PRIVACY-PRESERVING FINANCIAL FRAUD DETECTION

by

Tejas Sarvasetty

May 2026

A dissertation submitted to the
Faculty of the Graduate School of
the University at Buffalo, State University of New York
in partial fulfillment of the requirements for the
degree of

Master of Science

Department of Computer Science and Engineering

Acknowledgments

Foremost, I would like to express my sincere gratitude to my advisor, Professor Marina Blanton, for her continuous support throughout my master's studies. It was through her seminar class that I was introduced to the concepts of secure computation, which deeply inspired my interest in this field of work. I am truly thankful for her patience, enthusiasm, immense knowledge, and constant guidance, all of which helped me successfully complete this thesis. I could not have imagined having a better advisor and mentor.

Besides my advisor, I would also like to thank my family — especially my parents and my brother — for their unwavering love and support throughout my life. I am equally grateful to my friends and cousins, whose encouragement and support have meant so much to me during this journey.

Table of Contents

Abstract

Anomaly detection in transaction networks is more effective when institutions can jointly analyze data distributed across multiple organizations. However, direct sharing of such data is restricted by privacy regulations, confidentiality requirements, and competitive concerns. In this work, we present a privacy-preserving framework for detecting suspicious subgraphs in distributed transaction graphs using secure multiparty computation (MPC). Our approach adapts the AntiBenford Subgraphs method, which combines anomaly scoring based on deviations from Benford's law with densest-subgraph extraction, to the MPC setting. We design data-oblivious algorithms on the core graph computations for both dense and sparse graphs so that both sensitive values and graph access patterns remain hidden during execution. The resulting framework enables privacy-preserving collaborative, unsupervised financial anomaly detection over distributed transaction graphs.

Chapter 1

Introduction

Collaborative analysis of financial data is increasingly important for applications such as fraud detection, anti-money laundering, and financial risk analysis. Money laundering is a large-scale global problem, with estimates suggesting that the amount of money laundered through the financial system is on the order of 2%–5% of global GDP, with an estimate of approximately 2.7% (about $1.6 trillion) [35].

In the case of money laundering, the process is generally understood to occur in three phases: placement, layering, and integration [20]. The layering phase, in particular, involves moving funds between accounts, banks, and jurisdictions in order to conceal the origin of illicit proceeds and make transactions appear legitimate. Detecting such behavior is a complex and resource-intensive task, as transaction data is distributed across multiple institutions and regulatory constraints limit any single entity's visibility into global financial flows [18].

Transaction data is also highly sensitive. Banks and payment providers are often unable to share raw transaction records due to regulatory restrictions imposed in many countries, as well as competitive concerns and the need to protect customer privacy. As a result, individual institutions only have access to their own customers' transaction data. This creates a need for approaches that enable collaborative and secure analysis of financial data, allowing institutions to work together without directly sharing sensitive information.

Secure multiparty computation (MPC) is a mechanism that allows multiple parties to jointly compute a function over their private data. It can be viewed as $(y_1, y_2, \ldots, y_n) = f(x_1, x_2, \ldots, x_n)$, where each party privately provides input x_i and only receives output y_i, without learning anything else about the other parties' inputs. Both the inputs and outputs are typically secret-shared, which helps protect the underlying data values. There are existing frameworks, such as PICCO [36] and MP-SPDZ [21], that support the construction and execution of programs in an MPC setting.

However, while secret sharing protects the data itself, it does not protect the access patterns to that data. Therefore, in the MPC setting, algorithms must be data oblivious. An algorithm is data-oblivious if its control flow and memory access patterns are independent of the actual input values and depend only on the input size. That is, for all inputs of the same length, the sequence of memory accesses made by the program remains identical. In most cases, standard algorithms are not data-oblivious and therefore need to be adapted or redesigned to work correctly and securely within the MPC setting. These challenges become particularly important when applying graph-based methods for fraud detection.

Financial data can naturally be modeled as a transaction graph, where vertices represent accounts and edges between vertices represent transactions. This forms a type of knowledge graph, where attributes such as transaction amounts provide useful signals for identifying anomalies in financial networks. One important statistical property that can be leveraged in this context is Benford's law, also known as the Newcomb-Benford law or the first-digit law. It states that in many real-world datasets, the occurrence of the leading digit is not uniformly distributed, but instead follows a logarithmic distribution given by $\log_{10}(1 + 1/d)$. Deviations from this expected distribution can be indicative of manipulated or fraudulent data. Using this observation, [13] develops a framework called anti-Benford subgraphs that identify dense subgraphs in transaction networks whose edge weights significantly deviate from the expected Benford distribution. Such subgraphs often correspond to coordinated fraudulent activity, making them particularly useful for applications like fraud detection and

anti-money laundering.

Many graph algorithms depend on full access to the graph structure, operate directly on it, and rely heavily on data-dependent operations. For example, in breath first search (BFS), each iteration selects nodes only belonging to the next level based on the current frontier. Similarly, in densest subgraph algorithms, selecting vertices based on degree, traversing specific edges, or iteratively modifying the graph structure reveals information about connectivity and node degrees through their access patterns. To address this challenge, graph algorithms in MPC must be adapted to operate in a data-oblivious manner, ensuring that both the data values and the access patterns remain hidden.

In this work, we develop a privacy-preserving framework for detecting suspicious transaction subgraphs. Our starting point is the Anti-Benford Subgraphs method of [13], which combines Benford-law-based anomaly scoring at the node level and densest-subgraph extraction to identify suspicious subgraphs in transaction networks. We adapt this computation to the MPC setting. Our main contribution is the design of data-oblivious algorithms for the core graph computations, in particular densest-subgraph computation for both dense and sparse graph representations. Our work shows how unsupervised graph-based fraud detection methods can be restructured to satisfy the privacy requirements of collaborative financial analysis.

The remainder of the work is organized as follows. Section 2 reviews related work on fraud detection, secure collaborative financial analysis, privacy-preserving machine learning, and graph-based fraud detection. Section 3 introduces the required background on AntiBenford subgraphs, the densest subgraph problem, and the graph-processing primitives used in our constructions. Section 4 presents our privacy-preserving algorithms, including data-oblivious methods for dense and sparse graphs and an oblivious AntiBenford subgraph computation.

Chapter 2

Related Work

2.1 Conventional Fraud Detection Mechanisms

Conventional fraud detection methods increasingly model financial activity as graphs in order to analyze transaction patterns and identify suspicious behavior. Some approaches rely on handcrafted temporal and motif-based patterns in transaction networks; for example, [34] identifies suspicious sets of transactions by detecting a money laundering motif known as "smurfing."

Subsequent work explores graph-based anomaly detection techniques. For instance, [8] represents transactions between cards and merchants as a bipartite graph and detects fraudulent activity by identifying anomalous bicliques composed predominantly of fraudulent cards. HoloScope [25] is an unsupervised fraud detection method that combines graph topology with temporal event spikes and user rating to identify suspicious users. FlowScope [24] models laundering activity as a dense flow subgraph, detecting suspicious high-volume transfer patterns across multiple accounts. LaundroGraph [11] presents a supervised system combining graph neural networks for AML (anti-money laundering) detection.

In parallel, classical machine learning methods have also been widely applied to fraud detection on financial data [32, 4, 29, 27]. However, these approaches typically focus on

individual transactions or accounts rather than relational patterns across the full transaction network.

The conventional (non-secure) computation we adopt in this work is AntiBenford Subgraphs [13], which detects anomalous subgraphs in financial transaction networks using digit-based anomaly scores together with densest-subgraph techniques. Our work builds upon this line of graph-based anomaly detection while making the computation privacy-preserving.

An advantage of AntiBenford Subgraphs is that it combines a Benford-law-based anomaly signal with an unsupervised and scalable dense-subgraph mining procedure. As a result, it can detect suspicious transaction subgraphs without labeled data, remains practical on large networks, and has been shown to identify anomalous Ethereum subgraphs that are missed by methods such as HoloScope and FlowScope [13].

2.2 Secure Collaborative Financial Analysis

While the conventional methods have been described for detecting suspicious financial behavior, they do not address the setting where data is distributed across multiple institutions and cannot be shared directly. This has motivated research into privacy-preserving methods for collaborative financial analysis. [6] describes a system for analyzing financial data and business metrics securely using secret sharing and secure multiparty computation (MPC) implemented in the Sharemind framework [5], where the participating parties are geographically separated. [14] presents a secure system for banks to analyze customer information by comparing it with confidential data from another entity using linear programming and SPDZ [21]. These results demonstrate that MPC can support joint analysis of sensitive financial data without revealing each participant's raw inputs. Although these systems do not primarily focus on fraud detection or transaction graph analysis, they establish the feasibility of secure collaborative computation.

