Oryx: Private detection of cycles in federated graphs

Ke Zhong
University of Pennsylvania

Sebastian Angel
University of Pennsylvania

ABSTRACT

This paper proposes *Oryx*, a system for efficiently detecting cycles in federated graphs where parts of the graph are held by different parties and are private. Cycle identification is an important building block in designing fraud detection algorithms that operate on confidential transaction data held by different financial institutions. *Oryx* allows detecting cycles of various length while keeping the topology of the graphs secret, and it does so efficiently. *Oryx* leverages the observation that financial graphs are very sparse, and uses this to achieve computational complexity that scales with the average degree of nodes in the graph rather than the maximum degree. Our implementation of *Oryx* running on a single 32-core AWS machine (for each party) can detect all cycles of up to length 6 in under 5 hours in a financial transaction graph that consists of tens of millions of nodes and edges. While the costs are high, *Oryx*'s protocol parallelizes well and can use additional hardware resources. Furthermore, *Oryx* is, to our knowledge, the first system that can handle this task for large graphs.

1 INTRODUCTION

In our complex international financial ecosystem, fraudulent activities such as money laundering are commonplace, partly due to the decentralized and opaque nature of this ecosystem and the lack of auditing mechanisms. Financial institutions spend a lot of resources in order to detect and mitigate some of these fraudulent activities: in 2022, they collectively spent around $274 billion on financial-crime compliance [5]. A common approach for understanding financial transactions, and determining whether they are anomalous, is to treat account owners as vertices, transactions as edges, and then study certain structural properties of the resulting graph. A particularly helpful and important structural property is that of *cycles* within the graph [22, 24]. The intuition is that money is transferred between different accounts but eventually goes back to an account that belongs to the original sender, which forms a cycle, and is a strong signal of behaviors such as money laundering.

There is a large literature of works [8, 12, 22, 25, 30] that design algorithms and build systems for finding cycles or other graph structural patterns, but they all assume that a single entity holds (or has visibility into) the entire graph. Allowing financial institutions to do away with this requirement of having to reveal their entire transaction graph to a trusted intermediary (as in the status quo) could unlock impactful audits. Our goal is therefore to *privately find cycles over federated graphs*.

The setting of federated graphs closely resembles reality whereby each financial institution only sees the fraction of transactions that are directly involved with its own accounts and cannot see transactions that occur in other banks or institutions. As such, no party has a global view of the entire graph and cannot effectively detect cycles or other patterns besides those that are visible within their own subgraphs.

Computing privately over federated graphs is not a new problem. There are prior works in this space [6, 19–21, 23]. But there is one key difference between the types of computations that these works target, and those that we study in this paper. In particular, these prior works aim to compute an *aggregate statistic* on the graph, such as PageRank [7]. In other words, if one thinks of each vertex as holding some data, the goal of the existing works is to compute some aggregate function over the data held by the vertices. In contrast, our aim is to identify some property or pattern (cycles in our particular case) that exists within the graph's topology. This is a fundamentally different and more expensive type of computation: even in the non-private setting, the number of subgraphs one needs to process—and therefore the computational complexity—grows exponentially with the average number of neighbors that nodes have in the graph. As a result, existing works are ill-equipped to perform computations over the structure of the graph.

To support this challenging domain we propose *Oryx*, a system that detects cycles over federated graphs while hiding the graph's topology (i.e., the edges between different nodes). *Oryx* works in the client-server MPC setting [9, 10] whereby many clients (the banks in our context) have secret inputs (their subgraphs) and rely on a few servers to perform the computation on their behalf. *Oryx* can be instantiated with two or more semi-honest non-colluding servers, though our particular implementation uses a three-server semi-honest protocol that achieves better performance. In financial settings, these servers could be run by delegates from financial institutions as well as government regulators. These servers will learn nothing about the graphs of individual banks besides the number of vertices and edges, and the result of the cycle detection computation (including some information about the number of paths). We make this explicit in our ideal functionality in Section 6.1.

A key observation that *Oryx* leverages to be efficient is the fact that if the graph represents financial transactions, this graph is actually very sparse. We see this experimentally from a financial money laundering dataset released by IBM [3], but can also understand this intuitively: if vertices are people, then a very dense graph would mean that every person is sending money to nearly every other person which does not make sense. In reality, most people have few transactions; a minority of vertices (e.g., companies) have many transactions. The implication is that the average degree of a node is very small compared to the maximum degree. A generic MPC protocol for finding all cycles of a certain length in the graph would therefore scale exponentially with the maximum degree. In contrast, *Oryx* exploits the graph's sparsity to achieve a similar computational complexity to the non-private baseline: exponential in the average degree rather than the maximum degree.

Oryx makes the following technical contributions:

- **Private cycle detection protocol.** *Oryx* introduces a three-party privacy-preserving cycle detection protocol. The output of this protocol are all of the cycles of a given length, along with

all of the vertices that participate in each of those cycles. This information is precisely what prior works in non-federated graphs aim to collect in order to identify fraudulent transactions [22]. The computational costs of *Oryx*'s protocol are quasilinear in the number of subgraphs (which is itself exponential in the average degree across all nodes) and linear in the length of the cycle. Note that even non-private cycle detection algorithms that list all cycles of a certain length have complexity that is linear in the number of subgraphs. We give the full analysis in Appendix B. In *Oryx*, each data owner (e.g., bank) submits secret shares of its subgraph, including the nodes and edges, to these three parties. Then, using these shares the servers compute over the full graph and output the cycles they detect until they reach a pre-set maximum length of exploration (i.e., how many hops to consider). *Oryx*'s protocol combines a three-server oblivious shuffle protocol [6] with a tailored private message passing paradigm for graph pattern matching inspired by prior work [21].
- **Efficient parallelization.** *Oryx* proposes an efficient parallel version of the private cycle detection protocol. This parallelism allows *Oryx* to scale with multiple cores and multiple machines to handle large-scale financial graph data efficiently.

We evaluate *Oryx* with 3 AWS m5.16xlarge servers co-located in the same datacenter. We use an anti-money laundering financial transaction graph dataset from IBM [3] with tens of millions of vertices and edges, and find that *Oryx* can detect all cycles of up to length 6 (which the authors of prior studies have found sufficient for many applications [22]) in around 4.7 hours.

Limitations. While a lot of our techniques significantly reduce computational costs over using generic MPC or prior works, the servers still need to exchange large amounts of data. In financial settings, this may not be an issue since the servers can be co-located, in much the same way that stock trading servers and related infrastructure is in close proximity to each other. Indeed, our evaluation assumes such co-location.

Oryx's protocol also requires upper bounding the maximum degree across all nodes with some value d; d impacts the protocol's computation complexity and the amount of memory used by each server. Depending on the timescale on which one plans to detect cycles (within the last day versus the last month), d needs to be adjusted accordingly. In our evaluation we study values of d between 10 and 300 (meaning at most 300 transactions per account in the chosen time window for a financial dataset), which we admit might not be realistic. *This limitation is not fundamental*: it stems from the fact that even though *Oryx*'s algorithms are parallelizable, our prototype implementation parallelizes across cores rather than across different machines. As a result, we are bound by the amount of memory available in a single machine for each of our servers.

Finally, cycle detection is an instance of a large class of computations called *subgraph pattern matching*. Other computations in this class are also useful, but our current implementation does not support them (we discuss potential extensions in Section 11).

2 SETTING AND PROBLEM STATEMENT

2.1 Problem description

- $G(V, E)$ is a directed graph where V is the list of all nodes and $E \subseteq V \times V$ represents all the edges. An edge e is defined as a tuple of two nodes (v, v') which denotes that there is a directed path from v to v' and we call this is an out-edge for v and an in-edge for v'. We denote that there are N nodes in G and v_i is the i-th node in V.
- There are B parties who hold partial graph data and are denoted as P_i for $i \in [1, B]$. Each of them holds a disjoint set of nodes V_i where $i \in [1, B]$ and $V_1 \cup V_2 \cup \cdots \cup V_B = V$.
- For each node v in V_i, P_i knows all the edges of v and the edge list of P_i is denoted as E_i. $E_1 \cup E_1 \cup \cdots \cup E_B = E$. Note that the edge lists of two different parties may contain the same edges e which connects the nodes in the two parties' disjoint node lists.
- The in(out)-degree of a node v is defined as the number of incoming (outgoing) edges of v. We use d to denote the maximum in-degree and out-degree of all nodes in G.
- A path p of length k is a sequence of $k + 1$ nodes $v_1, \cdots, v_{k+1}$ such that $(v_i, v_{i+1}) \in E$ for $i \in [1, k]$ and $v_1, \cdots, v_{k+1}$ are distinct nodes.
- A cycle C of length k is a special type of path. It is a sequence of $k + 1$ nodes $v_1, \cdots, v_{k+1}$ such that $(v_i, v_{i+1}) \in E$ for $i \in [1, k]$, $v_1, \cdots, v_k$ are distinct nodes, but $v_1 = v_{k+1}$.

Problem definition. Given a static directed graph $G(V, E)$ held by B parties, P_1 to P_B, and a pre-defined parameter K, three non-colluding servers, S_1, S_2, and S_3, wish to detect all the cycles with a maximum length of K in G without leaking any other edge information besides what is revealed in these cycles. Specifically for each detected cycle, all the nodes and edges associated with the cycle will be revealed.

2.2 Threat model and assumptions

Semi-honest adversaries. We model the servers and graph data holders as *honest-but-curious* adversaries: they will follow the prescribed protocol but will try to infer graph information (i.e., the existence of edges between nodes). We also assume these parties will not collude with each other.

Participants instantiation. The data providers are financial institutions each holding their customers' information including accounts and internal transactions. The computing servers can be instantiated by designated banks or other financial institutions as well as government regulators.

Id alignment. We assume these financial institutions agree on the same id for each account and all ids are positive integers. For each account, only the data holder institution knows the detailed account information (the name of the account holder, balance information, value of internal transfers, etc.). Financial institutions with whom the account has transactions also see some basic information of the account required for processing transactions such as the name of account holder, type of account, etc. All other financial institutions only see that the id exists but know nothing about the account.

3 CAN WE USE GENERIC MPC?

Secure multi-party computation (MPC) frameworks [4, 14, 28] allow mutually distrusting parties to compute any arbitrary function that can be expressed as a boolean or arithmetic circuit on secret inputs without revealing anything else beyond the output of the function. A prior study [6] points out that it is challenging to run graph algorithm using generic MPC frameworks. The key challenge is that if one wishes to hide the graph's topology (as is the case in our setting), the circuit cannot directly follow this topology and must instead hide which node or edge is being processed by performing some (potentially noop) action on every node. For example, to find a neighbor of a given node, the circuit needs to iterate through every node in the graph.

To address this limitation of generic MPC frameworks, recent works [6, 19–21] propose protocols for computing graph analytics such as PageRank [7] while hiding the graph's topology. These works represent a huge improvement over generic MPC frameworks, but they are unfortunately not applicable to our setting. There are two key reasons for this. The first is that graph analytics computes some aggregate function over the data held by various nodes, so the protocol only needs to maintain a constant amount of space in which it collects and updates the result. This is not at all the case in pattern matching tasks such as cycle detection, where we are not interested in computing an aggregate value from data held by nodes but instead in some property about the structure of the graph itself. This requires tracking all relevant subgraphs that satisfy the property, the number of which grows exponentially as one explores deeper into the graph.

The second reason is that existing works adapt a node-centric programming paradigm proposed by graph processing frameworks such as Pregel [18], while (non-private) subgraph matching frameworks [25] typically adopt a different but more suitable subgraph-centric programming paradigm. It is challenging to express a subgraph pattern matching task using the current frameworks supported by private graph analytics. To address this, this paper proposes a way to bring subgraph-centric programming ideas to MPC.

