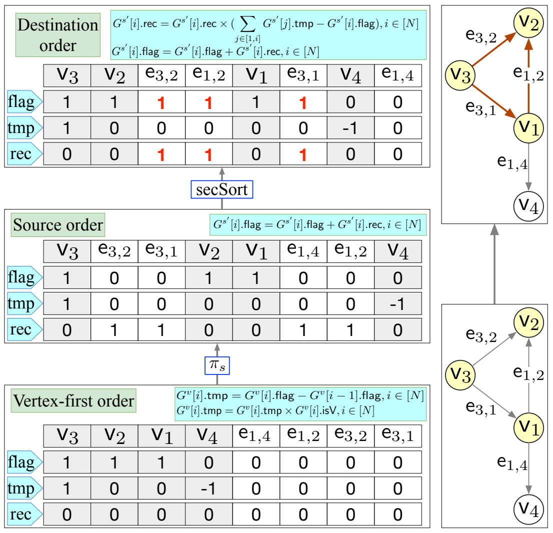
all vertices to be ordered before the edges, while maintaining the original vertex order during the oblivious message passing. Thus, unlike Graphiti [29], we adopt a more easily achievable vertex-first order as the input for Scatter. Next, the servers obliviously compute the flag to be propagated by each vertex using lines 9-10. The operation in line 10 serves to obliviously reset the propagated flags of the edges to 0, as edges themselves do not propagate flags further. The servers then transform the DAG-list from vertex-first order to source order in line 11. Finally, the servers obliviously propagate the flag from each source vertex to its outgoing edges in line 12. After Scatter, each vertex's flag is propagated and stored in rec of its outgoing edges, while rec remains 0 for all vertices.

Gather (lines 13-16). During Gather, each vertex aggregates the rec components of its incoming edges. To achieve this, the servers first convert the DAG-list from source order to destination order (line 13). Next, for each tuple, the servers aggregate the rec components from its preceding entries (line 14). However, during this process, the servers not only aggregate the rec component from each vertex's incoming edges but also the rec component accumulated by the preceding entries in the DAG-list. Therefore, the extra data gathered by each vertex must be removed. Note that this extra data corresponds to the data aggregated by the preceding vertex in the DAG-list. Thus, by converting back to vertex-first order (line 15), each vertex can remove the extra data from its rec (line 16).

A pictorial illustration of the steps in one invocation of Scatter – Gather is shown in Fig. 4. In this example, the query vertex v_3 propagates its flag 1 to its proximity vertices, v_1 and v_2. At the end of each invocation of Scatter – Gather, the servers aggregate the flag each vertex receives in this round (line 17). Note that only the vertex flags are updated, while the edge flags remain unchanged. The servers reset each tuple's rec and tmp in preparation for the next round (line 18). In the output of Algorithm 1, the vertices with flags greater than 0 are the proximity vertices.

5.2 Secure Vertex-to-Edge Flag Propagation

After propagating the flag $[\![1]\!]$ of the query vertex to the proximity vertices, the servers should obliviously propagate the proximity vertices' flags to the adjacent edges connecting them. To accomplish this, we design an oblivious dual-passing algorithm, presented in Algorithm 2 (named as secEdge). Its input is the DAG-list in vertex-first order, i.e., the output from Algorithm 1. First, the servers obliviously propagate each vertex's flag to its outgoing edges (stored in rec, lines 1-4). The rationale behind this design is the same as the



Destination order

$G^{s'}[i].\text{rec} = G^{s'}[i].\text{rec} \times (\sum_{j\in[1,i]} G^{s'}[j].\text{tmp} - G^{s'}[i].\text{flag}), i \in [N]$

$G^{s'}[i].\text{flag} = G^{s'}[i].\text{flag} + G^{s'}[i].\text{rec}, i \in [N]$

	v_3	v_2	$e_{3,2}$	$e_{1,2}$	v_1	$e_{3,1}$	v_4	$e_{1,4}$
flag	1	1	1	1	1	1	0	0
tmp	1	0	0	0	0	0	-1	0
rec	0	0	1	1	0	1	0	0

Source order

$G^{s'}[i].\text{flag} = G^{s'}[i].\text{flag} + G^{s'}[i].\text{rec}, i \in [N]$

	v_3	$e_{3,2}$	$e_{3,1}$	v_2	v_1	$e_{1,4}$	$e_{1,2}$	v_4
flag	1	0	0	1	1	0	0	0
tmp	1	0	0	0	0	0	0	-1
rec	0	1	1	0	0	1	1	0

Vertex-first order

$G^v[i].\text{tmp} = G^v[i].\text{flag} - G^v[i-1].\text{flag}, i \in [N]$

$G^v[i].\text{tmp} = G^v[i].\text{tmp} \times G^v[i].\text{isV}, i \in [N]$

	v_3	v_2	v_1	v_4	$e_{1,4}$	$e_{1,2}$	$e_{3,2}$	$e_{3,1}$
flag	1	1	1	0	0	0	0	0
tmp	1	0	0	-1	0	0	0	0
rec	0	0	0	0	0	0	0	0

Figure 5: Example of our oblivious message dual-passing.

Scatter function in Algorithm 1. However, this step only propagates each vertex's flag to its outgoing edges and does not propagate flags to incoming edges. This will result in edges with non-zero flags where only the source is a proximity vertex, while the destination is not. Some of these edges are not the required adjacent edges.

To address this, the servers additionally propagate each vertex's flag to its incoming edges (lines 5-6). Specifically, the servers first obliviously sort $[\![G^s]\!]$ in the *new destination order*, where each vertex appears immediately before all its incoming edges (e.g., $v_3, e_{1,3}, v_1, e_{2,1}, e_{1,2}, \ldots$). This can be achieved by invoking secSort$(\cdot)$ on $[\![G^s]\!]$ (line 5). Next, the servers obliviously multiply each edge's rec received from its source with the value received from its destination (line 6). This ensures that only edges which receive non-zero flags from both their source and destination will have a non-zero rec, namely the edges connecting two proximity vertices. Finally, the servers add each tuple's rec to its flag (line 7). Since the vertices do not receive any values during this process, the addition does not affect the vertex flags.

A pictorial illustration of the steps involved in a single invocation of the dual-passing method is shown in Fig. 5. In this example, the proximity vertices v_3, v_1, and v_2 propagate their flags to their adjacent edges, $e_{1,2}$, $e_{3,2}$, and $e_{1,2}$. In the output of Algorithm 2, the edges with flags greater than zero are adjacent edges. The servers then obliviously shuffle all tuples before recovering their flags, thereby identifying the proximity vertices and adjacent edges that together form the proximity subgraph. The number of tuples in the proximity subgraph is denoted as M. To prevent flag values from leaking any additional information beyond indicating whether a tuple belongs to the proximity subgraph, the servers can transform each nonzero flag to 1 by evaluating $[\![\text{flag}]\!] = \text{secCmp}([\![\text{flag}]\!], [\![1]\!])$ before flag recovery.