2.3 Privacy-Preserving Machine Learning for Fraud Detection

Another direction explores privacy-preserving approaches to financial analytics for fraud detection using machine learning models. [10] presents a study of a homomorphic-encryption-based fraud detector using a decision-tree classifier. They report that the large encryption key sizes required under homomorphic encryption significantly increase the latency of each classification request. [9] demonstrates an approach for detecting credit-card fraud with logistic regression. [3] proposes a framework to perform anomaly detection for banks acting as a central payment router by training a neural network. Both these works use federated learning (FL) to train a global model where each party updates the model using only its local data, in order to keep it private. However, federated learning alone does not fully guarantee privacy, as information may still leak through model updates to the aggregation server. It is therefore combined with differential privacy [28, 2] to ensure stronger privacy guarantees; however, this introduces an accuracy tradeoff. [1] introduces a privacy-preserving federated learning mechanism for fraud detection that addresses scalability and efficiency issues in prior FL-based implementations. These works show that analysis can be performed without pooling data. However, these approaches do not model the transaction network and instead focus on detecting anomalous individual accounts.

2.4 Privacy-Preserving Graph-Based Fraud Detection

Financial data can be modeled as transaction graphs, where nodes represent entities such as account holders and edges represent transactions between two endpoint nodes. Analytical techniques can then be applied to these graphs to detect suspicious patterns and relationships. For example, tasks such as fraud detection and anti-money laundering can be performed using transaction graph analysis.

[26] models data as a graph and develops a collaborative fraud detection framework for large-scale graph data that combines MPC and graph neural networks. Their work addresses privacy, graph structure, and scalability. However, their evaluation is not performed on a financial dataset. [16] studies collaborative anti-money laundering with fully homomorphic encryption (FHE), exploring two privacy-preserving pipelines: an FHE-compatible graph neural network (GNN) and a graph-feature-based XGBoost pipeline.

[17] proposes a privacy-preserving AML approach based on secure risk propagation across an inter-bank transaction network. Their method starts from per-account risk scores and propagates those scores through the graph using MPC, showing that cross-bank visibility can improve detection performance while preserving confidentiality. Similarly, [33] develops a secure multiparty PageRank protocol for collaborative fraud detection, allowing institutions to compute PageRank values over their joint transaction graph without revealing the underlying graph to one another. Both of these approaches use homomorphic encryption to enable privacy-preserving computation. These approaches are practical and graph-aware, but both depend on propagating a predefined risk over the network.

While prior work demonstrates privacy-preserving financial analysis and collaborative fraud detection, most approaches either rely on machine learning models applied to individual accounts rather than modeling the full transaction graph, or propagate predefined risk scores across the network. In contrast, our work focuses on privacy-preserving detection of anomalous transaction subgraphs using densest-subgraph techniques, enabling unsupervised detection without requiring labeled data.

Chapter 3

Background

We use uppercase bold uppercase letters for matrices, and bold lowercase letters for vectors. Thus, $G = (V, E, w)$ denotes a weighted graph, $\mathbf{W} \in \mathbb{R}^{n \times n}$ with $n = |V|$ denotes a weighted adjacency matrix, and $\mathbf{x} \in \mathbb{R}^n$ denotes a vector (consistent with prior literature, we represent V and E using the set notation). The (i, j) entry of $\mathbf{W}$ is written as $\mathbf{W}_{ij}$, and the i-th entry of $\mathbf{x}$ is written as $\mathbf{x}_i$. The i-th row of $\mathbf{W}$ is written as $\mathbf{W}_{i,:}$, and the i-th column is written as $\mathbf{W}_{:,i}$. The vector $\mathbf{1}_{1 \times n}$ represents the all-ones vector of length n. A one-hot vector $\mathbf{v} \in \{0, 1\}^n$ has exactly one nonzero entry, with $\mathbf{v}_k = 1$ indicating that vertex k is selected. In the secure computation setting, we use square brackets $[\cdot]$ to denote protected (secret-shared) values.

3.1 AntiBenford Subgraphs

The algorithm described by Chen and Tsourakakis [13] is based on Benford's law. Benford's law describes the distribution of the first digit of numbers that appear in many kinds of numerical data, including tax records and election results, and can be utilized in detecting fraud [15].

The input data is transaction data that can be viewed as a transaction multigraph, where multiple transactions might exist between two accounts, forming $G_T = (V, E_T)$, where each vertex represents an account and each edge corresponds to a single transaction. In practice,

this graph is represented as a transaction list of the form (`from_account`, `to_account`, `amount`).

Before the algorithm begins, this transaction list is transformed into a graph $G = (V, E, \mathbf{X})$ used by the AntiBenford computation. Here, the vertices in V are the accounts, an edge $(u, v) \in E$ exists if there is at least one transaction, and each vertex stores first-digit transaction counts. More specifically, for each vertex u we store a digit-count vector, $\mathbf{X}_{u,d}$, where the d-th entry is the number of transactions u is involved in that have the first digit d. This preprocessing also includes grouping individual transactions by account pair, computing these digit counts, determining the number of unique accounts $|V|$, and renumbering the accounts from $1, \ldots, |V|$ instead of using their original identifiers. This preprocessing can be performed on cleartext data before entering secure computation.

The approach then begins by computing an anomaly score $s(u)$ for each node u. This score measures how much the transactions incident to u deviate from the Benford distribution. For each digit $d \in \{1, \ldots, 9\}$, Benford's law describes the expected probability of digit d as

$$p_d = \log_{10}(1 + 1/d).$$

If $\deg(u)$ is the number of transactions incident to u, then the expected number of transactions at u whose first digit is d is

$$\mathbb{E}_{u,d} = p_d \cdot \deg(u) = p_d \cdot \sum_{d=1}^{9} \mathbf{X}_{u,d}$$

The anomaly score is then computed using a chi-square test:

$$s(u) = \sum_{d=1}^{9} \frac{(\mathbf{X}_{u,d} - \mathbb{E}_{u,d})^2}{\mathbb{E}_{u,d}}.$$

After that, the algorithm assigns new edge weights between nodes based on the anomaly scores:

$$W_{uv} = \sqrt{[s(u)] \cdot [s(v)]}.$$

Algorithm 1 AntiBenford Subgraph Computation

Input: $G = (V, E, X)$
Output: Set of nodes $S \subset V$

for $u \in V$ **do**
  $s(u) = \sum_{d=1}^{9} \frac{\left(X_{u,d} - \mathbb{E}_{u,d}\right)^2}{\mathbb{E}_{u,d}}$
end for
for $(u, v) \in E$ **do**
  $W_{uv} = \sqrt{s(v) \cdot s(u)}$
end for
$S = \mathsf{DSP}(G' = (V, E, W))$
if $\neg(\psi(S) \gg \psi(V))$ **then**
  return $\emptyset$
end if
return S

So an edge gets a high weight when both of its endpoints have high anomaly scores. The algorithm then finds the densest subgraph using an iterative greedy peeling algorithm. At the end, the algorithm checks whether the result of evaluating the ψ function on the returned subgraph is significantly different from the result of its evaluation on the entire graph. The ψ function on input a set of vertices S is defined as:

$$\psi(S) = \frac{\chi^2(S)}{|S|}, \tag{3.1}$$

where

$$\chi^2(S) = \sum_{d=1}^{9} \frac{\left(X_{S,d} - (p_d \cdot |S|)\right)^2}{p_d \cdot |S|}. \tag{3.2}$$

Here, $X_{S,d}$ is the number of edges in the set of edges induced by S, $E(S)$, whose first digit is d, $|S|$ is the number of edges in S, and p_d is the publicly known Benford probability for digit d. If the deviation of the subgraph induced by S from the overall graph is not statistically significant, it is rejected. To find multiple suspicious subgraphs, the algorithm can be run repeatedly on the residual graph obtained after removing the previously detected subgraph.

Algorithm 1 summarizes this computation.

3.2 Densest Subgraph Problem (DSP)

For a weighted graph $G = (V, E, w)$, the density of a set of vertices $S \subseteq V$ is defined as

$$D(S) = \frac{\sum_{(u,v)\in E(S)} w(u, v)}{|S|}$$

where $E(S)$ denotes the set of edges induced by S. The densest subgraph problem is to find a subset of vertices V_{densest} such that

$$V_{\text{densest}} = \arg\max_{S\subseteq V} D(S).$$

There exist both exact and approximate algorithms for the densest subgraph problem. Picard and Queyranne [31] presented one of the earliest exact algorithms as a series of maximum-flow computations, and Goldberg [19] later proposed an improved version. Charikar [12] also designed an LP-based exact algorithm. However, although these algorithms are solvable in polynomial time, they can still be expensive in practice, especially on large graphs, since maximum-flow computations are computationally costly. For this reason, Charikar [12] also proposed a 2-approximation algorithm based on iteratively removing from the graph the vertex with the smallest degree. Boob et al. [7] proposed Greedy++ for DSP; this algorithm runs Charikar's greedy peeling algorithm for T rounds while updating the priority of each vertex using information from past iterations. In our work, we use the algorithm proposed by Charikar.

The algorithm, included as Algorithm 2, begins by initializing a copy S of the original vertex set V and marking it as the current densest vertex set V_{densest}. While $|S| > 1$, the algorithm repeatedly identifies the vertex v_i with the minimum weighted degree (within the subgraph induced by S) and removes it from S. After each removal, if the density of the subgraph induced by S is greater than that of the previously recorded vertex set V_{densest}, then V_{densest} is updated to S. Once $|S| = 1$, the algorithm terminates, and the subgraph

Algorithm 2 Greedy Algorithm for Densest Subgraph Problem

Input: $G = (V, E, w)$
Output: Densest subgraph vertices $V_{\text{densest}} \subseteq V$

$S \leftarrow V$
$V_{\text{densest}} \leftarrow V$
while $|S| > 1$ **do**
  $v_i = \arg\min_{v \in S'} \sum_{(v,u) \in E(S')} w(v, u)$
  $S \leftarrow S \setminus \{v_i\}$
  if $D(S) > D(V_{\text{densest}})$ **then**
    $V_{\text{densest}} \leftarrow S$
  end if
end while
return V_{densest}

induced by V_{densest} is the densest subgraph. In the weighted version, the degree of a vertex is defined as the sum of the weights of its incident edges.