4 NON-PRIVATE CYCLE DETECTION

We start by giving a non-private cycle detection protocol to demonstrate the idea of the subgraph-centric programming paradigm [25], which is a major departure from the paradigm adopted by prior private graph analytics works. Here each subgraph represents a path of a specific length. We then discuss the intuition behind converting this non-private method into a privacy-preserving protocol.

Figure 1 gives the pseudocode for non-private cycle detection. The protocol runs in rounds where it finds out cycles with a specific length in the graph. Initially, paths of length one are initialized with all the edges in the graph. Then, in each round, the computation is divided into two phases, *extension* and *filter*.

In the *extension* phase, we iterate through each path found in the previous step. For each path, we find all the outgoing edges of the last node in the path and append the neighbor node of each edge to the existing path (lines 6–10 in Figure 1). Appending the neighbor node results in a new path with one more node.

Then, in the *filter* phase, we examine each newly generated path and find out which path forms a cycle by verifying whether the first and last node are the same. The detected cycles are removed from the list of paths. For each path, we also check whether the newly appended node occurs in the path twice. The repeating nodes mean that there is a cycle with a smaller length inside the path. Since cycles with smaller length have already been detected in the previous round we do not need to include them for the next round of extension. For example, a path of $a \to b \to c \to d \to b$ includes the cycle $b \to c \to d \to b$ which has been previously detected.

```
1:  function NON-PRIV-CYCLE(V, E, K)
2:    paths ← E
3:    for k ∈ [2, K] do
4:      # Phase 1: extension
5:      new_paths ← [ ]
6:      for p in paths do
7:        # Traverse all outgoing edges of the last node.
8:        for (p[-1], neighbor) ∈ E do
9:          np ← p.append(neighbor)
10:         new_paths.append(np)
11:     # Phase 2: filter
12:     paths ← new_paths
13:     cycles ← K ∗ []
14:     for p in paths do
15:       # Remove paths with repeating nodes.
16:       for i ∈ [1, k − 1] do
17:         if p[i] = p[−1] then
18:           paths.remove(p)
19:           continue
20:       # Detect cycles.
21:       if p[0] = p[−1] then
22:         paths.remove(p)
23:         cycles[k].append(p)
24:   return cycles
```

Figure 1: Pseudocode for non-private cycle detection. The inputs are the list of nodes V, the list of edges E, and the maximum length of cycles to detect K. It outputs the detected cycles with length from 2 to K in the graph.

4.1 Adding privacy to the strawman approach

To turn the non-private cycle detection strawman into a private protocol, we need to support the two phases *extension* and *filter* obliviously without leaking the graph topology. To achieve this goal, we first need a way to encode the graph including nodes, edges, and all the paths that are generated during execution such that the computing parties cannot learn the topology of the graph from the encoded data. Then, we need to design a protocol that can operate directly on this encoded data. In this section, we give some design choices in *Oryx* and defer the details to later sections.

Encoding the data. Prior works on private graph analytics [6, 19–21] store the graph (a set of nodes and a set of edges) as secret shares; each computing party receives one share of the graph, and all shares are needed to recover the graph. In *Oryx*, we follow these works and also store graphs as secret shares.

How to compute over secret shares. The goal of *Oryx* is to compute the entire process in Figure 1 in a private way. Specifically, each server inputs its secret shares of the graph (E and V) and the protocol only outputs the detected cycles (i.e., *cycles*, the return value of the pseudocode). All the intermediate results including the generated *paths* are stored as secret shares without being revealed in the clear so no servers ever know the exact values of the paths.

We now discuss how the two phases, *extension* and *filter*, can be conducted over secret shares.

Since all the generated paths in each round are stored as secret shares, and the filtering computation is performed on each path, we can implement the *filter* phase using generic MPC frameworks [4, 14, 28]. The servers use their local shares of one path to run an MPC to first check whether the path contains repeating nodes; the servers then remove all paths with repeating nodes. Over the paths with no repeating nodes, the servers run MPC again by inputting their local shares of the path and only output whether the path forms a cycle. Finally, the servers exchange their local shares of the cycles to reveal the nodes.

The difficult part is how to do the *extension* in an oblivious way without leaking edge information. Recall that our edges and generated paths are stored as secret shares. Thus, to run *extension* on a path, the servers need to fetch the neighboring nodes without knowing who they are. There are two challenges here. The first challenge is efficiency: how to find the neighbors of a node in an efficient way without naively traversing through each node and doing comparisons one by one. The second challenge comes from the potential to leak too much information: how can we avoid leaking the number of newly generated paths associated with each node given that different nodes have different numbers of neighbors.

To address the first challenge, we borrow ideas from existing works [6, 19–21] that use an oblivious sort operation to significantly reduce the amount of comparisons needed to find the neighbors of a node. We defer the details to Section 5. To deal with the second challenge, we pad each node's neighbor lists to the maximum degree with dummy neighbors so that each node has the same number of neighbors. Then, at a later stage, we remove the paths that contain dummy neighbor nodes in an oblivious way, as otherwise the number of paths would grow exponentially with the maximum degree. Removing these paths leaks the number of total paths of a specific length across all nodes in the graph. This is a significant improvement because instead of leaking per-node information, we leak a single aggregate value. We discuss this further in Section 6.1.

5 OBLIVIOUS MESSAGE PASSING

In this section we review the idea introduced in GraphSC [21] of using oblivious sorting as a way to obliviously pass data from one node to its neighbors. This idea has been used in a lot of follow up works [6, 19, 20]. We will use the PageRank protocol as an example.

Strawman message passing. In a PageRank task, each node has its own rank score and the goal is to pass a node's rank score to its neighboring nodes so that all nodes' scores can be updated. The main challenge is how to pass a node's data to its neighboring nodes while maintaining privacy. For simplicity, we assume that all nodes have the same number of neighbors n. The total number of nodes is denoted as $|V|$, and the total number of edges is denoted as $|E| = n|V|$. The naive way of doing this is as follows. First, we loop through all nodes. For each node i, we have an inner loop that goes over every other node j, and we check to see if j is a neighbor of i. If so, we update i's data so that it incorporates the data of j (e.g., we update the rank by applying some aggregate function on the two values). This results in a total of $n|V|^2$ comparisons.

```
function GraphSC-Pass(tuples)
  var ← 0
  for t in tuples do
    if t.isNode then
      var ← AGG(var, t.data)
    else
      t.data ← var; var ← 0
```

Figure 2: Pseudocode for passing data between sorted tuples.

5.1 Message passing in GraphSC

The previous naive approach is very expensive, which is why GraphSC [21] proposed the following improvement.

Representing the graph. GraphSC encodes both nodes and edges in the same format in order to make it hard to differentiate the two. Specifically, both are encoded as a tuple (*src*, *dst*, *data*). When $src = dst$, this tuple indicates a node with id *src*. Otherwise, it indicates an outgoing edge from node *src* to node *dst*. The *data* field is used to store values such as the rank score of each node in PageRank. The tuples are then split as secret shares.

Passing data. There are two rounds of data passing in GraphSC. First, the *data* of each node i is passed to its outgoing edge tuples (i.e., all edge tuples that contain $src = i$) by setting the *data* field of these edge tuples to be the *data* value of node i. Second, for each node j, an aggregate function is applied over the *data* fields of all the edge tuples where $dst = j$ to compute an aggregate value. This aggregate value is then written to the *data* field of node j.

Message passing with sorted tuples. To allow passing *data* from the source nodes to the outgoing edges, the servers first obliviously sort the tuples based on the *src* field in the tuple (*src*, *dst*, *data*). For node and edge tuples with the same *src* value, the sorting ensures that the node tuples always appear *before* the edge tuples. Likewise for the second data pass, we sort the tuples based on the *dst* field and ensure that for tuples with the same *dst* value, the node tuple always appear *after* the edge tuples.

After the first sort, the tuple for node i is *the closest node tuple* that appears before i's out edges (edge tuples with $src = i$). For example, suppose the servers initially have the shares they received from clients in an arbitrary but consistent order. Say the shares represent tuples $[2, 2, 3], [2, 3, 0], [3, 3, 1], [1, 2, 0], [1, 1, 2]$. After sorting, the list is $[1, 1, 2], [1, 2, 0], [2, 2, 3], [2, 3, 0], [3, 3, 1]$, which contains the first node tuple, followed by its edge, followed by the next node tuple, etc. Then the servers do a linear pass over all tuples to move *data* from the source node to the outgoing edges as shown in Figure 2.

The linear pass runs as follows: the servers begin iterating through the tuples from the start of the sorted list and use a global variable

Ideal functionality $\mathcal{F}$ of cycle detection

Parties: P_i for $i \in [1, B]$, S_1, S_2, and S_3.

Public parameters:

- d: maximum degree of every node in graph G.

Inputs:

- V_i, E_i: list of nodes and edges from each P_i.

Desired output:

- C_k: set of cycles of length k in G for $k \in [2, K]$.

Additional output (i.e., leakage):

- $|V_i| + |E_i|$: sum of the number of nodes and edges for each party P_i.
- pn_k: number of paths of length k in G for $k \in [1, K]$.

Figure 3: Ideal functionality of *Oryx*.

var during the iteration. When encountering a node tuple, *var* is written as the *data* field of the tuple. Otherwise, the tuple is an edge tuple and the aggregate function is applied over *var* and *data* of the tuple (for simplicity, we assume the aggregate function does additions over the inputs). The result after applying the aggregate function is written to the *data* field of the edge tuple. In the example above, *var* is first written as 2 when it encounters the first tuple $[1, 1, 2]$. And then *var* is written to the data field of edge tuple $[1, 2, 0]$ and it becomes $[1, 2, 2]$ after the update.

The second pass to send *data* from the edges to the destination nodes runs in a similar way but with a sorted list that arranges all edges before their destination nodes.

The complexity of the Bitonic sorting network [15] used in GraphSC is $O((|V| + |E|) \log^2(|V| + |E|))$, and the linear pass takes $O(|V| + |E|)$. As a result, the total complexity of private PageRank with sorting is $O((|V| + |E|) \log^2(|V| + |E|))$. If we assume the average number of neighbors is n, then $|E| = n|V|$, which results in $O((n+1)|V| \log^2((n+1)|V|))$—better than the strawman approach's running time of $O(n|V|^2)$.

Recent work by Araki et al. [6] proposed using efficient shuffle and sort protocols to further improve the efficiency of GraphSC assuming three non-colluding servers.

6 PRIVACY-PRESERVING CYCLE DETECTION

In this section, we describe our system *Oryx* which supports privacy-preserving cycle detection. We start by stating the desired privacy guarantee of *Oryx*, then give an overview of the end-to-end cycle detection protocol, describe the data format for edges and generated paths in *Oryx*, and talk about the details of each stage in order to achieve our privacy guarantee. *Oryx* consists of various subroutines. Our particular instantiation of these subroutines uses three servers since they were the most efficient protocols that we know of at present. However, if a better instantiation for any of these subroutines becomes available, *Oryx* could use those instead.

6.1 Privacy guarantee of *Oryx*

The privacy guarantee of *Oryx* is given by the functionality $\mathcal{F}$ in Figure 3. The graph is held by B parties, $P_1, P_2, \ldots, P_B$ and we have three computing servers, S_1, S_2, and S_3. $\mathcal{F}$ takes the graph as an input and it outputs the detected cycles up to length K, which is precisely what we want. However, $\mathcal{F}$ also leaks additional information, owing to the fact that *Oryx* is not perfect. Specifically, $\mathcal{F}$ outputs (1) the sum of the number of nodes and edges for each party P_i because in *Oryx* we will not ask parties to pad the number of their tuples with dummy entries (though we could); (2) the total number of paths in the graph of up to length K. This second leakage is the most fundamental and is specific to the way in which *Oryx* computes cycles efficiently and avoids increasing the number of paths exponentially with the maximum degree d.