6 SECURE LOW-SUPPORT EDGE FILTERING

6.1 Framework

The overall process of this phase is illustrated in Fig. 6. Given the secret-shared proximity subgraph, the next step is to compute the

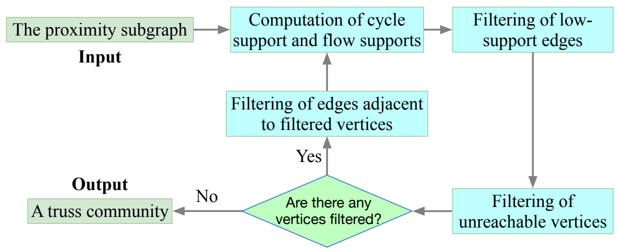

The proximity subgraph
Input
Computation of cycle support and flow supports
Filtering of low-support edges
Filtering of edges adjacent to filtered vertices
Yes
Output
A truss community
No
Are there any vertices filtered?
Filtering of unreachable vertices

Figure 6: Framework for secure low-support edge filtering.

cycle and flow supports of each edge and to filter out edges whose supports fall below the thresholds specified by the querier. Since an edge's cycle and flow supports equal the numbers of cycle and flow triangles it induces, respectively, the core challenge is to securely count triangles over the secret-shared proximity subgraph. However, directly counting triangles in this setting is nontrivial, as the computation requires access to each vertex's neighboring vertices, while the proximity subgraph is represented only as independent edges, with vertex information implicitly encoded in these edges. To address this challenge, we introduce a secure neighbor aggregation method (presented in Section 6.2) based on oblivious message passing.

After aggregating each vertex's neighbors—encoded in one-hot vector form—the servers compute the cycle and flow supports of each edge according to Eq. 1 and Eq. 2. They then filter out edges whose cycle support is smaller than k_c or whose flow support is smaller than k_f. Subsequently, the servers filter out vertices that become unreachable from the query vertex as a result of edge filtering, using oblivious message passing. If any vertices are filtered out, this indicates that the cycle or flow supports of some remaining edges may have further decreased, prompting the servers to iteratively recompute these supports. In the proximity subgraph, once a vertex is filtered out, its adjacent edges become irrelevant and must also be removed. Therefore, before recomputing cycle or flow supports, the servers first filter out edges adjacent to the removed vertices. Conversely, if no vertices are filtered out in an iteration, all remaining vertices are reachable from the query vertex, and all remaining edges satisfy the query requirements on cycle and flow supports. At this point, the remaining subgraph is returned as the truss community. We present the overall secure low-support edge filtering procedure in Section 6.3.

6.2 Secure Neighbor Aggregation

Overview. Our secure neighbor aggregation method takes as input the secret-shared proximity subgraph and outputs an updated proximity subgraph in which each vertex tuple is augmented with its secret-shared out-neighbor vector and in-neighbor vector. The core idea can be summarized as follows: (1) the servers obliviously pass each edge tuple's destination vertex ID (encoded as a one-hot vector) to the tuple corresponding to its source vertex (i.e., Scatter), and then obliviously aggregate the received IDs to obtain its out-neighbor vector (i.e., Gather); and (2) the servers obliviously pass each edge tuple's source vertex ID (encoded as a one-hot vector) to the tuple corresponding to its destination vertex (i.e., Scatter), and then obliviously aggregate the received IDs to obtain its in-neighbor vector (i.e., Gather). In these neighbor vectors, the positions of elements with value 1 correspond to the IDs of the neighboring vertices.

Algorithm 3 Secure Neighbor Aggregation

```
Input: Secret-shared proximity subgraph $\llbracket G_{\mathrm{P}} \rrbracket$.
Output: Proximity subgraph after the aggregation of neighbors.
1: $\llbracket G^v_{\mathrm{P}} \rrbracket = \mathrm{secSort}(\llbracket G_{\mathrm{P}} \rrbracket, \llbracket \mathrm{isV} \rrbracket)$. % Vertex-first order.
2: $\llbracket G^s_{\mathrm{P}} \rrbracket = \mathrm{secSort}(\llbracket G^v_{\mathrm{P}} \rrbracket)$. % Source order.
3: for all $i \in [M]$ do
4:     $\llbracket b \rrbracket = \llbracket G^s_{\mathrm{P}}[i].\mathrm{isV} \rrbracket$.
5:     for all $j \in [i+1, M]$ do
6:         $\llbracket b \rrbracket = \llbracket b \rrbracket \times \llbracket \neg G^s_{\mathrm{P}}[j].\mathrm{isV} \rrbracket$.
7:         $\llbracket G^s_{\mathrm{P}}[i].\mathrm{N}^+_{\mathrm{src}} \rrbracket += \llbracket b \rrbracket \times \llbracket G^s_{\mathrm{P}}[j].\mathbf{dst} \rrbracket$.
8:     end for
9:     $\llbracket G^s_{\mathrm{P}}[i].\mathrm{N}^+_{\mathrm{dst}} \rrbracket = \llbracket G^s_{\mathrm{P}}[i].\mathrm{N}^+_{\mathrm{src}} \rrbracket$. % Copy to the destination.
10: end for
11: $\llbracket G^d_{\mathrm{P}} \rrbracket = \mathrm{secSort}(\llbracket G^s_{\mathrm{P}} \rrbracket)$. % Destination order (vertex-first).
12: for all $i \in [M]$ do
13:     $\llbracket b \rrbracket = \llbracket G^d_{\mathrm{P}}[i].\mathrm{isV} \rrbracket$.
14:     for all $j \in [i+1, M]$ do
15:         $\llbracket b \rrbracket = \llbracket b \rrbracket \times \llbracket \neg G^d_{\mathrm{P}}[j].\mathrm{isV} \rrbracket$.
16:         $\llbracket G^d_{\mathrm{P}}[i].\mathrm{N}^-_{\mathrm{dst}} \rrbracket += \llbracket b \rrbracket \times \llbracket G^d_{\mathrm{P}}[i].\mathbf{src} \rrbracket$.
17:     end for
18:     $\llbracket G^d_{\mathrm{P}}[i].\mathrm{N}^-_{\mathrm{src}} \rrbracket = \llbracket G^d_{\mathrm{P}}[i].\mathrm{N}^-_{\mathrm{dst}} \rrbracket$. % Copy to the source.
19: end for
20: $\llbracket G^d_{\mathrm{P}}[i].\mathrm{N}^-_{\mathrm{dst}} \rrbracket += \llbracket G^d_{\mathrm{P}}[i-1].\mathrm{N}^-_{\mathrm{dst}} \rrbracket \times \llbracket \neg G^d_{\mathrm{P}}[i].\mathrm{isV} \rrbracket$, $i \in [M]$.
21: $\llbracket G^d_{\mathrm{P}}[i].\mathrm{N}^+_{\mathrm{dst}} \rrbracket += \llbracket G^d_{\mathrm{P}}[i-1].\mathrm{N}^+_{\mathrm{dst}} \rrbracket \times \llbracket \neg G^d_{\mathrm{P}}[i].\mathrm{isV} \rrbracket$, $i \in [M]$.
22: $\llbracket G^s_{\mathrm{P}} \rrbracket = \mathrm{secSort}(\llbracket G^d_{\mathrm{P}} \rrbracket)$. % Source order.
23: $\llbracket G^s_{\mathrm{P}}[i].\mathrm{N}^-_{\mathrm{src}} \rrbracket += \llbracket G^s_{\mathrm{P}}[i-1].\mathrm{N}^-_{\mathrm{src}} \rrbracket \times \llbracket \neg G^s_{\mathrm{P}}[i].\mathrm{isV} \rrbracket$, $i \in [M]$.
24: $\llbracket G^s_{\mathrm{P}}[i].\mathrm{N}^+_{\mathrm{src}} \rrbracket += \llbracket G^s_{\mathrm{P}}[i-1].\mathrm{N}^+_{\mathrm{src}} \rrbracket \times \llbracket \neg G^s_{\mathrm{P}}[i].\mathrm{isV} \rrbracket$, $i \in [M]$.
```

Preparation. Before performing secure neighbor aggregation, the servers first carry out the following preparations:

(1) To accommodate each vertex's neighboring vertices without leaking vertex degrees and to support efficient triangle counting in the secret-sharing domain, the servers store copies of each tuple's src and dst in secret-shared one-hot vector form, denoted as $\llbracket \mathbf{src} \rrbracket$ and $\llbracket \mathbf{dst} \rrbracket$. All the elements of the vectors are initialized to 0, except for the positions corresponding to the values of src and dst, which are set to 1. This transformation is achieved using secure equality rest operations [9].