The original complexity of Algorithm 1 is dominated by the densest subgraph computation which can be performed in near-linear time [23]. Hence, the complexity of Algorithm 1 is $O(|V| + |E| + T_{DSP})$. The minimum-degree selection for DSP can be implemented in linear time by maintaining vertices in buckets indexed by degree, using doubly linked lists [23]. At each step, a vertex is removed from the lowest non-empty bucket, and its neighbors are moved to the appropriate lower-degree buckets. Therefore, $T_{DSP} = O(|V| + |E|)$.

This algorithm is not data oblivious, and its access patterns do not protect the structure of the graph. To find the vertex V_i with minimum weighted degree, the algorithm must access the weights of its incident edges. This leaks structural information about the graph, in particular about the degree of a vertex.

When the density of the subgraph induced by S is compared with that of the subgraph induced by V_{densest}, and V_{densest} is updated, this reveals the size of the densest subgraph, that is, the number of vertices and edges in it.

3.3 Oblivious Graph Operations

In secure computation, protecting the data itself is not enough. When accessing data from a data structure, we must ensure that the access pattern does not reveal sensitive information. For example, if the same elements of an array are accessed repeatedly, an adversary observing the pattern could infer which elements are being used, even if the values themselves remain hidden. To prevent this type of leakage, one approach is to access all elements of the data structure.

Graphs are typically represented in two ways. One is as an adjacency matrix, where the element (i, j) in the matrix holds the weight of the edge from vertex i to vertex j if the edge exists. Working with graphs in this representation will incur a computation cost of $O(|V|^2)$ in the secure computation setting, because to hide access patterns every element of the matrix must be touched.

In the real world many graphs are sparse, so using an adjacency matrix results in unnecessary computation. Instead, if we represent the graph as an adjacency list, the complexity of the graph representation becomes $O(|V| + |E|)$. If the graph is sufficiently sparse, it is more efficient to represent it in this way.

In order to securely compute on an adjacency list, prior work GraphSC proposes a general abstraction for representing and processing adjacency-list graphs as message passing algorithms.

GraphSC [30] describes a framework for secure graph computation inspired by Pregel. The framework operates on a (DAG) data-augmented directed graph represented as $G(V, E, D)$, which contains the standard vertices and edges and, in addition, a data component D. The data component represents the information that is passed between connected vertices through the edges of the graph. The graph itself is represented as a list $\mathcal{G}$, with an entry corresponding to every node and edge in the graph and of size $|V| + |E|$.

The framework describes three primitives: Scatter, Gather, and Apply. The Scatter

operation propagates information from a vertex into its connected outgoing edges by copying its data elements into the edge data. The Gather operation then aggregates the data from the incoming edges into a vertex. The Apply operation is finally performed on the gathered data. Using these primitives, any graph algorithm can be realized.

To perform these primitives in a data-oblivious way, the graph vertices and edges must be represented in such a way that they cannot be differentiated. This is done by representing each entry in the list $\mathcal{G}$ as a tuple $(u, v, \texttt{data}, \texttt{isV})$, where u and v represent the source and destination, `data` is the data component, and `isV` indicates whether the tuple represents a vertex or an edge. For a vertex tuple, u and v are the same, representing the vertex identifier; `data` stores the vertex data, and `isV` is set to true. For an edge, u and v correspond to the endpoints of the edge, where u is the source vertex and v is the destination vertex, `data` is empty, and `isV` is set to false. The algorithm is described for a directed graph, but for an undirected graph an edge between u and v can be represented using two directed edges (u, v) and (v, u).

GraphSC pipeline: Sort $\mathcal{G}$ by source ordering $\rightarrow$ Scatter $\rightarrow$ Sort $\mathcal{G}$ by destination ordering $\rightarrow$ Gather $\rightarrow$ Apply.

The algorithm involves storing the list $\mathcal{G}$ in both source ordering and destination ordering. In source ordering, the list $\mathcal{G}$ is arranged such that each vertex appears before its outgoing edges. In destination ordering, the list $\mathcal{G}$ is arranged such that the entries corresponding to the incoming edges of a vertex appear before the vertex entry itself. The algorithm switches between these two orderings during the computation.

In the Scatter step, once the graph list $\mathcal{G}$ is sorted by source ordering, the data from a vertex is propagated to its outgoing edges. Consider a vertex u_1 with outgoing edges (u_1, v_1) and (u_1, v_2). The data from u_1 is propagated to each of these edges. After the data is propagated to the outgoing edges, the list is sorted into destination ordering. In this ordering, the edges that are incoming to vertex v_i appear before the vertex itself. For example, the edges (u_1, v_1) and (u_2, v_1) appear before the vertex entry (v_1, v_1) in the list.

Algorithm 3 Scatter

Input: graph-list $\mathcal{G}$ in source order

1: $tmp \leftarrow 0$
2: **for** $i = 1$ to $|\mathcal{G}|$ **do**
3: $\quad \mathcal{G}_i.\texttt{data} \leftarrow tmp + (\mathcal{G}_i.\texttt{data} - tmp) \cdot \mathcal{G}_i.\texttt{isV}$
4: $\quad tmp \leftarrow tmp + (\mathcal{G}_i.\texttt{data} - tmp) \cdot \mathcal{G}_i.\texttt{isV}$
5: **end for**

Algorithm 4 Gather

Input: graph-list $\mathcal{G}$ in destination order

1: $agg \leftarrow 0$
2: **for** $i = 1$ to $|\mathcal{G}|$ **do**
3: $\quad \mathcal{G}_i.\texttt{data} \leftarrow agg \cdot \mathcal{G}_i.\texttt{isV}$
4: $\quad agg \leftarrow (agg + \mathcal{G}_i.\texttt{data}) \cdot (1 - \mathcal{G}_i.\texttt{isV})$
5: **end for**

The Gather operation is then performed, which accumulates the data from these edges into the vertex. If the list entry corresponds to an edge entry, it is not impacted by accumulation. Finally, an Apply function is performed on the accumulated vertex data.

A graph algorithm in the GraphSC framework is executed by repeatedly performing the Scatter, Gather, and Apply operations. In each iteration, the data stored at the vertices is propagated to its neighboring edges during the Scatter step. This data is then aggregated from the incident edges into the corresponding vertex during the Gather step, and the vertex data is updated in the Apply step. Since the sorting between source ordering and destination ordering is performed in an oblivious manner, and the Scatter and Gather operations are executed on every entry in the list, every element of the list $\mathcal{G}$ is accessed regardless of whether it represents a vertex or an edge. As a result, the access patterns do not reveal any information about the structure or data of the graph. This iterative message passing process continues until the algorithm converges or until a fixed number of iterations has been completed. The Scatter and Gather procedures are shown in Algorithms 3 and 4, respectively.

3.4 Optimization of GraphSC Primitives

The follow-up work [22] optimizes the primitives defined in the previous section, but before we proceed with the description we provide some additional background. One way to achieve MPC is through a secret sharing scheme, where a secret value is split into shares distributed among computing parties, such that no individual party learns anything about the secret. Computations are then performed on these shares directly. In a linear secret sharing scheme, such as Shamir secret sharing, addition and subtraction operations are non-interactive each party can compute the sum of their shares locally, without any communication. Multiplication, however, is interactive: computing the product of two secret-shared values requires the parties to exchange information, requiring one round of communication.

The efficiency of a secure computation solution is characterized by three metrics: the computation complexity, which is the amount of computation performed by each party locally; the communication complexity, which is the amount of information exchanged between the parties; and the round complexity, which is the number of sequential interactions among the computing parties. The round complexity of a protocol is determined by the number of sequential interactive operations it performs. Looking at the GraphSC Scatter and Gather primitives (Algorithms 3 and 4), we see that they involve repeated multiplication operations over the graph list $\mathcal{G}$, resulting in a linear round complexity of $O(|V| + |E|)$. Therefore they need to be optimized to reduce round complexity.

In order to address this, Graphiti [22] proposes a solution, by modifying the Scatter and Gather operations to use non-interactive addition and subtraction operations. To enable this approach, a new ordering called vertex ordering is introduced along with the previously discussed source and destination ordering. A vertex order of the graph list $\mathcal{G}$ is an ordering in which all the vertices of the graph, ordered by their vertex identifier, appear first, followed by the edges of the graph.