What does this leakage mean in practice? Leaking pn_1, which is the number of paths of length 1 is equivalent to leaking $|E|$. Leaking $|V_i| + |E_i|$ for all P_i means that an adversary can recover $|V| = \sum_i(|V_i| + |E_i|) - |E|$. Finally, computing pn_{k+1}/pn_k leaks the average outgoing edges of all nodes in the *entire* graph G. We do not have a proof that this leakage will not allow an adversary to learn whether a particular pair of nodes in the graph has an edge or not with much higher probability than its prior, or other information about the structure of any of the parties' subgraphs (aside from trivial graphs). However, based on our survey of state-of-the-art techniques for reconstructing graphs from partial knowledge [13] they require *significantly* more information than what we leak. We thus conclude that there does not exist any known way to recover the topology of the graphs of any of the parties from the information that we leak, and we conjecture that doing so is actually hard since we only leak aggregate information (e.g., total number edges, vertices, and average out degree).

6.2 Overview of *Oryx*

The protocol consists of three stages. The first stage operates as an initialization phase, during which each data holder ($P_{i \in [B]}$) creates secret shares of its graph. Then Stage 2 and Stage 3 run in rounds in which the servers detect cycles of a specific length k. We give the overview of each stage here and defer the details of each stage to later sections.

Stage 1: Graph data holders create secret shares. Each $P_{i \in [B]}$ first formats its local graph data (i.e., the nodes and edges it owns) in the same way (§6.3) and creates secret shares of the formatted tuples. We use an edge tuple to include both the node ids and all the outgoing edges of the node. The secret shares of both edges and generated paths are indistinguishable. Then, P_i sends one secret share to a computing server respectively. The servers each receive secret shares from all $P_{i \in [B]}$, and then use the secret shares to compute cycles.

Stage 2: Computing servers run oblivious path extension. In each round of detecting cycles of length k, each server holds the secret shares of the edges and the paths of length $k - 1$. The goal for the oblivious path extension protocol is to input these secret shares, and output the secret shares of edges and paths of length k. Paths of length k are generated by extending each path p of length

Struct TUPLE
src
id
$vec = \underbrace{[v_1^1, \ldots, v_k^1, v_{k+1}^1], \ldots, [v_1^d, \ldots, v_k^d, v_{k+1}^d]}_{d}$

Figure 4: Data format definition of a tuple in *Oryx*.

$k-1$ using the outgoing neighbor nodes of the last node in p (as shown in lines 6–10 in Figure 1). For example, suppose node 2 has two neighbors 3 and 4. Given an input path $[1, 2]$, the output paths are $[1, 2, 3]$ and $[1, 2, 4]$.

We capture the above functionality with the function $([es_k]_1, [es_k]_2, [es_k]_3) \leftarrow$ *Ob-Extend*$([s_k]_1, [s_k]_2, [s_k]_3)$, where $[s_k]_1, [s_k]_2, [s_k]_3$ are input and $[es_k]_1, [es_k]_2, [es_k]_3$ are output shares for each of the servers S_1, S_2, S_3, respectively. We show how to build this function in Section 6.5.

Stage 3: Computing servers run oblivious filtering. In the oblivious filtering stage, the servers take as input secret shares of edges and generate paths of length k (i.e., the outputs from running *Ob-Extend* in Stage 2). The servers filter out invalid paths (as shown in lines 16–18 in Figure 1), and detect and reveal cycles (as shown in lines 21–23 in Figure 1). Note that only detected cycles are revealed along with the nodes that form each cycle. Each server then formats secret shares of edges and valid paths to be used for cycle detection of length $k+1$ in the next round.

We capture the above functionality with the function $(c_k, [s_{k+1}]_1, [s_{k+1}]_2, [s_{k+1}]_3) \leftarrow$ *Ob-Filter*$([es_k]_1, [es_k]_2, [es_k]_3)$. Here, the revealed cycles with length k are denoted c_k and the secret shares of paths and edges to be used in the next round are $[s_{k+1}]_1, [s_{k+1}]_2, [s_{k+1}]_3$. We show how to build this function in Section 6.6.

6.3 Data format and secret sharing

To ensure that the secret shares of both edges and generated paths are indistinguishable, we format them into the same structure. In *Oryx*, given the length of cycles to detect, k, and the maximum node degree, d, we format the edges or a path as shown in Figure 4. Each tuple begins with a non-negative integer *src* which indicates a path if $src = 0$ or the edges of node *src* otherwise. The tuple also has an *id* field which is a unique number among all tuples; this field is only used as a tie-breaker for the sorting operation which we will detail in later sections. Then it has field *vec*, which consists of d vectors where each vector contains $k+1$ positive integers. Note that the size of the tuples increases with the round of cycle detection (i.e., as k increases).

For a path $[v_1, \ldots, v_k]$, the formatted tuple has d vectors, each with $k+1$ elements. All of the d vectors in *vec* are the same (duplicates of each other). In each vector, the first k elements are the nodes of the path $[v_1, \ldots, v_k]$ and the last element is an empty placeholder 0. As shown in the example in Figure 5 with d set to 2, we represent the path of $[1, 2, 3]$ as $\{src = 0, vec = ([1, 2, 3, 0], [1, 2, 3, 0])\}$. The reason to have the d copies of the path vector is for oblivious extension which we will detail in section 6.5.

To represent edges, we use a tuple to represent all the neighbors from the outgoing edges of a node *src*. Additionally, the neighbor list of each node is padded with dummy zeros to match the maximum degree d. For example, in the graph shown in Figure 5 with $d = 2$, we use $[2, 0]$ as the neighbor list of node 1. Node 1 has a single neighbor, node 2, and we use the dummy id 0 to pad the neighbor list to two elements. We set *src* field in the tuple to u indicating that it represents the neighbor list of node u. Then we set the first k elements of the d vectors in *vec*, $[v_1^i, \ldots, v_k^i]_{i \in [d]}$, to zeros. And we set the last element of the d vectors, v_{k+1}^i for $i = 1$ to d, to the nodes in the padded neighbor list of node u individually. For example, $\{src = 1, vec = ([0, 0, 0, 2], [0, 0, 0, 0])\}$ represents the neighbor nodes of node 1 with $k = 3$.

Sharing method. In *Oryx*, all these tuples are encoded using replicated secret shares. Assume each tuple t is an ℓ-bit string. The original tuple data holder creates three random secret shares, a, b, c. The three secret shares are three ℓ-bit strings that satisfy $t = a \oplus b \oplus c$. The three computing servers each hold two of the three shares. S_1 holds a and b, S_2 holds b and c, and S_3 holds a and c. The shares held by S_i are denoted as $[s]_i$, for $i = 1$ to 3. We will keep using this notation of secret shares held by each server in later sections.

Subroutines. We define the following to format edges and paths and create secret shares of the formatted tuples.

- *Gen-Edges-Share*$(k, u, e) \rightarrow (ts^1, ts^2, ts^3)$. Takes the node u and the padded outgoing neighbor list e, $[v_1, \ldots, v_d]$, of node u. Outputs three secret shares, $[ts]_1, [ts]_2, [ts]_3$, of the formatted tuple for detecting cycles of length k.
- *Gen-Path-Share*$(k, p) \rightarrow (ts^1, ts^2, ts^3)$. Takes an integer k and path p of length $k-1$ and outputs three secret shares, $[ts]_1, [ts]_2, [ts]_3$, of the formatted tuple for detecting cycles of length k.

6.4 Create secret shares of graph

Each data holder $P_{i \in [B]}$ holds its own disjoint node list V_i and creates secret shares of both edge and path tuples for detecting cycles of length $k = 2$. For each node $u \in V_i$, P_i:

(1) Creates an empty list of nodes l. For each u such that $(v, u) \in E$, u is appended to l. The list l is padded to length d with dummy nodes of zeros.
(2) $[ets]_1, [ets]_2, [ets]_3 \leftarrow$ *Gen-Edges-Share*$(k = 2, u, e = l)$.
(3) $[pts]_1, [pts]_2, [pts]_3 \leftarrow$ *Gen-Path-Share*$(k = 2, p = (u, v))$.

Each P_i now has the three secret shares of its edge and path tuples, $([ets^i]_1, [ets^i]_2, [ets^i]_3)$ and $([pts^i]_1, [pts^i]_2, [pts^i]_3)$. P_i sends one of its secret shares, $[ets^i]_j$ and $[pts^i]_j$, to each computing server S_j, for $j \in [1, 3]$. The tuples are now formatted correctly but with *id* fields not populated yet, which are used as the tie-breakers for sorting. We denote these secret shares by S_j from all P_i as $[s\_no\_id_{k=2}]_j$.

The servers populate the *id* fields for these tuples by assigning each tuple the index i of the tuple in the list of all secret shares starting from 1. We assume that the index is an integer of m bits meaning the maximum possible index is $2^m - 1$. Recall that we use the replicated secret shares, a, b, c, and each secret share of a server has two out of the three shares. For secret share a in $[s\_no\_id_{k=2}]_1$ and $[s\_no\_id_{k=2}]_3$, S_1 and S_3 set the *id* field in a, $[id]_a$, to i. And for $[id]$ fields in the other two secret shares b and c, the servers set the corresponding fields to $2^m - 1$ (i.e., an integer of all m bits being ones). Note that $[id]_a \oplus [id]_b \oplus [id]_c = i \oplus 1 \ldots 1 \oplus 1 \ldots 1 = i$.