(2) The servers attach four components to each tuple in the DAG-list of the proximity subgraph: $\mathrm{N}^+_{\mathrm{src}}$ to store the source's out-neighbors, $\mathrm{N}^-_{\mathrm{src}}$ to store the source's in-neighbors, $\mathrm{N}^+_{\mathrm{dst}}$ to store the destination's out-neighbors, and $\mathrm{N}^-_{\mathrm{dst}}$ to store the destination's in-neighbors. All four components are 0 vector.

Detailed method. Algorithm 3 (denoted as secNeigh) presents our secure neighbor aggregation method. In this method, the servers first obliviously permute the proximity subgraph into source order (lines 1–2). Then, the servers obliviously aggregate each vertex's out-neighbors into its $\mathrm{N}^+_{\mathrm{src}}$ and $\mathrm{N}^+_{\mathrm{dst}}$ (lines 3-10). Here, we introduce a label $\llbracket b \rrbracket$ to ensure that each vertex's $\mathrm{N}^+_{\mathrm{src}}$ aggregates only the vertex's out-neighbors, ignoring others (line 4). During neighbor aggregation, the servers first obliviously update b (line 6), and then accumulate $G^s_{\mathrm{P}}[j].\mathbf{dst}$, $j \in [i+1, M]$ to $\mathrm{N}^+_{\mathrm{src}}$ (line 7). The correctness

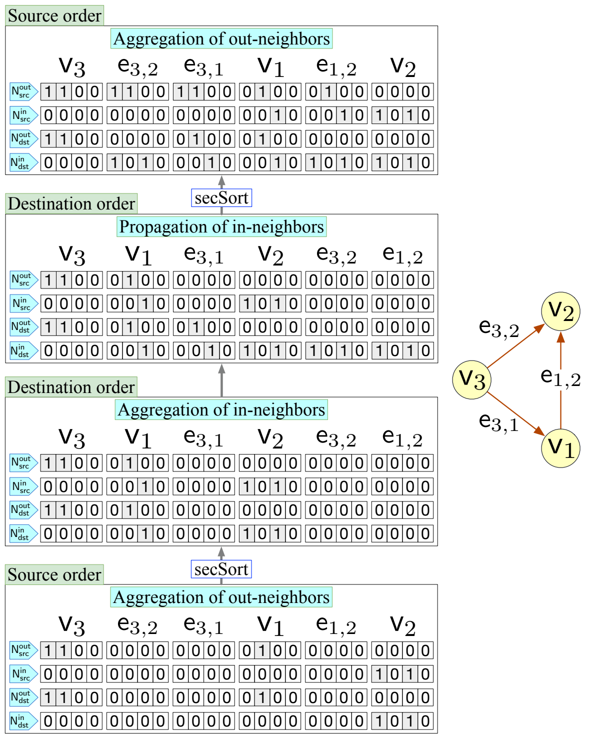


Figure 7: Example of the neighbor aggregation method.

Algorithm 4 Secure Low-Support Edge Filtering

Input: Proximity subgraph $[\![G_P]\!]$; support thresholds $([\![k_c]\!], [\![k_f]\!])$.
Output: The secret-shared truss community.

```
1: $[\![G_P^s]\!] = \text{secNeigh}([\![G_P]\!])$. % Algorithm 3.
2: repeat
3:     $[\![G_P^v]\!] = \text{secSort}([\![G_P^s]\!], [\![\text{isV}]\!])$. % Vertex-first order.
4:     for all $i \in [N-E, N]$ do
5:         $[\![\text{sup}_c]\!] = [\![G_P^v[i].\text{N}_{\text{src}}^-]\!] \cdot [\![G_P^v[i].\text{N}_{\text{dst}}^+]\!]$. % Cycle support
6:         $[\![\mathbf{a}]\!] = [\![G_P^v[i].\text{N}_{\text{src}}^-]\!] \odot [\![G_P^v[i].\text{N}_{\text{dst}}^-]\!]$.
7:         $[\![\mathbf{b}]\!] = [\![G_P^v[i].\text{N}_{\text{src}}^+]\!] \odot [\![G_P^v[i].\text{N}_{\text{dst}}^-]\!]$.
8:         $[\![\mathbf{c}]\!] = [\![G_P^v[i].\text{N}_{\text{src}}^+]\!] \odot [\![G_P^v[i].\text{N}_{\text{dst}}^+]\!]$
9:         $[\![\mathbf{d}]\!] = [\![\mathbf{a}]\!] + [\![\mathbf{b}]\!] - [\![\mathbf{a}]\!] \odot [\![\mathbf{b}]\!]$.
10:        $[\![\text{sup}_f]\!] = \text{sum}([\![\mathbf{d}]\!] + [\![\mathbf{c}]\!] - [\![\mathbf{d}]\!] \odot [\![\mathbf{c}]\!])$. % Flow support
11:        $[\![\text{isFltr}]\!] = \text{secCmp}([\![\text{sup}_c]\!], [\![k_c]\!]) \times \text{secCmp}([\![\text{sup}_f]\!], [\![k_f]\!])$.
12:        $[\![G_P^v[i].\text{isFltr}]\!] = [\![\text{isFltr}]\!] + [\![G_P^v[i].\text{isV}]\!] - [\![\text{isFltr}]\!] \times [\![G_P^v[i].\text{isV}]\!]$.
13:    end for
14:    Securely shuffle the edge tuples in $[\![G_P^v]\!]$, recover each edge
       tuple's isFltr, and filter out tuples with isFltr = 1.
15:    $(\pi_s, [\![G_P^v]\!]) = \text{secVertex}([\![G_P^v]\!], r)$. % Algorithm 1.
16:    Securely shuffle the vertex tuples in $[\![G_P^v]\!]$, recover each vertex
       tuple's flag, and filter out tuples with flag = 0.
17:    $[\![G_P^v]\!] = \text{secEdge}([\![G_P^v]\!], \pi_s)$. % Algorithm 2.
18:    Securely shuffle the edge tuples in $[\![G_P^v]\!]$, recover each edge
       tuple's flag, and filter out tuples with flag = 0.
19: until No vertex tuples are filtered
20: return The truss community $[\![G_P^v]\!]$.
```

of this step is as follows: For each tuple representing a vertex, b is initialized as 1, and it is set to 0 when encountering the next isV = 1 (line 6), which marks the boundary of the next vertex. This ensures that each vertex's N_{src}^+ only aggregates the vertex's out-neighbors and ignores out-neighbors of other vertices. For each tuple representing an edge, as its isV $=$ 0, b is initialized as 0, ensuring that the edge's N_{src}^+ remains unchanged. At the end of the iteration for a tuple, the servers copy its N_{src}^+ to N_{dst}^+ (line 9), as each vertex tuple's source is identical to its destination.