Pipeline: Sort $\mathcal{G}$ by vertex ordering $\rightarrow$ compute vertex data to be propagated $\rightarrow$ Sort $\mathcal{G}$

Algorithm 5 Scatter optimized

Input: graph list $\mathcal{G}$ in vertex order

1: For $i = 1$ to $|\mathcal{G}|$
2: $\mathcal{G}_i.\texttt{data\_s}' = \mathcal{G}_i.\texttt{data\_s} - \mathcal{G}_{i-1}.\texttt{data\_s}$
3: Sort $\mathcal{G}$ in source ordering
4: For $i = 1$ to $|V|$
5: $\mathcal{G}_i.\texttt{data\_r} = \sum_{j=1}^{i} \mathcal{G}_j.\texttt{data\_s}'$

Algorithm 6 Gather optimized

Input: graph list $\mathcal{G}$ in destination order

1: For $i = 1$ to $|\mathcal{G}|$
2: $\mathcal{G}_i.\texttt{data\_g} = \sum_{j=1}^{i} \mathcal{G}_j.\texttt{data\_r}$
3: Sort $\mathcal{G}$ in vertex ordering
4: For $i = |\mathcal{G}|$ to 2
5: $\mathcal{G}_i.\texttt{data\_g} = \mathcal{G}_i.\texttt{data\_g} - \mathcal{G}_{i-1}.\texttt{data\_g}$

by source ordering → Propagate → Sort $\mathcal{G}$ by destination ordering → Gather → Sort $\mathcal{G}$ by vertex ordering → compute gathered vertex data.

In this method each element of the graph list $\mathcal{G}$ is represented as a tuple $(u, v, \texttt{data})$. The data component is a tuple $(\texttt{data\_s}, \texttt{data\_s}', \texttt{data\_r}, \texttt{data\_g})$. Here, `data_s` is the vertex data that is propagated, `data_s'` is an intermediate variable, `data_r` stores the received propagated data, and `data_g` stores the gathered data. The optimized Scatter and Gather procedures are given in Algorithms 5 and 6, respectively.

Simply put, the Scatter step first computes the data to be propagated as a backward difference over the vertices, then propagates this information into the outgoing edges of a vertex by computing the prefix sum of these backward differences up to the graph cell $\mathcal{G}_i$, using the source ordering of $\mathcal{G}$. If we did not take the backward difference, the prefix sum would accumulate extra information from vertices that are not the source vertex for the element of the graph list $\mathcal{G}$. After the Scatter step is done, the edge data can optionally be updated with a custom function.

The Gather step is to accumulate the data into a vertex from the incoming edges using a prefix sum. Since the prefix sum accumulates information from all edges and not only those directly incoming to the vertex, we take a backward difference to remove the extra

information. After the gather step a function can be applied on the vertex data similar to graphSC final apply step.

Chapter 4

Privacy-Preserving Algorithms

We next present data-oblivious algorithms for both sparse and dense graphs.

4.1 Data-Oblivious Densest Subgraph for Dense Graphs

A convenient way to represent a dense graph on $n = |V|$ vertices is as a weighted adjacency matrix. In a weighted adjacency matrix $\mathbf{W} \in \mathbb{R}^{n \times n}$, the entry $\mathbf{W}_{ij}$ is the weight of the edge from vertex i to vertex j if such an edge exists, and is 0 otherwise. For simplicity, we refer to the weighted adjacency matrix simply as the adjacency matrix.

A standard non–data-oblivious approach to compute the densest subgraph for a graph represented as an adjacency matrix can proceed as follows. Initially, we begin by maintaining a record of the weighted degree of each vertex. In each iteration, we pick the vertex with the minimum weighted degree and delete this vertex from the graph. The weighted degree of the neighbors of the vertex must be recalculated. This can be done by subtracting the edge weights from the weighted degree vertex for each neighbor. Deleting the vertex is implemented by setting the weight of all incident edges to that vertex to zero, the vertex is also marked as inactive, and the density of the graph is calculated. The algorithm proceeds to the next iteration, and so on. This algorithm is not data oblivious and leaks information about the structure of the graph. Finding the minimum weighted-degree vertex and deleting

its edges reveals the number of edges incident to the vertex. Each vertex removal also reveals the relative ranking of each vertex's weighted degree.

To address this problem, we introduce an algorithm for the densest subgraph problem on dense graphs given in Algorithm 7. The key idea is to express every step as a fixed sequence of matrix and vector operations, with no data-dependent branching and no selective per-vertex updates. Every iteration performs the same operations regardless of the structure of the input graph. The algorithm maintains an active-node indicator vector, a weighted-degree vector, and the best density seen so far, while performing the same sequence of operations in every iteration.

The first step of the algorithm is calculating the weighted degree for each vertex (line 2). This can be viewed as taking the dot product of the adjacency matrix with a vector of ones, which sums the weights of all incident edges for each vertex. The resulting vector stores the weighted degree of each vertex, where the i-th entry corresponds to the weighted degree of vertex i.

At each iteration, the vertex with minimum degree is selected (line 6) and deleted from the graph. This selection is done using an oblivious argmin operation; the selection produces a one-hot vector corresponding to the minimum-degree vertex (line 7).

For an undirected graph, the adjacency matrix is symmetric; the i-th row or column corresponds to the weights of the edges incident to vertex i. Updating the weighted degree is done by multiplying the one-hot vector with the weight matrix and subtracting the resulting column from the weighted degree vector (line 8).

Vertex deletion is done by masking, by assigning it an infinite value (line 9), which prevents it from being selected in subsequent iterations.

Edges incident to the removed vertex are deleted by applying an element-wise mask to the adjacency matrix (lines 10–12). Multiplying the complement of the one-hot vector with each row of the weight matrix zeros out the column corresponding to the minimum-degree vertex. It is not required to zero out the row, since its information is not required for any

Algorithm 7 Data Oblivious DSP for Dense Subgraph

Input: $\mathbf{W} \in \mathbb{Z}^{n \times n}$, where $n = |V|$ is the number of vertices
Output: md (maximum density), **sn** (max density subgraph nodes)

1: $\mathbf{AN}_{1:} \leftarrow \mathbf{1}_{1 \times n}$ // active nodes indicator
2: $\mathbf{wd} \leftarrow \mathbf{W} \cdot \mathbf{1}_{1 \times n}$ // weighted degree vector
3: $md \leftarrow \left(\sum_{i=1}^{n} \mathbf{wd}_i\right) / (2n)$ // current density
4: $t^\star \leftarrow 1$
5: **for** $i = 1$ to $n - 2$ **do**
6: $\quad k \leftarrow \arg\min(\mathbf{wd})$
7: $\quad \mathbf{v} \leftarrow$ one-hot vector with $\mathbf{v}_k = 1$
8: $\quad \mathbf{wd} \leftarrow \mathbf{wd} - \mathbf{W} \cdot \mathbf{v}^T$
9: $\quad \mathbf{wd} \leftarrow \mathbf{wd} + \infty\mathbf{v}$
10: $\quad$ **for** $j = 1$ to n **in parallel do**
11: $\quad\quad \mathbf{W}_{j:} \leftarrow \mathbf{W}_{j:} \odot (1 - \mathbf{v}^T)$
12: $\quad$ **end for**
13: $\quad \mathbf{AN}_{i+1:} \leftarrow \mathbf{AN}_{i:} - \mathbf{v}$
14: $\quad d \leftarrow (\mathbf{AN}_{i+1:} \cdot \mathbf{wd})/(2(n-i))$
15: $\quad$ **if** $d > md$ **then**
16: $\quad\quad md \leftarrow d$
17: $\quad\quad t^\star \leftarrow i$
18: $\quad$ **end if**
19: **end for**
20: **return** $md, \mathbf{AN}_{t^*:}$

calculation in subsequent iterations.

The set of active vertices is updated by subtracting the one-hot vector from the active node indicator vector (line 13). After each iteration, the density of the remaining graph is computed (line 14), and the iteration at which the maximum density observed so far is updated (lines 15–17). Since the sum of weighted degrees equals twice the total edge weight in an undirected graph, the density of the remaining graph can be computed from the active-node indicator and the updated weighted-degree vector.

Privacy-preserving DSP work and communication based on the adjacency matrix are $O(|V|^3)$. The round complexity of the dense graph DSP computation can be implemented with $O(|V| \log |V|)$ rounds, each iteration involves an oblivious argmin requiring $O(|V|)$ comparisons but the number of sequential comparisons is $O(\log |V|)$. Matrix-vector operations are parallelizable and contribute $O(1)$ rounds per iteration. PICCO [36] implementation for

the dense-graph DSP computation is provided in Appendix A.1.

4.2 Data-Oblivious Densest Subgraph for Sparse Graphs

The graph is represented in the form of a list $\mathcal{G}$, where each graph element (node or edge) is a tuple with fields

$$\mathcal{G}_i = (\texttt{src}, \texttt{dst}, \texttt{isV}, \texttt{isActv}, S, S', R, \texttt{data}),$$

where each field contains:

$$\texttt{data} = (\texttt{wDeg}, \texttt{edgWt}), \qquad S = (S_0, S_1), \qquad S' = (S'_0, S'_1), \qquad R = (R_0, R_1).$$

If $\mathcal{G}_i$ represents an edge, then `isV` is initialized to `False`. In this case, `src` contains the source node and `dst` contains the destination node. If $\mathcal{G}_i$ represents a vertex, then `isV` is initialized to `True`, and both `src` and `dst` store the vertex number.