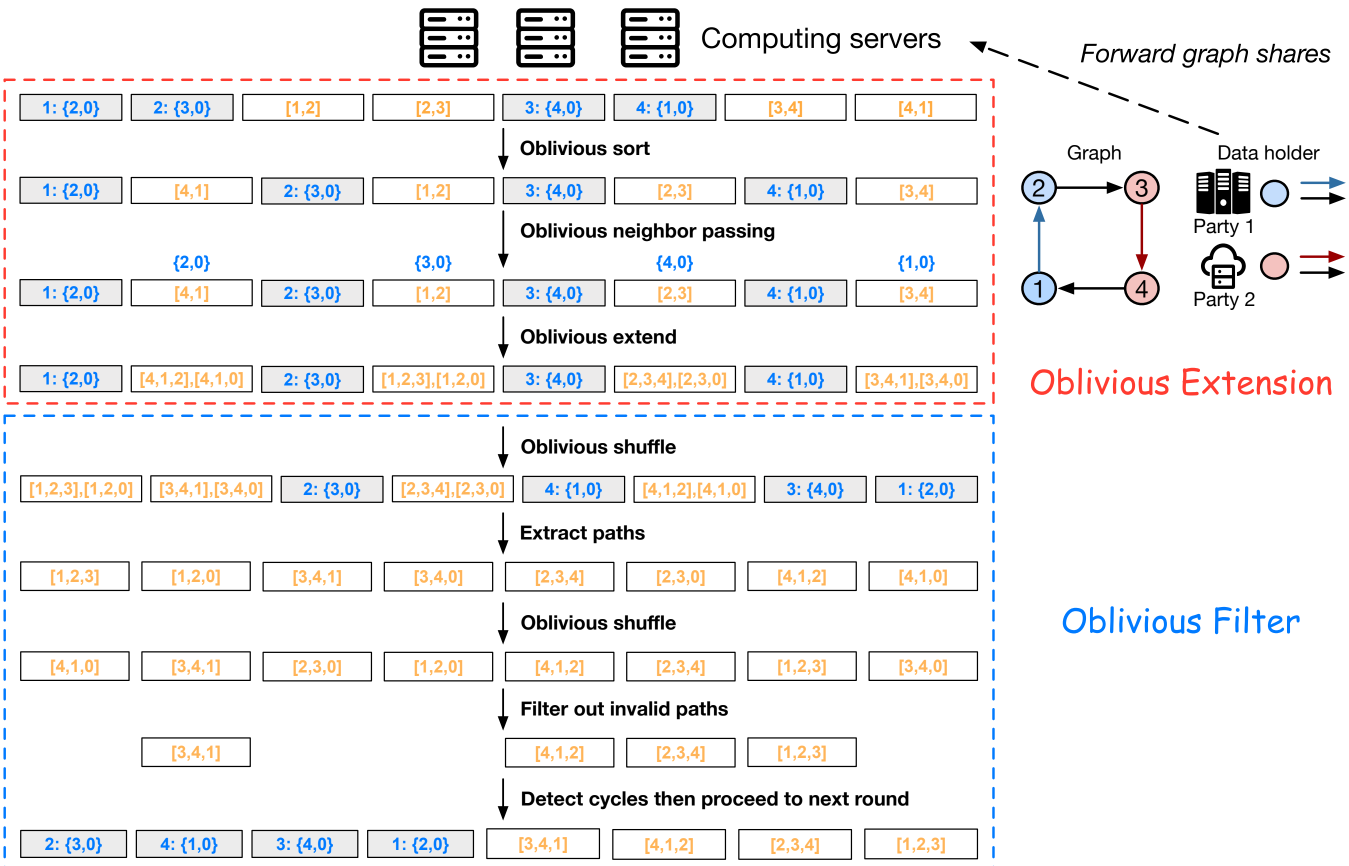


Figure 5: One round of cycle detection. Two data holders (blue and red) generate secret shares of their graphs and send the shares to three computing servers who run *Oryx*. The maximum degree of all nodes is $d = 2$. The grey cells represent edges and the white cells represent paths. At the end of a round, the grey cells and white cells are grouped together, while the internal sequences are random and not sorted.

By manipulating the local secret shares this way, we set the original value of a tuple's *id* to the index i as desired. The secret shares with *id* assigned are denoted as $[s_{k=2}]_1, [s_{k=2}]_2, [s_{k=2}]_3$. We denote the process of the three servers populating the *id*s as $([s]_1, [s]_2, [s]_3) \leftarrow \textit{Assign-Id}([s\_no\_id]_1, [s\_no\_id]_2, [s\_no\_id]_3)$.

6.5 Oblivious extension

This section details how to transform the *extension* phase in Figure 1 into an oblivious operation. The oblivious path extension protocol runs in the following two steps, as illustrated in Figure 5. In the first step, the servers execute an oblivious sort protocol, grouping all path tuples that end with node u alongside the edge tuple of node u. The sorting also ensures that the edge tuple of node u always appears *before* the path tuples that end with node u. In the second step, the servers perform a linear traversal of all the tuples to first pass the node u's neighbor nodes to the path tuples that end with node u. Then, each path tuple that ends with node u can extend the existing path by adding one more edge, using the previously passed neighbor list of node u.

Subroutines. Here we give some notation of the subroutines that will be used in the construction.

- *Ob-Shuffle*$([s]_1, [s]_2, [s]_3) \rightarrow ([rs]_1, [rs]_2, [rs]_3)$. Takes secret shares of a list of tuples from three servers, $([s]_1, [s]_2, [s]_3)$, and outputs the randomized secret shares of the shuffled list of tuples, $([rs]_1, [rs]_2, [rs]_3)$. Note that each server receives only one secret share of the shuffled list.
- *Ob-Sort*$(cmp, [s]_1, [s]_2, [s]_3) \rightarrow ([os]_1, [os]_2, [os]_3)$. Takes a comparator circuit *cmp* for comparing tuples and secret shares of a list of tuples from three servers, $([s]_1, [s]_2, [s]_3)$, and outputs the secret shares of the sorted list of tuples in ascending order, $([os]_1, [os]_2, [os]_3)$, based on *cmp*. Our construction follows the recent work by Araki et al. [6], which first shuffles the tuples using *Ob-Shuffle* and then does the comparison-based sorting over the randomly permuted tuples.

Step 1: Sort edge and path tuples. The pseudocode of the comparator to sort tuples is given in Figure 6. Note that all inputs and intermediate results are secret shares, and only the final comparison result is revealed in plain text. The servers first compute the node value n by XORing *src* and v_k^1 in each tuple (line 2 and 3 in Figure 6).

```
function PRIV-CMP-ON-TUPLES(t_1, t_2)
    n_1 ← t_1.src ⊕ t_1.v_k^1
    n_2 ← t_2.src ⊕ t_2.v_k^1
    if n_1 ≠ n_2 then
        return (n_1 > n_2)
    else if t_1.src ≠ t_2.src then
        return (t_1.src < t_2.src)
    else
        return t_1.id > t_2.id
```

Figure 6: Pseudocode of the comparator function to sort tuples to determine which tuple of t_1 and t_2 is larger. The tuple follows the data format in Figure 4. The input tuples t_1 and t_2 are stored in secret shares. All the computation are conducted over secret shares and only the final comparison boolean result is revealed in clear.

```
1: function PRIV-NEIGHBOR-PASSING(tuples)
2:     neighbors ← [0, . . . , 0]   (d entries)
3:     for t in tuples do
4:         if t.isEdgeTuple then
5:             for i ∈ [1, d] do
6:                 neighbors[i] ← t.v_{k+1}^i
7:         else
8:             for i ∈ [1, d] do
9:                 t.v_{k+1}^i ← neighbors[i]
```

Figure 7: Pseudocode of oblivious neighbor passing and path extension. d is the maximum degree in the graph. The input *tuples* (i.e., all the path and edge tuples) are stored in secret shares and follow the format in Figure 4.

When t is an edge tuple, *src* is the node id and v_k^1 will be 0 (§6.3). And when t is a path tuple, *src* is 0 and v_k^1 is the last node in the path. Thus, n will be either *src* of an edge tuple or v_k^1 in a path. The comparison using n groups the edge tuple of node u and the paths that end with u together. When two tuples have the same n, we further compare *src* of the two tuples. As *src* of path tuple will be 0, a path tuple that ends with node u is always larger than the edge tuple of node u. For paths that end with the same node both *src* fields would be zeros. We use the *id* fields, each of which is unique among all tuples in a round, as the tie-breaker. It ensures there are no equal tuples in the comparison and there is a strict sequence of all tuples after sorting. This approach prevents any additional information from being leaked regarding the number of tuples that end with the same node during the comparison-based sorting.

The servers run $([os_k]_1, [os_k]_2, [os_k]_3) \leftarrow$ *Ob-Sort* $(cmp, [s_k]_1, [s_k]_2, [s_k]_3)$, where k is the cycle length of the current round, using the comparator described in Figure 6. $([os_k]_1, [os_k]_2, [os_k]_3)$ are the secret shares of sorted tuples.

Step 2: Neighbor passing and path extension. The pseudocode of Step 2 (neighbor passing and path extension in Figure 5) is shown in Figure 7. It runs in a similar way to GraphSC (as shown in Figure 2), but tailored for our use case. The servers maintain a variable *neighbors*, which is a vector of d integers. They perform a linear pass over all the tuples. If an edge tuple is encountered, *neighbors* is updated as the current tuple's neighbors (line 4–6 in Figure 7). Otherwise, *neighbors* is written to v_{k+1}^i for $i \in [1, d]$ to add the neighbor to the path.

For example, when the servers encounter the first tuple in Figure 5, representing the neighbor list of node 1, the servers privately evaluate whether the current tuple is an edge tuple. As it is an edge tuple, they then privately assign the values of this tuple's neighbor nodes information to the *neighbors* variable. Now, *neighbors* is privately set to $\{2, 0\}$ (i.e., the neighbors of node 1). The servers then proceed to the next tuple which is the first path tuple of $[4, 1]$ in Figure 5, stored as secret shares of $\{src = 0, vec = ([4, 1, 0], [4, 1, 0])\}$. Again, the servers privately evaluate the tuple's type, and then extend the path by setting the last elements (i.e., two zeros) in the path tuple to the elements in the *neighbors* variable. After extension, the path tuple is written as $\{src = 0, vec = ([4, 1, 2], [4, 1, 0])\}$ by setting the original two zeros as $\{2, 0\}$.

We abstract the above with the following subroutine.

- *Ob-Extend*$([os_k]_1, [os_k]_2, [os_k]_3) \rightarrow ([es_k]_1, [es_k]_2, [es_k]_3)$. It takes the secret shares of sorted tuples, $([os_k]_1, [os_k]_2, [os_k]_3)$, and outputs the secret shares with newly extended path tuples $([es_k]_1, [es_k]_2, [es_k]_3)$.

6.6 Oblivious filtering

In this section, we address how to make the filtering phase of the non-private protocol in Figure 1 oblivious. This process is shown in the oblivious filtering phase of Figure 5. It takes the outputs from the oblivious extension protocol as inputs, which is the secret shares of the path tuples after extension. The goal of oblivious filtering is to filter out invalid extended paths (i.e., the paths that end with invalid node id 0 or with repeating nodes) and then perform cycle detection on the valid paths; revealing the cycles found. It runs in the following three steps: (1) find path tuples and extract d path vectors from each extended path tuple; (2) filter out invalid paths and detect cycles over the valid paths; and (3) format valid path tuples and edges tuples for the next round of detection.

Subroutines. Here we define some subroutines used later.

- *Check-Tuple-Type*$([t]_1, [t]_2, [t]_3) \rightarrow (type)$. Takes the secret shares of a tuple, $[t]_1, [t]_2, [t]_3$, and outputs a boolean *type* which is true if the tuple is an edge tuple or false otherwise.
- *Parse-Path*$([pt]) \rightarrow ([p_1], \dots, [p_d])$. Takes a secret share $[pt]$ of a path tuple (§6.3), and outputs d vectors $[p_1], \dots, [p_d]$. Specifically, the share is:

$$[pt] = \begin{Bmatrix} [src] = [s] \\ [vec] = ([[v_1^1], \dots, [v_{k+1}^1]], \dots, [[v_1^d], \dots, [v_{k+1}^d]]) \end{Bmatrix}$$

  where, $p_i = [[v_1^i], \dots, [v_{k+1}^i]]$ for $i \in [1, d]$. Note that this is a computation done by each server locally.
- *Private-Filter-Path*$([p]_1, [p]_2, [p]_3) \rightarrow valid$. Takes the secret shares of a path vector of length k (i.e., a vector of $k+1$ nodes), $([p]_1, [p]_2, [p]_3)$, and outputs a boolean variable *valid* which indicates whether this path is a valid path or not. The details are shown in Figure 8.

```
function PRIVATE-FILTER-PATH(p)
    if p[−1] == 0 then
        return False
    # Skip comparing the first node with the last one.
    for i ∈ [1, len(p) − 1] do
        if p[i] == p[−1] then
            return False
    return True
```

Figure 8: Pseudocode of *Private-Filter-Path* subroutine used in oblivious filtering protocol (§6.6). The input are secret shares of a path p. And all the computation are conducted over secret shares and only leaks the final boolean variable to indicate whether this path p is a valid one.

- *Private-Cycle-Detection*$([p]_1, [p]_2, [p]_3) \rightarrow$ *detected*. Takes the secret shares of a path vector of length k (i.e., a vector of $k+1$ nodes), $([p]_1, [p]_2, [p]_3)$, and outputs a boolean variable *detected*, which indicates whether it forms a cycle by privately evaluating whether the first and the last nodes in the path are the same.