Next, the servers aggregate each vertex's in-neighbors into its N_{dst}^- and N_{src}^- (lines 11-19). The process for aggregating in-neighbors is similar to that of the out-neighbors. In contrast, before aggregating the in-neighbors, the servers first obliviously sort G_P^s into the destination order (line 11), where each vertex appears immediately before all its incoming edges.

Finally, the servers obliviously propagate each vertex's in-neighbors and out-neighbors to the corresponding edges' sources and destinations (lines 20-24). At the start of this process, the proximity subgraph is in the destination order (sorted by line 11). The servers first obliviously propagate each vertex's in-neighbors and out-neighbors to the corresponding edges' destinations. This is done by copying each tuple's N_{dst}^- and N_{dst}^+ to those of the next tuple (lines 20-21). The multiplication with ¬isV prevents the copying of a vertex's neighbors to other vertices. Afterwards, the servers obliviously sort the proximity subgraph back into the source order and propagate each vertex's in-neighbors and out-neighbors to the corresponding edges' sources (lines 22-24).

A pictorial illustration of the steps involved in the neighbor aggregation method is provided in Fig. 7. In the output of Algorithm 3, each tuple's $\text{N}_{\text{src}}^+, \text{N}_{\text{src}}^-, \text{N}_{\text{dst}}^+, \text{N}_{\text{dst}}^-$ store the source's out-neighbors, source's in-neighbors, destination's out-neighbors, and destination's in-neighbors, respectively. Next, we detail how to compute each edge supports based on these neighbor vectors.

6.3 Secure Low-Support Edge Filtering

After the secure aggregation of neighboring vertices described in Section 6.2, each vertex tuple's $[\![\text{N}_{\text{src}}^+]\!]$ and $[\![\text{N}_{\text{dst}}^+]\!]$ store its out-neighbors, and each vertex tuple's $[\![\text{N}_{\text{src}}^-]\!]$ and $[\![\text{N}_{\text{dst}}^-]\!]$ store its in-neighbors. Specifically, N_*^* is vector whose entries with value 1 correspond to the IDs of neighboring vertices. In this section, we describe how to securely compute secret-shared cycle and flow supports based on the aggregated neighbor vectors, and then securely filter out low-support edges and their incident vertices. Algorithm 4 (denoted as secFltr) presents the overall method, which follows the framework illustrated in Fig. 6.

First, to compute each edge's supports, the servers must determine which tuples correspond to edges. To achieve this, the servers first obliviously sort the DAG-list into vertex-first order (line 3), after which the last E tuples correspond to edges. The value of E can be inferred from the size of the initial snapshot graph and the number of edges that have been filtered. They then compute each edge's secret-shared cycle and flow supports using Eq. 1 and Eq. 2 (lines 5–10). Next, the servers check whether each edge's supports satisfy the thresholds required by the querier (line 11); tuples with isFltr = 1 are marked for removal. To prevent vertex tuples from being incorrectly removed, the servers obliviously correct their isFltr values in line 12, ensuring that all vertex tuples have isFltr = 0 and thus will not be filtered. Before revealing the isFltr bits, the servers obliviously shuffle the DAG-list to hide access patterns.

After low-support edges are removed, the servers proceed to securely filter out vertices that become unreachable due to edge filtering. The key intuition is that every vertex in the final community must be reachable from the query vertex within radius r. To enforce this, the servers apply Algorithm 1 to determine which vertices remain reachable. The servers then obliviously shuffle the DAG-list and reveal each tuple's flag to identify and remove any vertices that are unreachable from the query vertex. Once any vertices are removed, their adjacent edges must also be removed. To securely filter these edges, the servers use Algorithm 2 to identify edges adjacent to removed vertices. Finally, after an oblivious shuffle, the servers reveal flag and remove all edges whose flags are 0.

7 SECURITY ANALYSIS

We use the simulation paradigm [31] to analyze the security guarantees of Cove. Proving security within this paradigm involves establishing two worlds: the real world, where the actual protocol is executed by honest parties, and the ideal world, where an ideal functionality $\mathcal{F}$ takes inputs from the parties and directly outputs the result to the relevant party. We define a probabilistic polynomial-time (PPT) simulator, who only accesses the leakage of $\mathcal{F}$ as claimed by our protocols, and "simulates" messages that resemble those sent by the honest parties to the corrupted server in the real world. If a PPT adversary cannot distinguish between the real and ideal worlds, we conclude that Cove is secure. The details of the security analysis are presented in Appendix B.

8 EVALUATION

In our evaluation, we aim to answer the following questions:

• *How does Cove perform compared to the baseline constructed using a generic MPC framework?* (Section 8.2)

• *How is Cove's performance affected by different parameter settings?* (Section 8.3)

• *What is the performance breakdown of individual components in Cove?* (Appendix C)

8.1 Experimental Setup

We implement a prototype of Cove using Python. The server-side experiments are conducted on a workstation equipped with an Intel Xeon Gold 6238R CPU, 128 GB of RAM, and 1 TB of external SSD storage. Each server in Cove is simulated by a separate process with a single thread. Consistent with prior MPC-based works [29, 55], we emulate server-side communication on a single workstation using the loopback filesystem, with a 10 ms network delay injected via the Linux tc command. Additionally, we use a MacBook Air with 8 GB of RAM as the data producer and querier to upload the streaming graph and queries to the workstation. Following the setup in previous work without privacy protection [30], we set the slide interval $\beta = \frac{W}{1000}$.

Datasets. We use three real-world streaming graph datasets of varying scales in our experiments: (1) MOOC user action[1]: This dataset represents a small, dense network, consisting of 7,143 vertices and 411,749 temporal edges. (2) Reddit hyperlink network[2]: This dataset is of medium size and sparse, containing 55,863 vertices and 858,490 temporal edges. (3) com-DBLP[3]: This dataset is a large, sparse network containing 317,080 vertices and 1,049,866 edges.

Baseline. As discussed in Section 1, no prior work addresses the specific problem we are targeting. While our approach builds upon the oblivious message passing technique in [29], [29] only supports secure proximity vertex search over static graphs and does not handle secure proximity subgraph search or truss community search over streaming graphs. So it cannot serve as a suitable baseline for performance comparison. To this end, we construct a baseline using the widely adopted generic MPC framework MP-SPDZ [28] for secure truss community search over streaming graphs. This allows us to fairly evaluate the effectiveness of our custom design in Cove. In the baseline, we also use the DAG-list structure to represent the streaming graph, and adopt the same system model as Cove. The baseline employs an exhaustive strategy to search for the truss community in each snapshot graph. Specifically, it first performs an oblivious shuffle on the DAG-list to randomize the tuple permutations. Then, it executes a secure breadth-first search on the snapshot graph, using an insecure equality test where the test results are revealed. Vertices and edges outside the target radius of the query vertex are filtered. The baseline then obliviously computes the cardinality of the intersection of the source and destination neighbors for each edge. Next, the baseline computes the cycle support and flow support of each edge based on the cardinalities in the secret sharing domain. It then compares each edge's cycle and flow support to the target values, then recovers the results to filter out edges. This process repeats until no tuples are filtered.