The field `data` contains two values: `wDeg` and `edgWt`. If $\mathcal{G}_i$ represents an edge, then `edgWt` is initialized to the weight of that edge, and `wDeg` is initialized to zero. If $\mathcal{G}_i$ represents a vertex, then `wDeg` stores the weighted degree of that vertex. This value is computed for each vertex before the algorithm begins.

The field S indicates whether a particular vertex is active or not; this is the information that is propagated during the algorithm. Initially, for every vertex entry, $\mathcal{G}_i.S$ is set to $(1, 1)$, and for every edge entry, $\mathcal{G}_i.S$ is set to $(0, 0)$.

The field S' is an intermediate variable, and R stores the information received after propagation. For all elements in the list $\mathcal{G}$, both $\mathcal{G}_i.S'$ and $\mathcal{G}_i.R$ are initialized to $(0, 0)$.

Algorithm 8 gives the full data-oblivious peeling procedure for the sparse graph representation. The algorithm operates on the graph list $\mathcal{G}$ and maintains both vertex and edge entries throughout the computation. In each iteration, it removes the active vertex with minimum weighted degree, propagates the updated activity information through the graph

using optimized scatter and gather style operations, and then updates the edge weights and vertex weighted degrees. The key idea is that every iteration performs the same fixed se-

Algorithm 8 Densest Subgraph peeling with propagation

$N \leftarrow |\mathcal{G}|$
$md \leftarrow \sum_{i=0}^{|V|-1} \mathcal{G}_i.\texttt{data.wDeg}/|V|$
for $k = 1$ to $|V| - 2$ **do**
  select min weighted-degree active vertex
  $i^* \leftarrow \arg\min_{\substack{\mathcal{G}_i.\texttt{isV}=1 \\ \mathcal{G}_i.\texttt{isActv}=1}} \mathcal{G}_i.\texttt{data.wDeg}$
  deactivate vertex
  $\mathcal{G}_{i^*}.\texttt{isActv} \leftarrow 0$
  $\mathcal{G}_{i^*}.S \leftarrow (0, 0)$
  propagate S'_0 and compute R_0
  switch to vertex ordering
  $\mathcal{G}_0.S'_0 \leftarrow \mathcal{G}_0.S_0$
  for $i \leftarrow 1$ to $|V| - 1$ **do**
    $\mathcal{G}_i.S'_0 \leftarrow \mathcal{G}_i.S_0 - \mathcal{G}_{i-1}.S_0$
  end for
  switch to source ordering
  $tmp \leftarrow 0$
  for $i \leftarrow 0$ to $N - 1$ **do**
    $\mathcal{G}_i.R_0 \leftarrow tmp$
    $tmp \leftarrow tmp + \mathcal{G}_i.S'_0$
  end for
  propagate S'_1 and compute R_1
  switch to reverse vertex ordering
  $\mathcal{G}_0.S'_1 \leftarrow \mathcal{G}_0.S_1$
  for $i \leftarrow 1$ to $|V| - 1$ **do**
    $\mathcal{G}_i.S'_1 \leftarrow \mathcal{G}_i.S_1 - \mathcal{G}_{i-1}.S_1$
  end for
  switch to destination ordering
  $tmp \leftarrow 0$
  for $i \leftarrow N - 1$ to 0 **do**
    $\mathcal{G}_i.R_1 \leftarrow tmp$
    $tmp \leftarrow tmp + \mathcal{G}_i.S'_1$
  end for
  update edge weights using R_0, R_1
  for $i \leftarrow 0$ to $N - 1$ **in parallel do**
    $\mathcal{G}_i.\texttt{data.edgWt} \leftarrow \mathcal{G}_i.R_0 \times \mathcal{G}_i.R_1 \times \mathcal{G}_i.\texttt{data.edgWt}$
  end for
  accumulate the degree
  $tmp \leftarrow 0$

```
39:    for i ← 0 to N − 1 do
40:      G_i.data.wDeg ← tmp + G_i.data.edgWt
41:      tmp ← G_i.data.wDeg
42:    end for
43:    switch to vertex ordering
44:    for i ← 1 to |V| − 1 do
45:      G_i.data.wDeg ← G_i.data.wDeg − G_{i−1}.data.wDeg
46:    end for
47:    recompute weighted degrees and density
48:    wd ← Σ_{i=0}^{|V|−1} G_i.data.wDeg
49:    if wd/(|V| − k) > md then
50:      md ← wd/(|V| − k)
51:    end if
52:  end for
```

quence of sorting, scan, and arithmetic operations, independent of the input data. The main steps of one peeling iteration of Algorithm 8 are summarized below.

Peeling Steps

1. Select the active vertex with minimum weighted degree (lines 4–5).

2. Deactivate the selected vertex by setting $S \leftarrow (0, 0)$ (lines 6–8).

3. **Forward propagation:**

   (a) Switch to vertex ordering and compute the backward difference S'_0 (lines 10–14).

   (b) Switch to source ordering and compute R_0 using prefix sum of S'_0 (lines 15–20).

4. **Reverse propagation:**

   (a) Switch to reverse vertex ordering and compute the backward difference S'_1 (lines 22–26).

   (b) Switch to destination ordering and compute R_1 using prefix sum of S'_1 (lines 27–32).

5. Update edge weights using R^0 and R^1 (lines 33–36).

6. **Recompute weighted degrees:**

   (a) Accumulate edge weights to obtain cumulative weighted degrees (lines 37–42).

   (b) Switch to vertex ordering and compute vertex weighted degrees via backward difference (lines 43–46).

7. Update density and continue (lines 47–51).

The transition permutations between vertex, source, and destination orderings are precomputed once at a cost of $O(|\mathcal{G}| \log |\mathcal{G}|)$, and reused across iterations. The DSP computation is therefore $O(|\mathcal{G}| \cdot |V| + |\mathcal{G}| \log |\mathcal{G}|)$ work. The round complexity is $O(|V| \log |V|)$, as the scatter and gather steps use only additions and contribute $O(1)$ rounds, the edge weight update is performed in parallel across all edges contributing $O(1)$ rounds, and the oblivious argmin computation at each iteration involves an oblivious argmin requiring $O(\log |V|)$ rounds. PICCO [36] implementation of this sparse-graph DSP computation is provided in Appendix A.2.

4.3 Oblivious AntiBenford Subgraph

We now present a privacy-preserving version of the AntiBenford subgraph computation for sparse graphs by combining the methods from the previous sections. The full procedure is Algorithm 9. As in the non-secure AntiBenford method from Section 3.1, the transaction graph undergoes the same preprocessing to obtain the digit counts for each vertex. The graph is represented as a list $\mathcal{G}$, where each element corresponds either to a vertex or to an edge. If an element corresponds to a vertex, then its data field stores the digit-count vector $\mathbf{x}$, where $\mathbf{x}_{u,d}$ denotes the number of transactions incident to vertex u whose leading digit is d. In the secure computation, these counts are represented as secret shares, so the protocol operates on protected values $[\mathbf{x}_d]$ throughout.

For each vertex u, we first compute the protected total number of incident transactions

$$[N_u] = \sum_{d=1}^{9} [\mathbf{x}_d],$$

and then compute the protected AntiBenford score

$$[s(u)] = \sum_{d=1}^{9} \frac{([\mathbf{x}_d] - [N_u]p_d)^2}{[N_u]p_d},$$

where

$$p_d = \log_{10}(1 + 1/d)$$

is the public Benford probability for digit d.

After computing the protected score $[s(u)]$ for each vertex, we construct protected edge weights for the transformed graph. This is done by propagating vertex scores to incident edges using the scatter primitive in both source and destination orderings, as in Algorithm 5. Thus, for each edge $e = (u, v)$, the edge obtains the protected endpoint scores $[s(u)]$ and $[s(v)]$, from which we compute the protected edge weight

$$[w'(u, v)] = \sqrt{[s(u)] \cdot [s(v)]}.$$

The resulting protected weighted graph is then given as input to the privacy-preserving densest subgraph algorithm from the previous subsection, namely Algorithm 8. Before running the DSP computation, the algorithm computes the baseline statistic $\psi(V)$ for the whole graph using the same ψ function as in the non-secure AntiBenford method. This value serves as the reference point for determining whether the subgraph returned by the densest-subgraph computation is statistically significant.