Step 1: Extract paths. The servers first perform an oblivious shuffle over the outputs from the oblivious extension phase to obfuscate the sequence of originally sorted tuples by running $([st_k]_1, [st_k]_2, [st_k]_3) \leftarrow$ *Ob-Shuffle*$([es_k]_1, [es_k]_2, [es_k]_3)$. For each tuple t in the shuffled tuples st_k, the servers run $(type) \leftarrow$ *Check-Tuple-Type*$([t]_1, [t]_2, [t]_3)$ to check the type of the tuple t. For all edge tuples, the servers store their local shares, denoted as $[edges_k]_i$ for $i \in [1, 3]$. For each path tuple pt, each S_i parses its local share $[pt]_i$ into local secret shares of d paths with $([p_1]_i, \dots, [p_d]_i) \leftarrow$ *Parse-Path*$([pt]_i)$ for $i \in [1, 3]$. For example, given a path tuple $pt = \{src = 0, vec = ([4, 1, 2], [4, 1, 0])\}$, with $d = 2$, each server parses its local shares of pt and obtains the local shares of $[4, 1, 2]$ and $[4, 1, 0]$ respectively.

Step 2: Filter out invalid paths and detect cycles. Before filtering out invalid paths, the servers shuffle the tuples again. The servers then use the *Private-Filter-Path* subroutine as defined in Figure 8 to privately check whether each path tuple is valid or not. A valid tuple should consist of all non-zero nodes and should not contain repeating nodes. All invalid paths are removed. For each valid path pt, the servers run *Private-Cycle-Detection*$([pt]_1, [pt]_2, [pt]_3)$ to check whether the current path forms a cycle. When a cycle is detected, the servers reveal their local shares to each other to reconstruct and reveal the cycle with all nodes. For all valid and non-cycle path tuples, each server retains the local share. These tuples are denoted as $paths_k$, and the local shares are denoted as $[paths_k]_i$ for $i \in [1, 3]$.

Step 3: Format tuples for next round. As mentioned in Section 6.3, the tuples of edges and paths have different sizes across cycle detection rounds. Thus, the servers need to set their local shares of $edges_k$ and $paths_k$ of length k to the proper format for use in the round of length $k+1$.

For each local share of an edge tuple $[et_k]_i = \{[src] = [s]_i, [vec] = ([[v_1^1]_i, \dots, [v_{k+1}^1]_i], \dots, [[v_1^d]_i, \dots, [v_{k+1}^d]_i])\}$ of S_i for $i \in [1, 3]$, S_i appends a zero before each $[v_1^d]_i$. So the local share is updated:

$$\left\{ \begin{array}{l} [src] = [s]_i, \\ [vec] = ([0, [v_1^1]_i, \dots, [v_{k+1}^1]_i], \dots, [0, [v_1^d]_i, \dots, [v_{k+1}^d]_i]) \end{array} \right\}$$

Appending a zero of each of the local shares is equivalent to appending a zero element to the original edge tuple since $0 \oplus 0 \oplus 0 = 0$. Now, the edge tuples have the format for detecting cycles of length $k+1$. Note that servers still only see their local shares so the original value of the edge tuple is still kept secret. This process of formatting the edge tuple is denoted as *Extend-Edge-Share*$(et_k) \rightarrow et_{k+1}$.

For each path vector $p_k = [v_1, \dots, v_{k+1}]$ in $paths_k$, each servers uses the local shares of a path vector to create local shares of a path tuple to detect cycles of length $k+1$. As an example, for a path vector $[1, 2, 3]$, each server uses its local share of the vector to compute a share of the formatted tuple $\{src = 0, vec = ([1, 2, 3, 0], [1, 2, 3, 0])\}$ with $d = 2$. The process is performed locally so the original values of the vector remain secret. They compute as follows. S_i with $[p_k]_i = [[v_1]_i, \dots, [v_{k+1}]_i]$ creates a local path tuple share $[pt_{k+1}]_i = \{[src] = 0, [[v_1^j] = [v_1]_i, \dots, [v_{k+1}^j] = [v_{k+1}]_i, [v_{k+2}^j] = 0]_{j \in [1,d]}\}$. Note that S_i sets its local share for the *src* field of the tuple as zero, and this is equivalent to setting the original value of *src* as zero as well. Similarly, the tuple's last elements in each vector v_{k+2}^j for $j \in [1, d]$ are also set to zeros. The process of formatting a path is denoted as *Format-Path-From-Share*$(p_k) \rightarrow (pt_{k+1})$.

After this step the servers can tell which tuples indicate edges or paths and the secret shares are not indistinguishable. However, the next round begins with an oblivious shuffle (the first step in the sorting), so both original sequences and values of secret shares are obfuscated and randomized again. After formatting the tuples, servers assign the *id* fields of the tuples by running the *Assign-Id* subroutine. These tuples with *id* assigned are denoted as $[s_{k+1}]_i$ for $i \in [1, 3]$ as the local secret shares held by S_i.

6.7 Security

We formalize the security of *Oryx* with the following theorem and give the proof in Appendix A.

Theorem 1. *Oryx* securely implements the ideal functionality in Figure 3 under the threat model of Section (§2.2).

6.8 Complexity analysis

Here we summarize the computation complexity of *Oryx*. The full analysis is available in Appendix B.

Let v be the total number of vertices, n be the average number of neighbors, and d be the maximum degree. In a round of cycle detection of length k, the number of total subgraphs to process is $T = v \cdot n^{k-1}$. *Oryx*'s computational complexity for that round is $O(kT(d + \log(T)))$. Note that even non-private protocols for cycle detection will be linear in T, as they have to at least iterate through each subgraph and do the path extension.

7 PARALLEL CYCLE DETECTION

Parallelism is essential for the efficiency of *Oryx*, which currently runs sequentially over each tuple. In this section, we discuss how to transform our protocol to a parallel version.

Except for the oblivious shuffle, sort, and neighbor passing subroutines, all other operations as shown in Figure 5 are performed over each tuple with no dependencies on other tuples, thus making it embarrassingly parallel (they can run on independent MPC instances). Now we discuss how to support parallelism of the remaining subroutines.

7.1 Parallel oblivious shuffle

We instantiate the oblivious shuffle with the three-server shuffle protocol of Araki et al. [6]. The main computation involves (1) computing XOR over two messages, where each message consists of multiple tuples; and (2) permuting the list of tuples using a seed agreed by two out of the three parties. XOR operations on multiple tuples can be computed in parallel. The permutation is a lightweight computation involving the relocation of tuples from their original positions to the permuted index. As such, it does not require parallelization.

7.2 Parallel oblivious sort

In the sorting operation, all tuples are initially shuffled, followed by a comparison-based sort over the shuffled tuples, which is quicksort in *Oryx*. In each round of quicksort, the data is split into multiple partitions. In each partition, a pivot is selected, then we perform comparisons between the pivot and each tuple. Therefore, once a pivot is chosen for each partition, the comparisons between each tuple and its respective pivot can be performed in parallel.

7.3 Parallel oblivious neighbor passing

For parallelism, all the tuples are split evenly into M partitions, with the intention of processing these M partitions of tuples simultaneously. However, the challenge in creating a parallel version of oblivious neighbor passing (Figure 7) is that the value of the *neighbors* variable, when the loop encounters a tuple, depends on the types and values of the previous tuples. As a result, for each parallel task, we require an additional step to privately compute the values of *neighbors* (still in secret share format), which are intended to be passed to the first tuple in its respective partition. We refer to these values as the *start_neighbors* of each partition.

As *neighbors* will only be updated when the servers encounter an edge tuple, finding the *start_neighbors* of each parallel task is equivalent to find the *nearest edge tuple* before the first tuple in this partition. One intuitive approach would be for each processor to iterate over all previous tuples from the end to the beginning to find out the *start_neighbors*. However, since the protocol needs to be oblivious, the protocol has to finish iterating through all tuples even though an edge tuple is found before reaching the beginning. Given t tuples in each partition, each task $m \in [1, M]$ needs to iterate through $t \cdot (m-1)$ tuples. This means that for the last tuple the servers need to go through all the tuples in the current round making the parallelism useless.

Instead, we find *start_neighbors* as in Figure 9. In the first round of computation, each task tries to find the *nearest_neighbors* within its partitioned data by iterating from the beginning to end. Since the data is evenly partitioned, there is a possibility that one partition might not contain an edge tuple. Thus, each task also computes a boolean value *encountered_edge* to indicate whether there is an

```
function PRIV-FIND-START-NEIGHBORS(tuples[M])
    nearest_neighbors ← [[0, ..., 0], ..., [0, ..., 0]]   (M lists, each of d zeros)
    start_neighbors ← nearest_neighbors
    encountered_edge ← [false, ..., false]   (M entries)
    for m ∈ [1, M] do
        # Round 1
        for t ∈ tuples[m] do
            if t.isEdgeTuple then
                nearest_neighbors[m] ← t.neighbors
                encountered_edge[m] ← true
        # Round 2
        Task m ≥ 2 waits for the tasks 1 to m − 1 to finish.
        if_update_neighbors ← encountered_edge[m − 1]
        start_neighbors[m] ← nearest_neighbors[m − 1]
        for i in [m − 2, 1] do
            if_update_neighbors ← (!if_update_neighbors & encountered_edge[i])
            if if_update_neighbors == True then
                start_neighbors[m] ← nearest_neighbors[i]
    return start_neighbors
```

Figure 9: Pseudocode of obliviously finding *starting neighbors* of in total M parallel tasks. Each task m processes its own partitioned data *tuples*[m]. Both inputs and outputs are stored in secret share format and nothing else in leaked during the computation.

edge tuple within this partition. In the second round, each task m only iterates through the *nearest_neighbors* found by previous tasks 1 to $m-1$. This is lightweight compared to the naive solution which requires iterating through $t \cdot (m-1)$ elements. Once *start_neighbors* are determined for each task, each task continues as the original protocol while initializing *neighbors* with the found *start_neighbors* instead of all zeros (line 2 in Figure 7).

8 IMPLEMENTATION

Oryx consists of around 3K lines of C++. For the oblivious shuffle protocol, we implement the three-server shuffle protocol proposed by Araki et al. [6]; for the oblivious sort protocol, we implement a prior protocol [6, 11, 16] that first shuffles and then does comparison sort over the shuffled tuples. We implement the parallel version of quicksort as the sorting algorithm. We use emp-toolkit's sh2pc [28] library as the MPC.

Run MPC with two servers. In *Oryx*, we use the three-server shuffle protocol, but for other MPC tasks, we only use two servers for computation. The detailed process of running the MPC tasks using two servers is as follows. As each server holds two out of three secret shares, a, b, c such that $a \oplus b \oplus c = m$ where m is the original data. One server S_1 could compute XOR over its local share as s_1 (e.g., $s_1 = a \oplus b$) and another server S_2 can use one of its secret shares as s_2 (e.g., $s_2 = c$) such that $s_1 \oplus s_2 = m$. Then the two server input s_1 and s_2 respectively to run the computations in 2PC.

Reassign secret shares. In some MPC subroutines, such as the path extension, the final outputs are also secret shares, but they are held by only two servers since only two servers are involved in the MPC tasks. When the third server is required for oblivious shuffle, the two servers holding the two output secret shares o_1 and o_2 can reconstruct the secret shares back to replicated secret shares as follows. S_1 randomly generates a' and b' such that $a' \oplus b' = o_1$ and sends b' to S_2 and a' to S_3. S_2 use $c' = o_2$ and sends c' to S_3. Now each server holds two out of the three shares a', b', c'.

Optimizations of oblivious sort. We followed the oblivious sorting used by Araki et al. [6]. We optimized it by selecting the pivot in quicksort using median values obtained from randomly sampled elements in each partition. We also directly sort all tuples, avoiding further recursion in quicksort, when the size of elements in a partition falls below a threshold of t tuples.