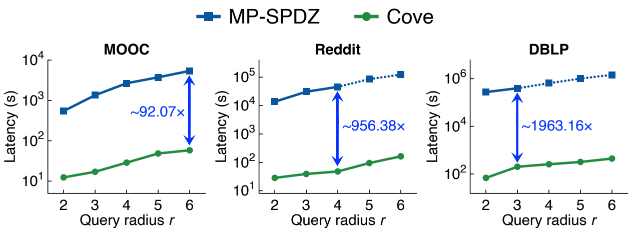


Figure 8: Search latency of Cove and the baseline.

8.2 Comparison with the Baseline

In the comparative experiment, we fix the window size at W = 50,000 and set (k_c = 3, k_f = 3), varying only the query radius $r \in \{2, 3, 4, 5, 6\}$, following the setting in prior plaintext work [30].

Latency. Fig. 8 compares the search latency of Cove with the baseline across varying query radius $r \in \{2, 3, 4, 5, 6\}$ on different datasets[4]. As expected, the latency of both Cove and the baseline increases with larger query radius, following a similar trend due to their costs scaling roughly with r^2. Despite this shared growth pattern, Cove consistently outperforms the baseline by a substantial margin: achieving speedups of approximately 92.07× on the MOOC

[1]https://snap.stanford.edu/data/act-mooc.html

[2]https://snap.stanford.edu/data/soc-RedditHyperlinks.html

[3]https://snap.stanford.edu/data/com-DBLP.html

[4]For cases where the baseline is extremely slow, its latency is estimated based on trend extrapolation and shown as a dotted line.

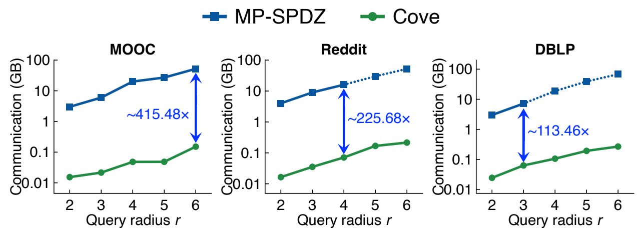


Figure 9: Cost of Cove and the baseline.

dataset, 956.38× on Reddit, and 1963.16× on DBLP. Moreover, the performance gap between Cove and the baseline widens as both the query radius r and dataset size increase. This is because, in order to check vertex reachability, the baseline must rely on a secure breadth-first search using equality tests, which incurs a computational cost on the order of $O(N^2)$. In contrast, Cove determines vertex reachability efficiently based on the concept of oblivious message passing, with a much lower complexity of only $O(r \cdot N)$. The results highlight the superior scalability of Cove.

Server-side communication. In Fig. 9, we compare the server-side communication cost of Cove and the baseline under varying query radius $r \in \{2, 3, 4, 5, 6\}$. As with the search latency results, the communication cost for both Cove and the baseline increases with larger query radius, following a similar trend due to their costs scaling approximately with r^2. Despite this shared growth pattern, Cove consistently achieves significantly lower communication overhead compared to the baseline, with improvements of approximately 415.48× on the MOOC dataset, 225.68× on the Reddit dataset, and 113.46× on the DBLP dataset. These results reveal that the communication efficiency advantage of Cove is more pronounced on smaller datasets but narrows on larger ones. This is because Cove, in order to perform efficient oblivious neighbor aggregation (i.e., Algorithm 3), requires each vertex to be transformed into a one-hot vector representation. The communication cost of this transformation is linear in the number of vertices, making Cove more sensitive to the graph size in terms of vertex count. However, even on the large-scale DBLP dataset, Cove still outperforms the baseline, due to its efficient oblivious message passing based protocol.

8.3 Performance under Varying Parameters

In the design of Cove, the primary parameters that affect performance are the window size W, the thresholds (k_c, k_f), and the query radius r. Since the effect of r has already been analyzed, we now focus on evaluating the remaining two factors.

Effect of window size. In this experiment, we evaluate the search latency and server-side communication cost of Cove by varying the window size, with fixed parameters ($k_c = 3, k_f = 3$) and $r = 3$. The results are shown in Fig. 10. As expected, both search latency and communication cost increase approximately linearly with the window size. This is because a larger window size results in a larger snapshot graph, which requires more computation and communication to process. Additionally, we observe that the increase in cost is sub-linear with respect to the total number of vertices in the dataset. This is because, for a fixed window size, the number of tuples in each snapshot graph remains relatively stable. As a result, the overall number of vertices in the full graph has limited

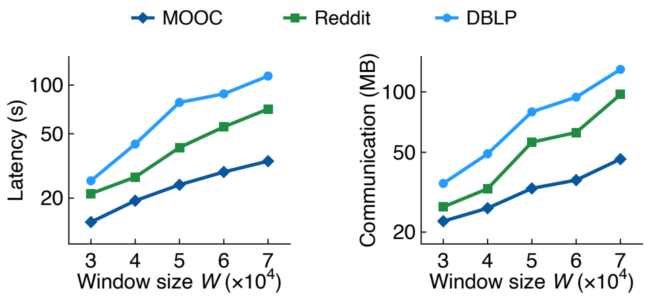


Figure 10: Performance of Cove under varying window sizes.

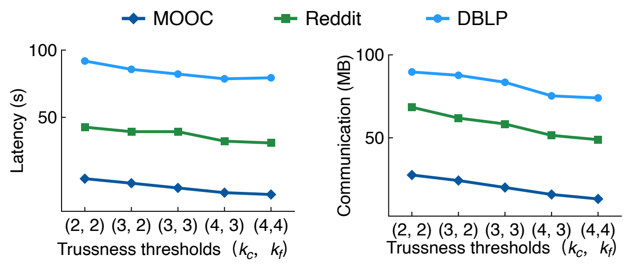


Figure 11: Search latency and server-side communication cost of Cove under varying values of k_c and k_f.

impact on the performance, confirming the efficiency of Cove even on larger datasets.

Effect of thresholds k_c and k_f. We evaluate the impact of parameters k_c and k_f on the search latency and server-side communication cost of Cove, with fixed settings of $W = 50{,}000$ and $r = 3$. The results are shown in Fig. 11. Overall, we observe that the effect of k_c and k_f on the cost of Cove is relatively minor. This is because these parameters only influence the secure edge filtering phase, which involves a limited number of edges from the proximity graph. As a result, the overall contribution of this phase to the total cost remains small. Additionally, we note that both search latency and communication cost tend to slightly decrease as k_c and k_f increase. This is because higher thresholds for cycle and flow support makes it more difficult for edges to meet the requirements, causing more edges to be filtered out in the early rounds of the filtering process. Hence, with larger k_c and k_f, edge filtering becomes more efficient, leading to improved overall performance.

9 CONCLUSION

We present Cove, the first system enabling oblivious community search on outsourced streaming graphs. At the core of Cove is a delicate synergy of insights on lightweight cryptography, graph modeling, and data encoding. Extensive evaluations on real-world datasets demonstrate that Cove can process a snapshot graph within a few minutes (with the time window size being 50,000). Compared to a baseline built on generic MPC, Cove achieves up to a 1963× speedup in search latency and up to a 415× reduction in communication cost. We believe that Cove provides a valuable step in pushing forward the frontier of research on oblivious query processing over streaming graphs.