The DSP computation updates the protected activity field `isActv`, setting it to 0 for vertices that are removed during the peeling process and leaving it equal to 1 for vertices that remain in the subgraph. From this field, the protected indicator vector an is computed

Algorithm 9 Oblivious AntiBenford Subgraph

Input: Graph list $\mathcal{G}$ in vertex ordering and protected digit-count vector $\mathbf{x}_i$ for each vertex i
Output: Protected suspicious vertex indicator $[\mathbf{S}]$

for $i = 1$ to $|V|$ in parallel **do**
  $[N_i] \leftarrow \sum_{d=1}^{9} [\mathbf{x}_{i,d}]$
  $[\mathcal{G}_i.s] \leftarrow \sum_{d=1}^{9} \frac{([\mathbf{x}_{i,d}] - [N_i]p_d)^2}{[N_i]p_d}$
end for
$[\psi(V)] = \left(\sum_{d=1}^{9} \frac{\left(\sum_{i=1}^{|V|} [\mathcal{G}_i.\mathbf{x}_d] - (p_d \cdot |V|)\right)^2}{p_d \cdot |V|} \right) / |V|$
Run Scatter-Graphiti in source ordering to propagate protected source scores to edges and store as $\mathcal{G}_i.s(u)$
Run Scatter-Graphiti in destination ordering to propagate protected destination scores to edges and store as $\mathcal{G}_i.s(v)$
for $i = 1$ to N in parallel **do**
  $[G_i.w] \leftarrow \sqrt{[\mathcal{G}_i.s(u)] \cdot [\mathcal{G}_i.s(v)]} \cdot (1 - \mathcal{G}_i.\mathsf{isV})$
end for
$[\mathcal{G}'] \leftarrow$ Oblivious-DSP$(\mathcal{G})$
ensure vertex order
for $i \in |\mathcal{G}'|$ **do**
  $[\mathbf{an}_i] = [\mathcal{G}'_i.\mathsf{isV}] \cdot [\mathcal{G}'_i.\mathsf{isActv}]$
end for
$[|S|] = \sum_i \mathbf{an}_i$
$[\psi(S)] = \left(\sum_{d=1}^{9} \frac{\left(\sum_{i=1}^{|V|} [\mathbf{an}_i] \cdot [\mathbf{x}_{i,d}] - (p_d \cdot |S|)\right)^2}{p_d \cdot |S|} \right) / |S|$
Open $\mathsf{found} = [\psi(S)] > c[\psi(V)]$
if found **then**
  Open $[\mathbf{an}_i] \cdot [\mathcal{G}'_i.\mathsf{src}]$
  return every non-zero value
end if

for the selected vertices. Using this indicator vector, $\psi(S)$ is computed for the subgraph and compared with the previously computed baseline value $\psi(V)$. Finally, the acceptance bit is opened. If the subgraph is statistically significant, the identifiers of the suspicious vertices are opened. The overall complexity of this algorithm is dominated by the DSP computation having the same communication and round complexity.

Appendix A provides PICCO [36] implementations for the dense AntiBenford subgraph computation and the sparse densest-subgraph computation.

Reference

[1] Aydin Abadi et al. "Starlit: Privacy-preserving federated learning to enhance financial fraud detection". In: *International Conference on Federated Learning Technologies and Applications (FLTA)*. IEEE. 2025, pp. 126–133.

[2] Galen Andrew et al. "Differentially private learning with adaptive clipping". In: *Advances in neural information processing systems* 34 (2021), pp. 17455–17466.

[3] Sunpreet Arora et al. "Privacy-preserving financial anomaly detection via federated learning & multi-party computation". In: *2024 annual computer security applications conference workshops (ACSAC workshops)*. IEEE. 2024, pp. 270–279.

[4] Alejandro Correa Bahnsen, Djamila Aouada, and Björn Ottersten. "Example-dependent cost-sensitive decision trees". In: *Expert Systems with Applications* 42.19 (2015), pp. 6609–6619.

[5] Dan Bogdanov, Sven Laur, and Jan Willemson. "Sharemind: A framework for fast privacy-preserving computations". In: *European Symposium on Research in Computer Security*. Springer. 2008, pp. 192–206.

[6] Dan Bogdanov, Riivo Talviste, and Jan Willemson. "Deploying Secure Multi-Party Computation for Financial Data Analysis: (Short Paper)". In: *International Conference on Financial Cryptography and Data Security*. Springer. 2012, pp. 57–64.

[7] Digvijay Boob et al. "Flowless: Extracting densest subgraphs without flow computations". In: *Proceedings of The Web Conference 2020*. 2020, pp. 573–583.

[8] Fabian Braun et al. "Improving card fraud detection through suspicious pattern discovery". In: *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*. Springer. 2017, pp. 181–190.

[9] David Byrd and Antigoni Polychroniadou. "Differentially private secure multi-party computation for federated learning in financial applications". In: *ACM International Conference on AI in Finance*. 2020, pp. 1–9.

[10] Rémi Canillas et al. “Exploratory study of privacy preserving fraud detection”. In: *International Middleware Conference Industry*. 2018, pp. 25–31.

[11] Mário Cardoso, Pedro Saleiro, and Pedro Bizarro. “Laundrograph: Self-supervised graph representation learning for anti-money laundering”. In: *Proceedings of the third ACM international conference on AI in finance*. 2022, pp. 130–138.

[12] Moses Charikar. “Greedy approximation algorithms for finding dense components in a graph”. In: *International workshop on approximation algorithms for combinatorial optimization*. Springer. 2000, pp. 84–95.

[13] Tianyi Chen and Charalampos Tsourakakis. “Antibenford Subgraphs: Unsupervised Anomaly detection in Financial Networks”. In: *ACM Conference on Knowledge Discovery and Data Mining (KDD)*. 2022, pp. 2762–2770.

[14] Ivan Damgård et al. “Confidential benchmarking based on multiparty computation”. In: *International Conference on Financial Cryptography and Data Security*. Springer. 2016, pp. 169–187.

[15] Cindy Durtschi, William Hillison, Carl Pacini, et al. “The effective use of Benford’s law to assist in detecting fraud in accounting data”. In: *Journal of forensic accounting* 5.1 (2004), pp. 17–34.

[16] Fabrianne Effendi and Anupam Chattopadhyay. “Privacy-preserving graph-based machine learning with fully homomorphic encryption for collaborative anti-money laundering”. In: *International Conference on Security, Privacy, and Applied Cryptography Engineering*. Springer. 2024, pp. 80–105.

[17] Marie Beth van Egmond et al. “Privacy-preserving anti-money laundering using secure multi-party computation”. In: *International Conference on Financial Cryptography and Data Security*. Springer. 2024, pp. 331–349.

[18] Jiani Fan et al. “Deep learning approaches for anti-money laundering on mobile transactions: Review, framework, and directions”. In: *IEEE Internet of Things Journal* (2026).

[19] Andrew V Goldberg. “Finding a maximum density subgraph”. In: (1984).

[20] Angela Samantha Maitland Irwin, Kim-Kwang Raymond Choo, and Lin Liu. “An analysis of money laundering and terrorism financing typologies”. In: *Journal of Money Laundering Control* 15.1 (2012), pp. 85–111.

[21] Marcel Keller. “MP-SPDZ: A versatile framework for multi-party computation”. In: *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*. 2020, pp. 1575–1590.

[22] Nishat Koti et al. “Graphiti: Secure graph computation made more scalable”. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2024, pp. 4017–4031.

[23] Tommaso Lanciano et al. “A survey on the densest subgraph problem and its variants”. In: *ACM Computing Surveys* 56.8 (2024), pp. 1–40.

[24] Xiangfeng Li et al. “FlowScope: Spotting Money Laundering Based on Graphs”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. 04. 2020, pp. 4731–4738.

[25] Shenghua Liu, Bryan Hooi, and Christos Faloutsos. “HoloScope: Topology-and-Spike Aware Fraud Detection”. In: *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. 2017, pp. 1539–1548. DOI: 10.1145/3132847.3133018.

[26] Xin Liu et al. “Collaborative fraud detection on large scale graph using secure multi-party computation”. In: *ACM International Conference on Information and Knowledge Management*. 2024, pp. 1473–1482.

[27] Mark Eshwar Lokanan and Kush Sharma. “Fraud prediction using machine learning: The case of investment advisors in Canada”. In: *Machine Learning with Applications* 8 (2022), p. 100269. ISSN: 2666-8270. DOI: https://doi.org/10.1016/j.mlwa.2022.100269.

[28] Brendan McMahan et al. “Learning Differentially Private Recurrent Language Models”. In: *International Conference on Learning Representations (ICLR)*. 2018. URL: https://openreview.net/pdf?id=BJ0hF1Z0b.

[29] Miguel Ângelo Lellis Moreira et al. “Exploratory analysis and implementation of machine learning techniques for predictive assessment of fraud in banking systems”. In: *Procedia Computer Science* 214 (2022). 9th International Conference on Information Technology and Quantitative Management, pp. 117–124. DOI: https://doi.org/10.1016/j.procs.2022.11.156.

[30] Kartik Nayak et al. “GraphSC: Parallel secure computation made easy”. In: *IEEE symposium on security and privacy*. IEEE. 2015, pp. 377–394.

[31] Jean-Claude Picard and Maurice Queyranne. “A network flow solution to some nonlinear 0-1 programming problems, with applications to graph theory”. In: *Networks* 12.2 (1982), pp. 141–159.

[32] Yusuf Sahin and Ekrem Duman. “Detecting credit card fraud by ANN and logistic regression”. In: *2011 international symposium on innovations in intelligent systems and applications*. IEEE. 2011, pp. 315–319.

[33] Alex Sangers et al. “Secure multiparty PageRank algorithm for collaborative fraud detection”. In: *International Conference on Financial Cryptography and Data Security*. Springer. 2019, pp. 605–623.

[34] Michele Starnini et al. “Smurf-based anti-money laundering in time-evolving transaction networks”. In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer. 2021, pp. 171–186.

[35] RESULTING FROM DRUG TRAFFICKING. “Estimating Illicit Financial Flows Resulting From Drug Trafficking and Other Transnational Organized Crimes”. In: (2011).