9 EVALUATION

In this section, we answer the following questions:

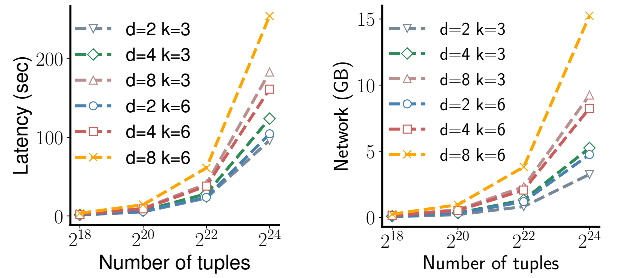
(1) What are the costs of each subroutine in *Oryx*?
(2) What are the end-to-end costs (including both computation and network communication) of *Oryx*'s protocol?

Evaluation setting. We run all of our experiments on AWS m5.16xlarge instances (32-core Intel Xeon and 256 GB RAM) with Ubuntu 20.04. All instances are launched in US East (Ohio) and we allocate one instance for each computing party. Note that by leveraging the parallelism of *Oryx*'s protocol, it is possible to scale out *Oryx* further by employing multiple servers for each computing party, but we have not yet implemented this.

Parameters. We represent node ids using 23-bit integers, allowing for a maximum of 2^{23} nodes in the graph. We use 25-bit integers to represent tuple ids, supporting a maximum of 2^{25} tuples for processing. For quicksort, we set the number of randomly sampled tuples to find the median to 7, and we set the threshold to directly sort all tuples to 10.

Datasets. We use two datasets. The first is a synthetic graph with 1,000 vertices and 3,500 edges; 5 nodes have 300 neighbors. This serves as a microbenchmark where a few nodes in the graph have a much higher degree than the others and a higher max degree d (100 to 300) is required. The second dataset was published by IBM [3] and represents financial transactions, including some money laundering activities. We preprocess the second dataset by limiting the maximum degree d to 10. This ensures that the memory of our servers is enough to complete the experiment (to support larger d we would need either servers with more memory or an implementation that uses many servers). The resulting graph comprises 7,339,522 vertices and 9,328,103 edges. The numbers of cycles from length 2 to 6 are 499,141; 152,170; 60,868; 25,717; and 11,071 respectively.

Graph partitioning. There are four graph data holders, each possessing one-fourth of the total nodes for both datasets, as described in Section 2.1. Each data holder creates secret shares of their local graph, following the procedure outlined in Stage 1 of the protocol



(a) Shuffle latency (sec). **(b) Shuffle communication (GB).**

Figure 10: Microbenchmark for shuffle.

Number of tuples	2^{18}	2^{20}	2^{22}	2^{24}
Sorting	40.587	76.850	224.032	853.138
Neighbor passing				
$d = 2$	1.640	3.330	11.388	43.692
$d = 4$	2.150	5.261	19.288	75.582
$d = 8$	3.498	9.564	35.632	139.397
Type checking	0.747	1.153	3.017	8.506
Filtering				
$k = 3$	0.976	1.615	5.051	19.045
$k = 4$	0.974	1.908	6.515	24.497
$k = 5$	0.999	2.518	7.994	29.882

Figure 11: Latency measured in seconds of the sort, neighbor passing, tuple type checking, and filtering subroutines.

(Section 6.2), and sends these secret shares to the three computing servers. Note that the number of graph holders does not impact the performance of the protocol, only the size of the graph does.

9.1 Costs of each subrountine

We measure the costs of the servers running the five subroutines as depicted in Figure 5: (1) shuffling tuples; (2) sorting over shuffled tuples; (3) neighbor passing and path extension; (4) checking tuples' types over shuffled tuples; and (5) filtering invalid paths and cycle detection. For all subroutines, we measure the latency of servers from beginning to end. We also use tcpdump [2] to measure the total network traffic. We report the mean of values over five runs.

Costs of shuffle. The total network traffic varies among the three servers, with S_2 experiencing the highest network traffic load. The network traffic of S_1 and S_3 accounts for 1/4 and 3/4 of that of S_2, respectively. Here we only report the communication costs of S_2; the latency and communication costs are shown in Figure 10a and Figure 10b with varied number of tuples T, maximum degree d, and the length of cycles k. The results show that both metrics increase sublinearly with d and k, and linearly with T.

Costs of sort. Since the sorting algorithm is data-oblivious, meaning that it operates independently of the distribution of the tuples' types, the sorting runtime should remain constant when given the same number of tuples; the costs are not related to k (§B). We run

Number of tuples	2^{18}	2^{20}	2^{22}	2^{24}
Sorting	2.956	11.190	40.983	127.050
Neighbor passing				
$d = 2$	1.233	3.306	11.144	36.909
$d = 4$	2.220	5.399	21.606	52.206
$d = 8$	3.869	15.826	41.108	119.770
Type checking	0.203	0.819	2.569	5.804
Filtering				
$k = 3$	0.604	2.440	8.461	22.596
$k = 4$	0.849	3.166	11.844	28.579
$k = 5$	1.086	4.373	16.458	47.999

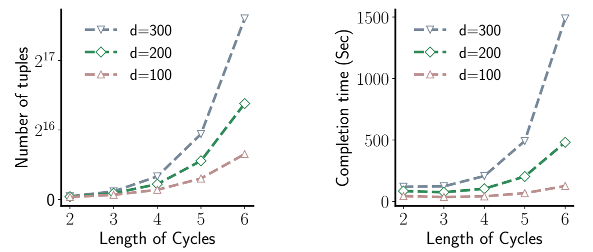
Figure 12: Network traffic measured in GB of the sort, neighbor passing, tuple type checking, and filtering subroutines.

the evaluation over a set of tuples, half of which are path tuples and the other half are edge tuples. The maximum degree is $d = 10$ and the cycle length is $k = 4$. The latency and communication costs are shown in Figure 11 and Figure 12 respectively. The complexity is, as expected, quasilinear in T.

Costs of neighbor passing and path extension. As only the maximum degree d and the number of tuples T impacts the runtime of this subrountine, we fix the length of cycles k to 4 and run the evaluation over a set of tuples, half of which are path tuples and the other half are edge tuples. We display the latency and communication costs in Figure 11 and Figure 12 for different values of d and T. Both metrics grow linearly with both d and T.

Costs of tuple type checking. Checking the types of each tuple only uses the s field, and hence the runtime is only related to the number of tuples. We therefore run the evaluation over sorted tuples, half of which are path tuples and the other half are edge tuples. As with prior experiments, we keep $d = 10$ and $k = 4$. The latency and communication costs are shown in Figure 11 and Figure 12 respectively—both metrics increase linearly with T.

Costs of filtering and cycle detection. In this subrountine, the maximum degree d affects the number of tuples T. However, for simplicity, we omit d and directly experiment with different values of T. We fix the percentage of cycles in all path tuples to 0.5%, as this does not impact the runtime. We experiment with varying percentages of valid paths in all path tuples, specifically 5%, 10%, and 15%. This choice aligns with our end-to-end evaluation, where the majority of path tuples are invalid. The latency remains nearly the same, while the communication costs experience a slight increase with different percentages of valid paths. This outcome is expected since checking cycles over each valid path involves only one comparison, and the percentage of valid paths is relatively small. Consequently, this component is relatively minor in the overall computation. We report the latency and network communication for a graph with a percentage of valid paths set to 15% in Figure 11 and Figure 12 respectively. We vary the number of tuples T and the cycle length k. Both metrics grow linearly with k and T.



(a) Number of processed tuples **(b) Completion time (Seconds)**

Figure 13: End-to-end results of *Oryx* on small dataset.

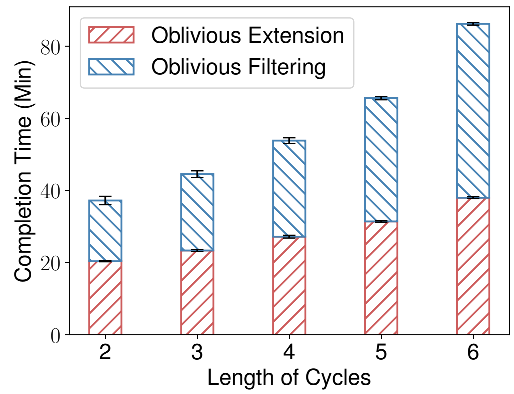


Figure 14: End-to-end completion time of *Oryx* on the large IBM dataset with varying detection cycle length.

9.2 End-to-end evaluation

We use three servers and the two datasets described in Section 9 (small synthetic dataset and large dataset from IBM). The evaluation concludes when the servers detect cycles up to length 6. We chose length 6 because it strikes a balance between keeping the number of subgraphs T manageable (which scales exponentially with k and is an issue even in the non-private protocol, not just in *Oryx*), and being useful in practice, as identified by prior work that detected fraudulent activities in Alibaba with cycle length 6 [22]. We report metrics as the mean over three runs.

9.2.1 Evaluation on small synthetic dataset. We start by studying the number of tuples that *Oryx* must process as a function of the maximum degree d and the length of the cycle to be detected. We can immediately observe that cycle detection, even with *Oryx*'s optimizations, is a high-complexity operation: as shown in Figure 13a, the number of tuples T in each round of cycle detection grows exponentially with the average number of neighbors (n). Note that without *Oryx*'s filtering optimization, the number of tuples would be exponential in the maximum degree d rather than in n, and hence much worse than what is depicted in Figure 13a since $n \ll d$.

We also study the end-to-end completion time and show the results in Figure 13b. If we focus on the completion time for a given round k but under a different maximum degree d, we find that the time grows linearly with dT. If we then look at the completion time under the same maximum degree d but with different cycle length (i.e., different rounds) k, the completion time is also linear with kT. These results are consistent with our complexity analysis in Section B, which indicates that the total end-to-end completion time is linear with respect to the total number of processed tuples T, the cycle length k, and the maximum degree d.

9.2.2 Evaluation on IBM's financial dataset.

Local storage. Each server locally stores two out of three secret shares, with each share being 2.2 GB of data. This requires a total of 4.4 GB of local storage for each server.

Peak memory usage. During the entire run, the peak memory usage is around 230 GB memory.

Total network traffic. As we run the MPC program using only two servers, these two servers handle the majority of data exchange during execution. Due to the substantial network traffic, it is not feasible to capture packets using tcpdump. Instead, we rely on the native cloudwatch [1] metrics for inbound and outbound network traffic provided by AWS. These metrics provide an upper bound estimate of the total network traffic for each end-to-end run as the total network traffic encompasses other parts of traffic on each instance, in addition to what is incurred by the end-to-end run. On average, each of the two servers needs to exchange approximately 20.7 TB of data for a complete end-to-end run. This significant network traffic characterizes *Oryx* as network-bound, necessitating high network bandwidth for deployment.

Completion time. The time breakdown for each round of detecting cycles of length k is in Figure 14. In rounds with cycle length 2 to 6, the number of processed tuples are 16,666,380; 18,100,272; 19,995,417; 22,267,002; and 25,190,191. As both the number of processed tuples and the length of cycles for detection increase, the completion time also grows. In most rounds, the process can be completed within half to one hour, while the most time-consuming round, used to detect cycles of length 6, can be finalized within 1.5 hours. These costs are practical for applications such as money laundering as they typically run in the background.

10 RELATED WORK

Private graph algorithms. GraphSC [21] first studied private graph analytics, and this was improved by Araki et al. [6] who introduce an efficient 3-server shuffle protocol. Other works [19, 20] use four servers but they leak differentially private information about the degrees of nodes. In all cases, these works do not handle cycle detection or other tasks that analyze the graph's structure.

Vorstermans's masters thesis [26] proposed the only other work in the literature for privately detecting cycles over federated graphs. The techniques used by Vostermans are very different to *Oryx*: they rely on MPC over adjacency matrix multiplications. A consequence of Vostermans use of adjacency matrices is that the computational complexity of their proposal grows exponentially with the total number of vertices, which is essentially the maximum possible node degree in the graph. In contrast, *Oryx*'s computation scales with the average node degree, which in the sparse graphs that we target, is significantly smaller.