REFERENCES

[1] Airbnb on AWS. 2018. AWS Case Study: Airbnb. https://aws.amazon.com/solutions/case-studies/airbnb/?nc1=h_ls. [Online; Accessed 9-Jul-2025].

[2] Apple and Google. 2021. Exposure notification privacy-preserving analytics white paper. https://covid19-static.cdn-apple.com/applications/covid19/current/static/contact-tracing/pdf/ENPA_White_Paper.pdf.

[3] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. 2016. High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority. In *Proc. of ACM CCS*.

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

[5] ArangoDB. 2024. ArangoGraph-Where Cloud Meets Performance. https://cloud.google.com/blog/products/data-analytics/keep-graph-data-updated-on-google-cloud-using-confluent--neo4j.

[6] 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 *Proc. of ACM CCS*.

[7] James Bell, Adria Gascon, Badih Ghazi, Ravi Kumar, Pasin Manurangsi, Mariana Raykova, and Phillipp Schoppmann. 2022. Distributed, Private, Sparse Histograms in the Two-Server Model. In *Proc. of ACM CCS*.

[8] 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 *Proc. of EUROCRYPT*.

[9] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2016. Function secret sharing: Improvements and extensions. In *Proc. of ACM CCS*.

[10] Xihui Chen, Sjouke Mauw, and Yunior Ramírez-Cruz. 2020. Publishing Community-Preserving Attributed Social Graphs with a Differential Privacy Guarantee. *Proceedings on Privacy Enhancing Technologies* 2020, 4 (2020).

[11] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. 2018. SPD: efficient MPC mod for dishonest majority. In *Annual International Cryptology Conference*.

[12] Max Curran, Xiao Liang, Himanshu Gupta, Omkant Pandey, and Samir R Das. 2019. Procsa: Protecting privacy in crowdsourced spectrum allocation. In *Proc. of ESORICS*.

[13] Michael Curtiss, Iain Becker, Tudor Bosman, Sergey Doroshenko, Lucian Grijincu, Tom Jackson, Sandhya Kunnatur, Søren B. Lassen, Philip Pronin, Sriram Sankar, Guanghao Shen, Gintaras Woss, Chao Yang, and Ning Zhang. 2013. Unicorn: A System for Searching the Social Graph. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1150–1161.

[14] Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. 2006. Searchable symmetric encryption: improved definitions and efficient constructions. In *Proc. of ACM CCS*.

[15] Emma Dauterman, Mayank Rathee, Raluca Ada Popa, and Ion Stoica. 2022. Waldo: A Private Time-Series Database from Function Secret Sharing. In *Proc. of IEEE S&P*.

[16] Francesca Falzon, Esha Ghosh, Kenneth G Paterson, and Roberto Tamassia. 2024. PathGES: An Efficient and Secure Graph Encryption Scheme for Shortest Path Queries. In *Proc. of ACM CCS*.

[17] Yixiang Fang, Xin Huang, Lu Qin, Ying Zhang, Wenjie Zhang, Reynold Cheng, and Xuemin Lin. 2020. A survey of community search over big graphs. *The VLDB Journal* 29 (2020), 353–392.

[18] Fireblocks. [n. d.]. Remove the complexity of working with digital assets. online at https://www.fireblocks.com. [Online; Accessed 9-Jul-2025].

[19] Esha Ghosh, Seny Kamara, and Roberto Tamassia. 2021. Efficient Graph Encryption Scheme for Shortest Path Queries. In *Proc. of ACM AsiaCCS*.

[20] Google Cloud. 2023. Streaming graph data with Confluent Cloud and Neo4j on Google Cloud. https://cloud.google.com/blog/products/data-analytics/keep-graph-data-updated-on-google-cloud-using-confluent--neo4j.

[21] Xiangyang Gou and Lei Zou. 2021. Sliding window-based approximate triangle counting over streaming graphs with duplicate edges. In *Proc. of ACM SIGMOD*.

[22] Tingxuan Han, Wei Tong, Jiacheng Niu, and Sheng Zhong. 2023. Practical Privacy-Preserving Community Detection in Decentralized Weighted Networks. In *International Conference on Security and Privacy in Communication Systems*. 302–320.

[23] Kai Huang, Haibo Hu, Shuigeng Zhou, Jihong Guan, Qingqing Ye, and Xiaofang Zhou. 2022. Privacy and efficiency guaranteed social subgraph matching. *The VLDB Journal* 31, 3 (2022), 581–602.

[24] Jacob Imola, Takao Murakami, and Kamalika Chaudhuri. 2022. Communication-Efficient triangle counting under local differential privacy. In *Proc. of Usenix Security*.

[25] Jacob Imola, Takao Murakami, and Kamalika Chaudhuri. 2022. Differentially private triangle and 4-cycle counting in the shuffle model. In *Proc. of ACM CCS*.

[26] Jiaxin Jiang, Peipei Yi, Byron Choi, Zhiwei Zhang, and Xiaohui Yu. 2016. Privacy-preserving reachability query services for massive networks. In *Proc. of ACM CIKM*.

[27] Peipei Jiang, Qian Wang, Muqi Huang, Cong Wang, Qi Li, Chao Shen, and Kui Ren. 2021. Building in-the-cloud network functions: Security and privacy challenges. *Proc. IEEE* 109, 12 (2021), 1888–1919.

[28] Marcel Keller. 2020. MP-SPDZ: A versatile framework for multi-party computation. In *Proc. of ACM CCS*.

[29] Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, and Bhavish Raj Gopal. 2024. Graphiti: Secure Graph Computation Made More Scalable. In *Proc. of ACM CCS*.

[30] Xuankun Liao, Qing Liu, Xin Huang, and Jianliang Xu. 2024. Truss-based community search over streaming directed graphs. *Proceedings of the VLDB Endowment* 17, 8 (2024), 1816–1829.

[31] Yehuda Lindell. 2017. How to Simulate It - A Tutorial on the Simulation Proof Technique. In *Tutorials on the Foundations of Cryptography*. 277–346.

[32] Qing Liu, Minjun Zhao, Xin Huang, Jianliang Xu, and Yunjun Gao. 2020. Truss-based community search over large directed graphs. In *Proc. of ACM SIGMOD*.

[33] Shang Liu, Yang Cao, Takao Murakami, Jinfei Liu, and Masatoshi Yoshikawa. 2024. CARGO: Crypto-Assisted Differentially Private Triangle Counting without Trusted Servers. In *Proc. of IEEE ICDE*. 1671–1684.

[34] Toni Mastelic, Ariel Oleksiak, Holger Claussen, Ivona Brandic, Jean-Marc Pierson, and Athanasios V Vasilakos. 2014. Cloud computing: Survey on energy efficiency. *Comput. Surveys* 47, 2 (2014), 1–36.

[35] Meta. 2025. The value of secure multi-party computation. online at https://privacytech.fb.com/multi-party-computation/. [Online; Accessed 9-Jul-2025].

[36] Mozilla Security Blog. 2019. Next steps in privacy-preserving Telemetry with Prio. https://blog.mozilla.org/security/2019/06/06/next-steps-in-privacy-preserving-telemetry-with-prio/.