[36] Yihua Zhang, Aaron Steele, and Marina Blanton. “PICCO: a general-purpose compiler for private distributed computation”. In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 2013, pp. 813–826.

Appendix A

A.1 PICCO code for AntiBenford Subgraph on Dense Graphs

This appendix provides PICCO code implementation for the dense-graph version of the privacy-preserving AntiBenford subgraph computation. The program takes an adjacency-matrix representation of the transaction graph, computes Benford-based anomaly scores for each vertex, reweights edges with the anomaly score, and then runs the DSP for dense graphs and returns the nodes present in the densest subgraph. It also computes ψ for the full graph and the returned subgraph.

```
// Public parameter: number of nodes 16.
public int n=16;
public int dsp_nodes[16];
public int scale=1000;


private int w[n][n];
private int w2[n][n];
private int weightedDegree[n];
private int<1> oneHotMinimum[n];
private int graphcount[10];
```

```
public int reweightEdges();
public int DSP();
public int PSI_FULL();
public int PSI_SUBGRAPH();
public int chisq(public int k);
public int onehotminrecur(public int l,public int r);
public int multiplymerge(public int l,public int mid,public int r);

public int main(){
    smcinput(w,1,n,n);

    reweightEdges();
    smcoutput(w2,1,n,n);
    DSP();
    PSI_FULL();
    PSI_SUBGRAPH();

    return 1;

}

public int reweightEdges() {

    public int i;
    public int j;
```

```
public int k;
public int d;
private int sumOfCount[n];
private int count[10];
private int count2D[n][10];
private int<64> expected[n][10];
public int prob[10] = {0, 301, 176, 124, 96, 79, 66, 57, 51, 45};
private int<64> s[n];
private int<64> scaledCount[n][10];



for(i=0;i<n;i++){
    s[i]=0;
}


for(k=0;k<n;k++){
    // Compute digit counts for node k and accumulate graph-wide counts.
    for(d=0;d<10;d++){
        count[d]=0;
    }
    for(i=0;i<n;i++)[
        count[w[k][i]]=count[w[k][i]]+1;
    ]
    for(d=0;d<10;d++){
        graphcount[d] = graphcount[d] + count[d];
    }
    for(d=0;d<10;d++){
```

```
        count2D[k][d] =  count[d];
    }

    sumOfCount[k] = 0;
    for(d=1;d<10;d++){
        sumOfCount[k] = sumOfCount[k] + count2D[k][d];
    }
}
for(k=0;k<n;k++)[
    for(d=1;d<10;d++)[
        expected[k][d] = prob[d] * sumOfCount[k];
    ]
]
for(k=0;k<n;k++)[
    for(d=1;d<10;d++)[
        scaledCount[k][d] = count2D[k][d] * scale;
        scaledCount[k][d] = scaledCount[k][d] - expected[k][d];
        scaledCount[k][d] = scaledCount[k][d] * scaledCount[k][d];
        scaledCount[k][d] = scaledCount[k][d] / expected[k][d];
    ]
]

for(k=0;k<n;k++){
    for(j=0;j<10;j++){
        s[k] = s[k] + scaledCount[k][j];
    }
}
```

```
for(k=0;k<n;k++)[
    s[k] = s[k]/scale;
]

// Approximate the square root of each node score with Newton's method.
public int prec = 3;
public int twos[n];
private int temp = 1;
for (i = 0; i < prec ; i ++){
    temp = temp * 10;
}
for(k=0;k<n;k++){
    for (i = 0; i < prec ; i ++){
        s[k] = s[k] * 100;
    }
}
for(k=0;k<n;k++){
    twos[k] = 2;
}
private int<64> kk[n];
private int tt[n];
private int<64> kk_squared[n];
private int<64> numerator[n];

for(k=0;k<n;k++){
```

```
    kk[k] = 1;
    tt[k] =1;
}
for (i = 0; i < prec + 10; i++) {
    for(k=0;k<n;k++)[
        tt[k] = 2 * kk[k];
        kk[k] = kk[k] * kk[k];
        kk[k] = kk[k] + s[k];
        kk[k] = kk[k] / tt[k];
    ]
}

for(k=0;k<n;k++)[
    s[k]  = kk[k]/temp;
]



for(i=0;i<n;i++)[
    for(j=i+1;j<n;j++)[
        w2[i][j] = s[i]*s[j];
    ]
]

for(i=0;i<n;i++){
    for(j=0;j<i;j++){
        w2[i][j] = w2[j][i];
    }
```

```
    }

    return 1;

}

public int multiplymerge(public int l,public int mid,public int r){
    int lmin=0;
    int rmin=0;
    public int i;
    for(i=l;i<=mid;i++){
        lmin = lmin + (weightedDegree[i]*oneHotMinimum[i]);
    }
    for(i=mid+1;i<=r;i++){
        rmin = rmin + (weightedDegree[i]*oneHotMinimum[i]);
    }
    int c = 0;
    if (lmin<rmin){
        c = 1;
    }

    for(i=l;i<=mid;i++){
        oneHotMinimum[i] = oneHotMinimum[i]*c;
    }
    for(i=mid+1;i<=r;i++){
        oneHotMinimum[i] = oneHotMinimum[i]*(1-c);
    }
```

```
    return 1;

}

public int onehotminrecur(public int l,public int r){
    if (l==r){
        return 1;
    }
    else{
        public int mid = (l + r) / 2;
        onehotminrecur(l,mid);
        onehotminrecur(mid+1,r);
        multiplymerge(l,mid,r);
    }
    return 1;

}

// Store the minimum weighted-degree node as a one-hot vector.
public void findMinimumOneHot() {
    public int i;
    public int j;
    public int k;

    for(i=0;i<n;i++){
        oneHotMinimum[i] = 1;
```

```
    }

    onehotminrecur(0,n-1);
}

public int DSP() {
    public int numberOFActiveNodes = n;
    public int iteration = 0;
    public int infinity = 999999999;
    public int temp_pub;
    public int i;
    public int j;
    public int k;

    private int temp;
    private int<64> sumWeightedEdges;
    private int<1> activeNodes[n][n];
    private int<64> density;
    private int<64> maxDensity=0;
    private int<1> invOneHotMinimum[n];
    private int columnSum[n];
    private int tempmatrix[n][n];
    private int condition;
    private int maxDensityNodesRow;
    private int tempArr[n];

    // Each row stores the active-node vector for one DSP iteration.
```

```
for(i=0; i<n; i++) {
    for(j=0; j<n; j++) {
        activeNodes[i][j]=1;
    }
}
for(i=0; i<n; i++) {
    oneHotMinimum[i] = 0;
}

// Compute the initial weighted degrees and total edge weight.
sumWeightedEdges = 0;
for(i=0; i<n; i++) {
    weightedDegree[i] = 0;
    for(j=0;j<n;j++){
        weightedDegree[i] = weightedDegree[i] + w2[i][j];
    }
}

sumWeightedEdges=0;
for(i=0;i<n;i++){
    sumWeightedEdges = sumWeightedEdges + weightedDegree[i];
}
sumWeightedEdges = sumWeightedEdges/2;

density = sumWeightedEdges*100/numberOFActiveNodes*100;
maxDensity = density;
```

```
iteration = 0;
maxDensityNodesRow = iteration;

while(numberOFActiveNodes>2){
    iteration++;

    // Remove the minimum weighted-degree active node.
    findMinimumOneHot();

    for (i = 0; i < n; i++) [
        for (j = 0; j < n; j++) [
            tempmatrix[i][j] = w2[i][j] * oneHotMinimum[i] ;
        ]
    ]
    for(j=0;j<n;j++){
        columnSum[j] = 0;
        for(i=0;i<n;i++){
            columnSum[j] = columnSum[j] + tempmatrix[i][j];
        }
    }


    for(i = 0; i < n; i++) {
        weightedDegree[i] = weightedDegree[i] - columnSum[i];
        weightedDegree[i] = weightedDegree[i] + (oneHotMinimum[i] * infinity);
        sumWeightedEdges = sumWeightedEdges- columnSum[i];
    }
```

```
for(i = 0; i < n; i++) {
    invOneHotMinimum[i] = !oneHotMinimum[i];
}

for(i = 0; i < n; i++) [
    for(j=0;j<n;j++)[
        w2[i][j] = w2[i][j]*invOneHotMinimum[i];
    ]
]



// Record the active-node vector after this removal.
for(i=0; i<n; i++){
    activeNodes[iteration][i] = activeNodes[iteration-1][i] - oneHotMinimum[i];
}
numberOFActiveNodes = numberOFActiveNodes - 1;

// Track the active-node set with maximum density.
density = sumWeightedEdges*100/numberOFActiveNodes*100;



if (density>=maxDensity){
    maxDensity = density;
    maxDensityNodesRow = iteration;
}
```

```
    }

    smcoutput(maxDensityNodesRow,1);
    smcoutput(activeNodes,1,n,n);
    smcoutput(maxDensity,1);
    temp_pub = smcopen(maxDensityNodesRow);
    smcoutput(activeNodes[temp_pub],1,n);
    for(i=0;i<n;i++){
        dsp_nodes[i] = smcopen(activeNodes[temp_pub][i]);
    }
    smcoutput(dsp_nodes,1,n);

    return 1;
}

public int PSI_FULL() {
    private int sumOfCount;
    public int prob[10] = {0, 301, 176, 124, 96, 79, 66, 57, 51, 45};
    private int<64> psi_full;
    public int i;
    public int j;
    public int k;
    private int<64> expected[10];
    private int<64> scaledCount[10];

    sumOfCount = 0;
    for(j=1;j<10;j++){
```

```
        sumOfCount = sumOfCount + graphcount[j];
    }

    for(j=1;j<10;j++)[
        expected[j] = prob[j] * sumOfCount;
    ]



    for(j=1;j<10;j++)[
        scaledCount[j] = graphcount[j] * scale;
        scaledCount[j] = scaledCount[j] - expected[j];
        scaledCount[j] = scaledCount[j] * scaledCount[j];
        scaledCount[j] = scaledCount[j] / expected[j];
    ]

    psi_full=0;
    for(j=1;j<10;j++){
        psi_full = psi_full + scaledCount[j];
    }

    psi_full = psi_full/sumOfCount;
    smcoutput(psi_full,1);

    return 1;

}
```

```
public int PSI_SUBGRAPH() {
    private int count[10];
    private int sumOfCount;
    public int prob[10] = {0, 301, 176, 124, 96, 79, 66, 57, 51, 45};
    private int<64> psi_subgraph;
    public int i;
    public int j;
    public int k;
    private int<64> expected[10];
    private int<64> scaledCount[10];


    for(j=0;j<10;j++){
        count[j]=0;
    }


    for(k=0;k<n;k++)[
        if(dsp_nodes[k] == 1){
            for(i=0;i<n;i++)[
                if(dsp_nodes[i] == 1){
                    count[w[k][i]] = count[w[k][i]] + 1;
                }
            ]
        }
    ]
    sumOfCount = 0;
```

```
    for(j=1;j<10;j++){
        sumOfCount = sumOfCount + count[j];
    }

    for(j=1;j<10;j++)[
        expected[j] = prob[j] * sumOfCount;
    ]

    for(j=1;j<10;j++)[
        scaledCount[j] = count[j] * scale;
        scaledCount[j] = scaledCount[j] - expected[j];
        scaledCount[j] = scaledCount[j] * scaledCount[j];
        scaledCount[j] = scaledCount[j] / expected[j];
    ]

    psi_subgraph=0;
    for(j=1;j<10;j++){
        psi_subgraph = psi_subgraph + scaledCount[j];
    }

    psi_subgraph = psi_subgraph/sumOfCount;
    smcoutput(psi_subgraph,1);
    return 1;

}
```