Outsourcing graph pattern queries. Prilo [29] and OblivGM [27] enable a single data owner to outsource its graph to some untrusted service and then perform private subgraph pattern queries on this graph. The target domain for these works is different from *Oryx*: *Oryx* targets federated graphs that belong to different data owners. In terms of techniques, Prilo uses trusted hardware (very different from *Oryx*), while OblivGM uses non-colluding servers (similar to *Oryx*) but provides a weaker privacy guarantee [31].

11 DISCUSSION

Support node and edge attributes. In some graphs nodes or edges might have attributes and it might be desirable to detect cycles only within vertices or edges that have a specific attribute. To achieve this, we can include the attributes of vertices and edges in the tuples. Then, in the filtering phase, *Oryx* can inspect the attributes when checking whether a path forms a desired cycle.

Support more subgraph patterns. *Oryx* can be extended to support more subgraph pattern matching queries besides cycles. This extension requires a change in the logic of the extension phase to determine the source vertices for extension instead of always using the last vertex in the path. *Oryx*'s neighbor passing and path extension can still be reused as a building block for matching new subgraph patterns.

Source code

Our code is available at:
https://github.com/eniac/oryx.

Acknowledgments

We thank Brett Hemenway Falk, Betül Durak, and Matan Shtepel for helpful conversations that improved this work. This work was funded in part by NSF grants CNS-2045861, CNS-2107147, and a gift from the Arcological Swiss Association.

REFERENCES

[1] Application and Infrastructure Monitoring – Amazon CloudWatch. https://aws.amazon.com/cloudwatch/.
[2] TCPDUMP & LIBPCAP. https://www.tcpdump.org/.
[3] AML-Data. https://github.com/IBM/AML-Data, 2021.
[4] SCALE and MAMBA. https://github.com/KULeuven-COSIC/SCALE-MAMBA, 2021.
[5] Cost of Compliance is Rising: How to Cut Down Your AML Costs. https://www.tookitaki.com/compliance-hub/reducing-aml-compliance-costs-tookitaki-solutions, 2023.
[6] T. Araki, J. Furukawa, K. Ohara, B. Pinkas, H. Rosemarin, and H. Tsuchida. Secure graph analysis at scale. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2021.
[7] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 1998.
[8] J. Cheng, J. X. Yu, B. Ding, P. S. Yu, and H. Wang. Fast graph pattern matching. In *2008 IEEE 24th International Conference on Data Engineering*, 2008.
[9] R. Cramer, I. Damgård, and Y. Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In *Proceedings of the Theory of Cryptography Conference (TCC)*, 2005.
[10] I. Damgård and Y. Ishai. Constant-round multiparty computation using a black-box pseudorandom generator. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2005.
[11] K. Hamada, R. Kikuchi, D. Ikarashi, K. Chida, and K. Takahashi. Practically efficient multi-party sorting protocols from comparison sort algorithms. In *Information Security and Cryptology*, 2012.
[12] A. P. Iyer, Z. Liu, X. Jin, S. Venkataraman, V. Braverman, and I. Stoica. ASAP: Fast, approximate graph pattern mining at scale. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[13] S. Kannan, C. Mathieu, and H. Zhou. Near-linear query complexity for graph inference. In *Automata, Languages, and Programming*, 2015.
[14] M. Keller. MP-SPDZ: A versatile framework for multi-party computation. Cryptology ePrint Archive, Report 2020/521, 2020.
[15] D. E. Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. 1998.
[16] S. Laur, J. Willemson, and B. Zhang. Round-efficient oblivious database manipulation. In *Information Security*, 2011.
[17] Y. Lindell. How to simulate it - a tutorial on the simulation proof technique. Cryptology ePrint Archive, Report 2016/046, 2016. https://ia.cr/2016/046.
[18] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *ACM SIGMOD*, 2010.
[19] S. Mazloom and S. D. Gordon. Secure computation with differentially private access patterns. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2018.
[20] S. Mazloom, P. H. Le, S. Ranellucci, and S. D. Gordon. Secure parallel computation on national scale volumes of data. In *Proceedings of the USENIX Security Symposium*, 2020.
[21] K. Nayak, X. S. Wang, S. Ioannidis, U. Weinsberg, N. Taft, and E. Shi. Graphsc: Parallel secure computation made easy. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2015.
[22] X. Qiu, W. Cen, Z. Qian, Y. Peng, Y. Zhang, X. Lin, and J. Zhou. Real-time constrained cycle detection in large dynamic graphs. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2018.
[23] E. Roth, K. Newatia, Y. Ma, K. Zhong, S. Angel, and A. Haeberlen. Mycelium: Large-scale distributed graph queries with differential privacy. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2021.
[24] T. Suzumura, Y. Zhou, N. Barcardo, G. Ye, K. Houck, R. Kawahara, A. Anwar, L. L. Stavarache, D. Klyashtorny, H. Ludwig, and K. Bhaskaran. Towards federated graph learning for collaborative financial crimes detection. http://arxiv.org/abs/1909.12946, 2019.
[25] C. H. C. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga. Arabesque: A system for distributed graph mining. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2015.
[26] M. H. L. Vorstermans. Secure graph algorithms and oblivious data structures for multiparty computation, 2023.
[27] S. Wang, Y. Zheng, X. Jia, H. Huang, and C. Wang. Oblivgm: Oblivious attributed subgraph matching as a cloud service. *IEEE Transactions on Information Forensics and Security*, 2022.
[28] X. Wang, A. J. Malozemoff, and J. Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.
[29] L. Xu, B. Choi, Y. Peng, J. Xu, and S. S Bhowmick. A framework for privacy preserving localized graph pattern query processing. In *ACM SIGMOD*, 2023.
[30] Z. Zhu, K. Wu, and Z. Liu. Arya: Arbitrary graph pattern mining with decomposition-based sampling. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2023.
[31] L. Zou, L. Chen, and M. T. Özsu. k-automorphism: a general framework for privacy preserving network publication. *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2009.

A SECURITY PROOF

We first give the formal description of our protocol and then do a simulation proof [17] to prove that our construction leaks no more information than the outputs from the ideal functionality (§6.1).

A.1 *Oryx*'s protocol of cycle detection

Oryx's protocol of cycle detection

Step 0 (Create secret shares of graph):
Each partial graph holder P_j, where $j \in [1, B]$ holds its own disjoint node list V_j. P_i create secret shares of both edge and path tuples. For each node $u \in V_i$, P_i computes as follows.

- P_j creates an empty list of nodes l. For each u such that $(v, u) \in E$, u is appended to l. And the list l is padded to length d with dummy nodes of zeros.
- $[ets^j]_1, [ets^j]_2, [ets^j]_3 \leftarrow$ *Gen-Edges-Share*$(k = 2, u, e = l)$.
- $[pts^j]_1, [pts^j]_2, [pts^j]_3 \leftarrow$ *Gen-Path-Share*$(k = 2, p = (u, v))$.
- $[ets^j]_i$ and $[pts^j]_i$ are denoted as $[ts^j]_i$ for $j \in [1, B]$. Each $P_{j_{j\in[1,B]}}$ sends $[ts^j]_i$ to S_i. And all secret shares from all $P_{j_{j\in[1,B]}}$ are denoted as $[s\_no\_id_2]_i$ for $i \in [1, 3]$.

S_1, S_2, and S_3 assign ids to the tuples using its local shares by running $([s_{k=2}]_1,[s_{k=2}]_2,[s_{k=2}]_3)\leftarrow$*Assign-Id* $([s\_no\_id_2]_1,[s\_no\_id_2]_2, [s\_no\_id_2]_3)$.

For $k \in [2, K]$, the servers repeat the following steps.

Step 1 (Sort the edges and paths):

- S_1, S_2, S_3 run the oblivious sort operation over the secret shares using the comparator as in Figure 6 as follows. $([os_k]_1, [os_k]_2, [os_k]_3)\leftarrow$*Ob-Sort*$(cmp, [s_k]_1, [s_k]_2, [s_k]_3)$.
- $([os_k]_1, [os_k]_2, [os_k]_3)$ are the secret shares of the sorted tuples.

Step 2 (Obliviously extend paths):

- The servers run the oblivious extend protocol to extend the path as follows. $([es_k]_1,[es_k]_2,[es_k]_3)\leftarrow$*Ob-Extend*$([os_k]_1, [os_k]_2, [os_k]_3)$.
- $([es_k]_1,[es_k]_2,[es_k]_3)$ are the secret shares of the tuples after neighbor passing and extension.

Step 3 (Extract paths from path tuples)

- The servers first shuffle all the secret shares and obtain the shuffled secret shares by running $([st_k]_1, [st_k]_2, [st_k]_3) \leftarrow$ *Ob-Shuffle*$([es_k]_1,[es_k]_2,[es_k]_3)$.
- Over the shuffled tuples, the servers check each tuple t in the shuffled tuples st_k to check the tuple type, we denote the process as $(types_k)\leftarrow$*Check-Tuple-Type*$([st_k]_1, [st_k]_2, [st_k]_3)$ and use $types_k$ to denote the found types of all shuffled tuples.
- We denote all found path tuples as pt_k and edge tuples as et_k. Each server S_i, for $i \in [1, 3]$, parse its local shares $[pt_k]_i$ into $[paths_k]_i$ by running $([paths_k]_i)\leftarrow$*Parse-Path*$([pt]_i)$.

Step 4 (Filter out invalid paths and detect cycles):

- The servers filter out invalid paths in $paths_k$ by running $valid_k\leftarrow$*Private-Filter-Path*$([paths_k]_1,[paths_k]_2, [paths_k]_3)$. $valid_k$ is a vector of boolean values for all paths in $paths_k$ indicating whether a tuple is valid or not.
- We denote $vpaths_k$ as the valid paths and for each path in $vpaths_k$, the servers run $isCycle_k\leftarrow$*Private-Cycle-Detection*$([vpaths_k]_1$,

$[vpaths_k]_2$,$[vpaths_k]_3$). $isCycle_k$ is the vector of all boolean values for all valid paths in $vpaths_k$ indicating whether the path forms a cycle.

- For each cycle, the servers reveal the local shares of the cycles and we use C_k to denote all the detected cycles along with the nodes that form each cycle.
- We denote all the non-cycle and valid paths as vpt_k.

Step 5 (Pad tuples for next round):

- For each edge tuple in et_k, each S_i locally runs *Extend-Edge-Share* to pad the edge tuples for next round of detection. We denote the process as $[pad\_et_k]_i \leftarrow$*Extend-Edge-Share*$([et_k]_i)$ for $i \in [1,3]$. And $[pad\_et_k]_i$ represents all padded edge tuples.
- For each path tuple in vpt_k, each S_i locally runs *Format-Path-From-Share* to pad the path tuples for next round of detection. We denote the process as $[pad\_pt_k]_i \leftarrow$*Format-Path-From-Share*$([vpt_k]_i)$. And $[pad\_pt_k]_i$ represents all padded path tuples.
- $[pad\_et_k]_i$ and $[pad\_pt_k]_i$ are denoted as $[s\_no\_id_{k+1}]_i$ for $i \in [1,3]$. And the servers assign ids to the tuples by running $([s_{k+1}]_1,[s_{k+1}]_2,[s_{k+1}]_3) \leftarrow$*Assign-Id* $([s\_no\_id_{k+1}]_1,[s\_no\_id_{k+1}]_2, [s\_no\_id_{k+1}]_3)$.
- $[s_{k+1}]_i$ is the local secret share of S_i to be used in next round of cycle detection of length $k+1$.