[37] MPCVault. 2025. Multisig crypto wallet for business. online at https://mpcvault.com. [Online; Accessed 9-Jul-2025].

[38] NEO4J CLOUD. 2025. The world's most popular graph data platform, available for any cloud environment. https://neo4j.com/cloud/.

[39] PIXNET on AWS. 2014. AWS Case Study: PIXNET. https://aws.amazon.com/solutions/case-studies/pixnet/. [Online; Accessed 9-Jul-2025].

[40] Amrita Roy Chowdhury, Chenghong Wang, Xi He, Ashwin Machanavajjhala, and Somesh Jha. 2020. Cryptϵ: Crypto-assisted differential privacy on untrusted servers. In *Proc. of ACM SIGMOD*.

[41] Safeheron. 2025. Safeheron MPC Wallet. https://safeheron.com.

[42] Yunjiao Song, Xinrui Ge, and Jia Yu. 2023. Privacy-preserving reachability query over graphs with result verifiability. *Computers & Security* 127 (2023), 103092.

[43] Fangyuan Sun, Jia Yu, and Jiankun Hu. 2024. Privacy-Preserving Approximate Minimum Community Search on Large Networks. *IEEE Transactions on Information Forensics and Security* 19 (2024), 4146–4160.

[44] Sijun Tan, Brian Knott, Yuan Tian, and David J Wu. 2021. CryptGPU: Fast Privacy-Preserving Machine Learning on the GPU. In *Proc. of IEEE S&P*.

[45] Chenghong Wang, Johes Bater, Kartik Nayak, and Ashwin Machanavajjhala. 2022. IncShrink: architecting efficient outsourced databases using incremental mpc and differential privacy. In *Proc. of ACM SIGMOD*.

[46] Songlei Wang, Yifeng Zheng, and Xiaohua Jia. 2024. GraphGuard: Private Time-Constrained Pattern Detection Over Streaming Graphs in the Cloud. In *Proc. of Usenix Security*.

[47] Zuan Wang, Xiaofeng Ding, Hai Jin, and Pan Zhou. 2022. Efficient secure and verifiable location-based skyline queries over encrypted data. *Proceedings of the VLDB Endowment* 15, 9 (2022), 1822–1834.

[48] Yulin Wu, Lanxiang Chen, Gaolin Chen, Yi Mu, and Robert H Deng. 2025. Private Reachability Queries on Structured Encrypted Temporal Bipartite Graphs. *IEEE Transactions on Dependable and Secure Computing* 22, 2 (2025), 1810–1826.

[49] Yubao Wu, Ruoming Jin, Jing Li, and Xiang Zhang. 2015. Robust local community detection: on free rider effect and its elimination. *Proceedings of the VLDB Endowment* 8, 7 (2015), 798–809.

[50] Yingying Wu, Jiabei Wang, Dandan Xu, and Yongbin Zhou. 2024. Spidey: Secure Dynamic Encrypted Property Graph Search With Lightweight Access Control. *IEEE Internet of Things Journal* (2024).

[51] Lyu Xu, Byron Choi, Yun Peng, Jianliang Xu, and Sourav S Bhowmick. 2023. A Framework for Privacy Preserving Localized Graph Pattern Query Processing. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.

[52] Tianyang Xu, Zhao Lu, and Yuanyuan Zhu. 2022. Efficient triangle-connected truss community search in dynamic graphs. *Proceedings of the VLDB Endowment* 16, 3 (2022), 519–531.

[53] Sen Zhang, Weiwei Ni, and Nan Fu. 2020. Community preserved social graph publishing with node differential privacy. In *Proc. of IEEE ICDM*.

[54] Yanping Zhang, Johes Bater, Kartik Nayak, and Ashwin Machanavajjhala. 2023. Longshot: Indexing growing databases using MPC and differential privacy. *Proceedings of the VLDB Endowment* 16, 8 (2023), 2005–2018.

[55] Zhenhua Zou, Zhuotao Liu, Jinyong Shan, Qi Li, Ke Xu, and Mingwei Xu. 2025. RingSG: Optimal Secure Vertex-Centric Computation for Collaborative Graph Processing. In *Proc. of ACM CCS*.

A LIMITATIONS AND FUTURE WORK

As the first research effort on oblivious community search over outsourced streaming graphs, the current design of Cove inevitably leaks certain count information during the search process. But we note that extensions can be made to mitigate this leakage. Specifically, the servers can pad dummy tuples before each step of the flag-revealing process. The servers then obliviously shuffle the tuples before revealing the tuples, making it indistinguishable whether a flag corresponds to a real or dummy tuple. Each dummy edge is constructed with either its src or dst field (or both) set to -1, ensuring that it is not interpreted as part of the real graph and does not interfere with the correctness of the community search. A more detailed exploration is left to our future work.

In addition, it would also be interesting to extend Cove along two key directions. The first is on the exploration of techniques to hide timestamps, such as using batch updates to obscure precise temporal information. The second is on the exploration of enhancing Cove to achieve malicious security, by incorporating techniques like information-theoretic message authentication codes [11].

B SECURITY ANALYSIS

Ideal functionality. The ideal functionality $\mathcal{F}$ for truss community search on streaming graphs is formalized as follows. (1) Query: The querier sends the query $Q = \{\mathsf{q}, r, (k_c, k_f), W, \beta\}$ to $\mathcal{F}$.

(2) Append: Each data producer sends a newly emerged edge $(\mathsf{u}, \mathsf{v}, \mathsf{t})$ to $\mathcal{F}$.

(3) Search: $\mathcal{F}$ performs truss community search for Q on each snapshot of the streaming graph.

(4)Output: If any truss community g is detected, $\mathcal{F}$ outputs g to the querier. Otherwise, $\mathcal{F}$ outputs nothing (i.e., $\perp$).

Note that $\mathcal{F}$ does not output anything to the servers or data producers. We allow $\mathcal{F}$ to leak $\mathcal{L}(\mathcal{F}(Q)) = (\{\mathsf{t}\}, \{\pi\}, \mathsf{Num}, r, W, \beta)$ to the servers. Here, $\mathsf{Num} = (N, M, \{N^v_{\mathsf{fltr}}\}, \{N^e_{\mathsf{fltr}}\})$, where N and M represent the number of tuples in the snapshot graph and the proximity subgraph, respectively, and $\{N^v_{\mathsf{fltr}}\}$ and $\{N^e_{\mathsf{fltr}}\}$ denote the numbers of filtered vertices and edges, respectively.