A.2 PICCO Code for Densest Subgraph Computation on Sparse Graphs

This appendix provides PICCO prototype code for the sparse-graph version of the privacy-preserving densest-subgraph computation.

```
// n = |V+E|
public int n = 32;
public int vertexCount = 6;

// Parallel-array representation of a graph cell.
private int data[n];
private int source[n];
private int destination[n];
private int<1> isVertex[n];
private int<1> isEdge[n];
private int<1> isActive[n];
private int sortKey[n];
private int perm[n];
private int sourceTag[n];
private int destinationTag[n];

public void applyGraphPerm() {
    applyPerm(data, perm, n);
    applyPerm(source, perm, n);
    applyPerm(destination, perm, n);
    applyPerm(isVertex, perm, n);
```

```
    applyPerm(isEdge, perm, n);
    applyPerm(isActive, perm, n);
    applyPerm(sourceTag, perm, n);
    applyPerm(destinationTag, perm, n);
}

// key = (source, not isVertex, destination)
public void sourceOrderer() {
    public int i;

    for (i = 0; i < n; i++) {
        // Encode tuple lexicographically: (source, 1-isVertex, destination)
        sortKey[i] = source[i] * 10000 + (1 - isVertex[i]) * 100 + destination[i];
    }

    generatePerm(sortKey, perm, n);//bitonic sort
    applyGraphPerm();

    for (i = 0; i < n; i++) {
        sourceTag[i] = i;
    }
}

// key = (destination, isVertex, source)
public void destinationOrderer() {
    public int i;
```

```
    for (i = 0; i < n; i++) {
        // Encode tuple lexicographically: (destination, isVertex, source)
        sortKey[i] = destination[i] * 10000 + isVertex[i] * 100 + source[i];
    }

    generatePerm(sortKey, perm, n); //bitonic sort
    applyGraphPerm();

    for (i = 0; i < n; i++) {
        destinationTag[i] = i;
    }
}


public int main() {
    public int i;
    public int j;
    public int ptmp;

    public int activeNodes;
    public int iteration;
    private int scanHasSelection;
    private int sumOfEdgeWeights;
    private int density;
    private int bestDensity;
    private int accumulator;
    private int minWeightedDegree;
```

```
private int indexToDecativate;
private int bestIsActive[n];

private int tmp;
private int t1,t2,t3,t4,t5,t6,t7,t8,t9;

private int weightedDegree[n];

// Input all cell fields as parallel arrays.
smcinput(data, 1, n);
smcinput(source, 1, n);
smcinput(destination, 1, n);
smcinput(isVertex, 1, n);
smcinput(isEdge, 1, n);
smcinput(isActive, 1, n);

for (i = 0; i < n; i++) {
    sourceTag[i] = i;
    destinationTag[i] = i;
}

sourceOrderer();
destinationOrderer();

destinationOrderer();
tmp = 0;
for (i = 0; i < n; i++) {
```

```
    t1 = data[i] * isEdge[i];
    tmp = tmp + t1;
    weightedDegree[i] = tmp * isVertex[i];
    t2 = 1 - isVertex[i];
    tmp = tmp * t2;
}

sumOfEdgeWeights = 0;
for (i = 0; i < n; i++) {
    t1 = weightedDegree[i] * isVertex[i];
    sumOfEdgeWeights = sumOfEdgeWeights + t1;
}
sumOfEdgeWeights = sumOfEdgeWeights / 2;

activeNodes = vertexCount;
density = (sumOfEdgeWeights * 1000) / activeNodes;
bestDensity = density;

iteration = 0;
while (iteration < vertexCount - 2) {

    //finding and deativating the num degree vertex
    minWeightedDegree = 10000000;//infinity
    indexToDecativate = 0;
    private int index = 0;
    for (i = 0; i < n; i++) {
        t1 = isActive[i] * isVertex[i];
```

```
    t2 = 1 - t1;
    tmp = weightedDegree[i] * t1 + 10000000 * t2; //infinity*t2
    if (tmp < minWeightedDegree){
        minWeightedDegree = tmp;
        indexToDecativate = index;
    }
    index = index+1;
}
ptmp = smcopen(indexToDecativate);

isActive[ptmp] = 0;

//scatter the deactivation to the edges connected to the deactivated vertex
sourceOrderer();
tmp = 1;
for (i = 0; i < n; i++) {
    t1 = 1 - isVertex[i];
    t2 = tmp * t1;
    t3 = isActive[i] * isVertex[i];
    tmp = t2 + t3;

    t4 = 1 - isEdge[i];
    t5 = isEdge[i] * tmp;
    t6 = t4 + t5;
    isActive[i] = isActive[i] * t6;
}
```

```
//scatter in reverse direction to ensure all edges connected to the deactivated
destinationOrderer();
tmp = 1;
for (i = n - 1; i >= 0; i--) {
    t1 = 1 - isVertex[i];
    t2 = tmp * t1;
    t3 = isActive[i] * isVertex[i];
    tmp = t2 + t3;

    t4 = 1 - isEdge[i];
    t5 = isEdge[i] * tmp;
    t6 = t4 + t5;
    isActive[i] = isActive[i] * t6;
}

//gather
tmp = 0;
accumulator = 0;

for (i = 0; i < n; i++) {
    t1 = 1 - isActive[i];
    t2 = data[i] * t1;
    t3 = t2 * isEdge[i];
    tmp = tmp + t3;

    t4 = 1 - isVertex[i];
    t5 = weightedDegree[i] - tmp;
```

```
    t6 = weightedDegree[i] * t4;
    t7 = t5 * isVertex[i];
    weightedDegree[i] = t6 + t7;

    t8 = tmp * isVertex[i];
    accumulator = accumulator + t8;

    t9 = 1 - isVertex[i];
    tmp = tmp * t9;
}

//zero out removed edges using only arithmetic masking
for (i = 0; i < n; i++) {
    data[i] = data[i] * isEdge[i] * isActive[i];
}

activeNodes = activeNodes - 1;
sumOfEdgeWeights = sumOfEdgeWeights - (accumulator / 2);
density = (sumOfEdgeWeights * 1000) / activeNodes;

if (density > bestDensity) {
    bestDensity = density;
    for (i = 0; i < n; i++) {
        bestIsActive[i] = isActive[i] * isVertex[i] * source[i];
    }
}
iteration = iteration + 1;
```

```
    }


    smcoutput(data,1,n);
    smcoutput(source,1,n);
    smcoutput(destination,1,n);
    smcoutput(weightedDegree,1,n);
    smcoutput(isActive,1,n);
    smcoutput(bestIsActive, 1, n);
    smcoutput(bestDensity, 1);
    return 1;
}
```