A.2 Simulation proof

Without loss of generality, we assume that S_1 is the adversary in the proof. We build a simulator *Sim* for one of the computing servers and use $\mathcal{A}$ to denote an adversary who corrupts S_1. In following simulation, when three secret shares are inputs, $\mathcal{A}$ inputs its own secret share and *Sim* inputs another two secret shares. Recall that $\mathcal{F}$ is the ideal functionality given in Figure 3.

Sim for S_1

Step 0 (Create secret shares of graph):

- $\mathcal{F}$ outputs $|V_i| + |E_i|$ for $i \in [1,B]$, pn_1, and C_k, pn_k for $k \in [2,K]$ to *Sim*. $|E| = pn_1$ and $|V|$ is derived by computing $|V| = \sum_i(|V_i| + |E_i|) - |E|$.
- *Sim* first creates an empty graph G' with $|V|$ nodes with no edges. For each edge e' in the detected cycles C_k for $k \in [2,K]$, the edge e' is added to G'.
- *Sim* takes a greedy approach to try and add edges into G' such that the numbers of paths of length 2 to K are $cpath_2, \ldots, cptah_k$ respectively, there are no other cycles and exactly $|E|$ edges in G'.
- *Sim* creates the secret shares of the graph G' as in Section 6.3. The total number of tuples is $|V| + |E|$. And these secret shares are denoted as gs_i for $i \in [1,3]$.
- *Sim* partition gs_i into $[ts'^j]_i$ for $j \in [1,B]$ such that the number of tuple secret shares of $[ts'^j]_i$ is $|V_j| + |E_j|$.
- All $[ts'^j]_1$ of all $j \in [1,B]$ are sent to $\mathcal{A}$.
- $[ts'^j]_i$ of all $j \in [1,B]$ are denoted as $[s\_no\_id'_k]_i$ for $i \in [1,3]$.
- *Sim* and $\mathcal{A}$ assign ids by running $([s_{k=2}{}']_1,[s_{k=2}{}']_2,[s_{k=2}{}']_3) \leftarrow$*Assign-Id* $([s\_no\_id'_2]_1,[s\_no\_id'_2]_2, [s\_no\_id'_2]_3)$.

For $k \in [2,K]$, *Sim* and $\mathcal{A}$ repeat the following steps.

Step 1 (Sort the edges and paths):

- *Sim* and $\mathcal{A}$ run the oblivious sort operation over the secret shares using the comparator as in Figure 6 as follows. $([os_k{}']_1, [os_k{}']_2, [os_k{}']_3) \leftarrow$*Ob-Sort*$(cmp, [s_k{}']_1, [s_k{}']_2, [s_k{}']_3)$.

Step 2 (Obliviously extend paths):

- The servers run the oblivious extend protocol to extend the path as follows. $([es_k{}']_1,[es_k{}']_2,[es_k{}']_3) \leftarrow$*Ob-Extend*$([os_k{}']_1, [os_k{}']_2,[os_k{}']_3)$.

Step 3 (Extract paths from path tuples)

- *Sim* and $\mathcal{A}$ shuffle all the secret shares by running $([st_k{}']_1, [st_k{}']_2, [st_k{}']_3) \leftarrow$ *Ob-Shuffle*$([es_k{}']_1,[es_k{}']_2,[es_k{}']_3)$.
- *Sim* and $\mathcal{A}$ check the tuple type of all shuffled tuples by running $(types_k{}') \leftarrow$*Check-Tuple-Type*$([st_k{}']_1, [st_k{}']_2, [st_k{}']_3)$. $types_k{}'$ is the found types of all shuffled tuples.
- We denote all found path tuples as $pt_k{}'$ and edge tuples as $et_k{}'$.
- $\mathcal{A}$ parses its local shares $[pt_k{}']_1$ into $[paths_k{}']_1$ by running $([paths_k{}']_1) \leftarrow$*Parse-Path*$([pt']_1)$.
- *Sim* parses its local shares $[pt_k{}']_2$ and $[pt_k{}']_3$ into $[paths_k{}']_2$ and $[paths_k{}']_3$ as $\mathcal{A}$ does above.

Step 4 (Filter out invalid paths and detect cycles):

- *Sim* and $\mathcal{A}$ filter out invalid paths by running $valid_k{}' \leftarrow$*Private-Filter-Path*$([paths_k{}']_1,[paths_k{}']_2, [paths_k{}']_3)$. $valid_k{}'$ is a vector of boolean values for all paths in $paths_k{}'$ indicating whether a tuple is valid or not.
- We denote $vpaths_k{}'$ as the valid paths. *Sim* and $\mathcal{A}$ run $isCycle_k{}' \leftarrow$*Private-Cycle-Detection*$([vpaths_k{}']_1, [vpaths_k{}']_2,[vpaths_k{}']_3)$. $isCycle_k{}'$ is the vector of all boolean values for all valid paths in $vpaths_k{}'$ indicating whether the path forms a cycle.
- For each cycle, *Sim* and $\mathcal{A}$ reveal the local shares of the cycles which is C_k.
- We denote all the non-cycle and valid paths as $vpt_k{}'$.

Step 5 (Pad tuples for next round):

- For each edge tuple in et_k, $\mathcal{A}$ locally run $[pad\_et_k{}']_1 \leftarrow$*Extend-Edge-Share*$([et_k{}']_1)$. And *Sim* does the same for its local shares.

- For each path tuple in vpt_k, $\mathcal{A}$ locally runs $[pad\_pt'_k]_1 \leftarrow$ *Format-Path-From-Share*$([vpt_k{}']_1)$. And *Sim* does the same for its local shares.
- $[pad\_et'_k]_i$ and $[pad\_pt'_k]_i$ are denoted as $[s\_no\_id'_{k+1}]_i$ for $i \in [1,3]$. And *Sim* and $\mathcal{A}$ assign ids to the tuples by running $([s'_{k+1}]_1,[s'_{k+1}]_2,[s'_{k+1}]_3) \leftarrow$ *Assign-Id* $([s\_no\_id'_{k+1}]_1,[s\_no\_id'_{k+1}]_2,[s\_no\_id'_{k+1}]_3)$.

The view of S_1 in the real world includes:

- $|V|, |E|$
- $[ts^i]_1, |V_i| + |E_i|$ for $i \in [1, B]$
- For $k = 2$ to K:
  - $[s\_no\_id_k]_1$
  - $[s_k]_1$, $[os_k]_1$, $[es_k]_1$, $[st_k]_1$
  - $[pt_k]_1$, $[et_k]_1$, $[vpt_k]_1$
  - $[paths_k]_1$, $[vpaths_k]_1$
  - $types_k$, $valid_k$, $isCycle_k$
  - $[pad\_et_k]_1$, $[pad\_pt_k]_1$
  - C_k, pn_k

The view of $\mathcal{A}$ in the ideal world includes:

- $|V|, |E|$
- $[ts'^i]_1, |V_i| + |E_i|$ for $i \in [1, B]$
- For $k = 2$ to K:
  - $[s\_no\_id'_k]_1$
  - $[s_k{}']_1$, $[os_k{}']_1$,$[es_k{}']_1$,$[st_k{}']_1$
  - $[pt_k{}']_1$,$[et_k{}']_1$, $[vpt_k{}']_1$
  - $[paths_k{}']_1$, $[vpaths_k{}']_1$
  - $types_k{}'$, $valid_k{}'$,$isCycle_k{}'$,
  - $[pad\_et_k{}']_1$, $[pad\_pt_k{}']_1$
  - C_k, pn_k

Now we compare the two views in both worlds. All the secret shares in both views are uniform random numbers thus are indistinguishable. So we only need to compare the number of secret shares in both views. $[ts^j]_1$ and $[ts'^j]_1$ have the same size of $|V_i| + |E_i|$ for $j \in [1, B]$. $[s\_no\_id_k]_1$,$[s_k]_1$,$[os_k]_1$,$[es_k]_1$, $types_k$, and $[s\_no\_id'_k]_1$,$[s_k{}']_1$,$[os_k{}']_1$,$[es_k{}']_1$, $types_k{}'$ all have $pn_{k-1} + |V|$ elements. $[pt_k]_1$ and $[pt_k{}']_1$ both represents the number of path tuples with pn_{k-1} elements. $[paths_k]_1$, $[paths_k{}']_1$ are induced from $[pt_k]_1$ and $[pt_k{}']_1$ with $d \times pn_{k-1}$ elements. $[et_k]_1$ and $[et_k{}']_1$ both represents the number of edge tuples and have $|V|$ elements. $[vpaths_k]_1$, $[vpaths_k{}']_1$ are secret shares of valid paths and cycles with $pn_k + |C_k|$ elements. $[vpt_k]_1$, $[vpt_k{}']_1$ are secret shares of valid paths with pn_k elements. $[pad\_et_k{}']_1$ and $[pad\_et_k]_1$, $[pad\_pt_k]_1$ and $[pad\_pt_k{}']_1$ are induced from $[et_k]_1$ and $[et_k{}']_1$, $[vpt_k]_1$ and $[vpt_k{}']_1$ respectively, thus have the same size.

Now we compare the remaining non-secret-shared outputs. $types_k$ and $types_k{}'$ have the same amount of zeros and ones as the numbers of edge and path tuples are the same in both worlds. The exact distributions of values are uniform random as they are the results of obliviously shuffled data. For the similar reasoning, $types_k$ and $types_k{}'$, $isCycle_k$ and $isCycle_k{}'$ are indistinguishable. Now we conclude the proof that the views in both worlds are indistinguishable.

B COMPLEXITY ANALYSIS

In this section, we analyze the computation complexity of *Oryx*. We denote the total number of nodes as v, the average number of neighbors as n, the maximum degree as d. In a round of detecting cycles of length k, the number of total subgraphs (tuples) to process in that round is $T = v \cdot n^{k-1}$. In each round of detecting cycles of length k, with maximum degree d, the size of each tuple is $O(kd)$. Each comparison between two numbers in MPC has constant cost when the bit length of the numbers is fixed. For simplicity, we will omit including this constant in the following analysis.

Shuffle for sort. The computation complexity of the shuffle operation proposed by Araki et al. [6] is linear to the number of tuples and the size of each tuple. Thus, it has $O(kdT)$ computation complexity.

Sort over shuffled tuples. The oblivious sort operation first shuffles all the data with $O(kdT)$ complexity. Then the servers use comparison-based sorting such as quicksort over tuples each with size of $O(k)$ (as in Figure 6) with $O(kT\log(T))$ complexity. In total, the complexity is $O(kT(d + \log(T)))$.

Neighbor passing and path extension. The neighbor passing and path extension takes a linear pass over each tuple. In each iteration, the protocol does one comparison over the *src* field, and then either reads or writes the variable, *neighbors*, which has d elements. As a result, it has complexity of $O(dT)$.

Shuffle in filtering. In the first shuffle operation during the filtering phase, T tuples are shuffled, with each tuple having a size of kd. Therefore, the first shuffle has complexity of $O(kdT)$. The second shuffle involves shuffling the path vectors, and their count is at most dT, with each path vector having a size of $O(k)$. Thus, the second shuffle's complexity is $O(kdT)$.

Check the type of tuples. To evaluate which tuples are path tuples in all the shuffled tuples, servers perform one comparison of the *src* field for each tuple. In total, this is $O(T)$.

Filtering and cycle detection. The filtering and cycle detection involve comparing the last elements with all the previous nodes of each tuple, requiring k comparisons. Given there are at most dT path vectors in total as each path tuple is parsed into d vectors, this part has computation complexity of $O(kdT)$.

Padding tuples. Padding tuples for the next round iterate through each tuple and has $O(T)$ complexity.

Total computation complexity. In total, our protocol has computation complexity of $O(kT(d + \log(T)))$.