Definition 6. *Let $\prod$ denote the protocol for privacy-preserving truss community search, where the querier provides the query Q as input. Let $\mathcal{A}$ be an adversary who statically corrupts one server and let $\mathsf{View}^{\prod(Q)}_{\mathsf{Real}}$ represent the view of the corrupted server during the protocol execution. In the ideal world, a simulator generates a simulated view $\mathsf{View}^{\mathcal{L}(\mathcal{F}(Q))}_{\mathsf{Ideal}}$ using only the leakage $\mathcal{L}(\mathcal{F}(Q))$. We say that $\prod$ is secure if there exists a PPT simulator such that $\mathsf{View}^{\mathcal{L}(\mathcal{F}(Q))}_{\mathsf{Ideal}}$ is indistinguishable from $\mathsf{View}^{\prod(Q)}_{\mathsf{Real}}$.*

Proposition 1. *Based on Definition 6, Cove securely realizes $\mathcal{F}$ with the leakage $\mathcal{L}(\mathcal{F}(Q))$.*

Proof. Since the roles of S_{123} in Cove are symmetric, it suffices to prove the existence of simulator for S_1. During the Query and Append phases, S_1 receives only the secret shares of private values in the query and the streaming graph, along with the claimed leakage $(\{\mathsf{t}\}, M, r, W, \beta)$. So we can trivially construct the simulator by invoking the RSS simulator with $(\{\mathsf{t}\}, M, r, W, \beta)$ as input. In addition, since S_1 does not observe any information during the Output phase, there is no need to simulate its view in this phase. Search is realized by Algorithm 1 (secVertex), Algorithm 2 (secEdge), Algorithm 3 (secNeigh), and secure vertex and edge filtering, Algorithm 4 (secFltr). Since they are invoked in order and their inputs and outputs are secret shares, we analyze the existence of their simulators separately [12].

Simulator for secVertex**:** At the beginning of secVertex (Algorithm 1), S_1 holds $\{(\langle\mathsf{src}\rangle_{\{1,2\}}, \langle\mathsf{dst}\rangle_{\{1,2\}}, \langle\mathsf{isV}\rangle_{\{1,2\}}, \langle\mathsf{flag}\rangle_{\{1,2\}}, \langle\mathsf{tmp}\rangle_{\{1,2\}}, \langle\mathsf{rec}\rangle_{\{1,2\}})\}$ of the DAG-list along with the public radius r. Later, S_1 receives secret shares during the execution of secure sorting (line 2), and the simulator can be constructed by invoking the simulator for secure sorting [6]. Note that, since S_1 is unaware of the number of vertices and each tuple's component isV is treated as the sorted key, the secure sorting process can be viewed as an oblivious shuffle on the DAG-list. Subsequently, S_1 learns the permutations π_s, π_e, π_d from the insecure sorting processes (lines 3-7). Since these permutations are calculated on the shuffled DAG-list using independent sort keys, they are uniformly random in the view of S_1 and can be simulated given M. The remainder of Algorithm 1 consists of basic operations in the secret-sharing domain. Their simulator can be trivially constructed by invoking the RSS simulator [3] with the public radius r as input. Therefore, the simulator secVertex exists.

Simulator for secEdge**.** Note that secEdge (Algorithm 2) only consists secure sorting and of basic operations in the secret sharing domain. Therefore, the simulator secEdge can be trivially constructed by invoking the simulators for the secure sorting [6] and the RSS [3]. In addition, S_1 learns the flag of each tuple in the output of secEdge. The simulator can simulate the revealed bit-string $\{\mathsf{flag}\}$ as follows. Firstly, the simulator holds the number of tuples in the snapshot graph, i.e., N, which corresponds to the size of $\{\mathsf{flag}\}$. Since the proximity subgraph are produced from the snapshot graph, which is produced from the original snapshot graph after secure sort, the positions of 1s in $\{\mathsf{flag}\}$ are random. Therefore, the simulator can first generate an all-zeros string with a length of N, and then randomly select M (i.e., the number of tuples in the proximity subgraph) positions in the string to replace the 0s with 1s.

Simulator for secNeigh**.** Note that secNeigh (Algorithm 3) only consists secure sorting and basic operations in the secret sharing domain. Therefore, the simulator secNeigh can be trivially constructed by invoking the simulators for the secure sorting [6] and the RSS [3]. However, S_1 learns the flag of each tuple in the output of secNeigh. The simulator can be constructed as that for secEdge.

Simulator for secFltr**.** The secFltr phase mainly involves secure shuffling and basic operations in the secret sharing domain, except for the recovery of isFltr and flag values used to identify filtered vertices and edges. As such, a simulator for secFltr can be straightforwardly constructed by composing the simulators for secure shuffling [4] and RSS operations [3]. For the recovery of values, the simulator can follow the same approach as used in the simulator for secEdge, taking as input the number of filtered vertices and edges in each round, i.e., $\{N^v_{\mathsf{fltr}}\}$ and $\{N^e_{\mathsf{fltr}}\}$. □

C PERFORMANCE BREAKDOWN OF COVE

Cove comprises four key components: secVertex, secEdge, secNeigh, and secFltr. In the experiment, we fix the window size at $W = 50{,}000$ and set $(k_c = 3, k_f = 3)$, varying only the query radius $r \in \{2, 3, 4, 5, 6\}$.

Latency. We present a detailed breakdown of the search latency for each component of our protocol in Fig. 12. The results show that search latency is predominantly driven by the oblivious neighbor

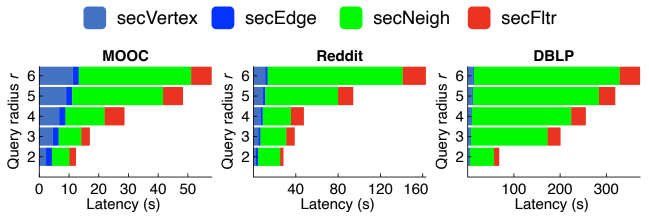


Figure 12: Component-wise breakdown of latency in Cove.

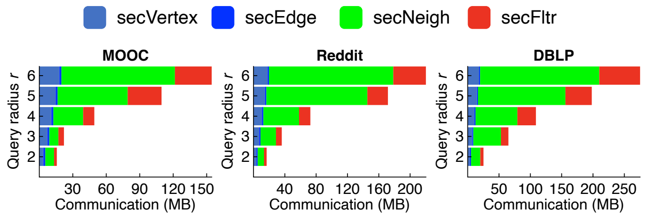


Figure 13: Component-wise breakdown of communication cost in Cove.

aggregation component (i.e., secNeigh). This is because secNeigh requires the servers to transform the src and dst of each tuple in the proximity graph into secret-shared one-hot vector form—a process with complexity on the order of $O(N \times M)$. Additionally, we observe that the latency of oblivious flag propagation to proximity vertices (i.e., secVertex) and to adjacent edges (i.e., secEdge) remains relatively stable across datasets. This is because they involve only oblivious message passing, which depends on the number of tuples in the snapshot graph and the number of oblivious message passing rounds, rather than the total number of vertices in the dataset. Therefore, when the window size is fixed, the costs of secVertex and secEdge remain nearly constant.

Server-side communication. We present a detailed breakdown of the server-side communication cost for each component of Cove in Fig. 13. Similar to the search latency analysis, we observe that the majority of communication cost is dominated by the oblivious neighbor aggregation component (i.e., secNeigh). This is primarily because secNeigh requires transforming vertex into secret-shared one-hot vectors, whose size scales linearly with the size of the streaming graph. As a result, the communication cost of secNeigh grows significantly on larger datasets. In contrast, the communication costs of secVertex and secEdge remain relatively stable across datasets of different sizes. These components rely on the concept of oblivious message passing, which only depends on the number of tuples in the snapshot and the number of rounds, rather than the total number of vertices. The cost of secFltr is also relatively small, as it involves lightweight operations and affects only a limited number of edges and vertices during the secure filtering process. Overall, this highlights that communication bottlenecks mainly stem from the data-dependent encoding operations in secNeigh, while other components are more scalable with respect to graph size.