Multilateral Trade Credit Set-off in MPC via Graph Anonymization and Network Simplex

Enrico Bottazzi[1], Chan Nam Ngo[1], and Masato Tsutsumi[2]

[1] Ethereum Foundation
{enrico,namncc}@pse.dev
[2] Waseda University
masato.11.soccer@ruri.waseda.jp

Abstract. Multilateral Trade Credit Set-off (MTCS) is a process run by a service provider that collects trade credit data (i.e. obligations from a firm to pay another firm) from a network of firms and detects cycles of debts that can be removed from the system. The process yields liquidity savings for the participants, who can discharge their debts without relying on expensive loans. We propose an MTCS protocol that protects firms' sensitive data, such as the obligation amount or the identity of the firms they trade with. Mathematically, this is analogous to solving the Minimum Cost Flow (MCF) problem over a graph of n firms, where the m edges are the obligations. State-of-the-art techniques for Secure MCF have an overall complexity of $O(n^{10} \log n)$ communication rounds, making it barely applicable even to small-scale instances. Our solution leverages novel secure techniques such as Graph Anonymization and Network Simplex to reduce the complexity of the MCF problem to $O(max(n, \log\log n + m))$ rounds of interaction per pivot operations in which $O(max(n^2, nm))$ comparisons and multiplications are performed. Experimental results show the tradeoff between privacy and optimality.

Keywords: Minimum Cost Flow · Multi-party Computation

1 Introduction

Multilateral Trade Credit Set-off (MTCS), also referred to as obligation clearing, is a process run by a service provider that collects trade credit data (i.e. obligations from a firm to pay another firm or IOUs) from a network of firms, detects cycles of debts that can be removed from the system and returns the updated obligations to the firms [22]. Removing cycles is beneficial since it reduces the aggregate indebtedness in the system, unlocks payment gridlocks, and yields liquidity savings for the participants, as demonstrated by Fleischman et al. [29].

Figure 1 illustrates, on the left, a trade credit network in which participants cannot repay their obligations due to insufficient liquidity. The system is brought to a halt, known as gridlock. On the right, the same network after MTCS: a cycle of debt is removed without requiring any expensive injection of external liquidity. A complete view of the trade credit network is required to perform MTCS. Note that the net balance of the firms, which is equal to the sum of their balance and credit minus debt, remains unchanged: the network is balanced.

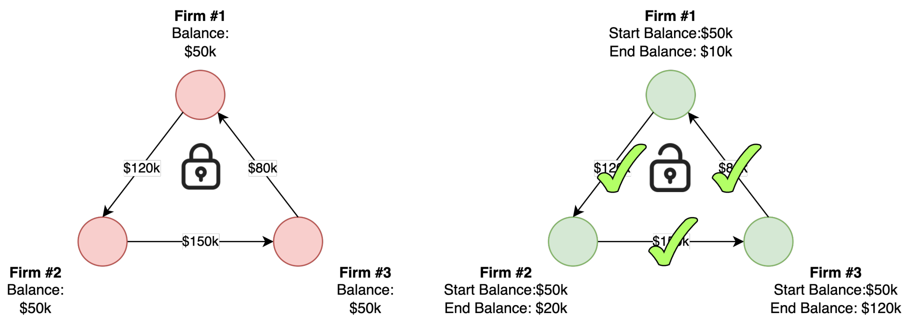

Fig. 1: Gridlock resolution in a trade credit network after MTCS

An analogous technique, commonly referred to as multilateral netting, is used daily by banks, which clear their credit/debt obligations between each other to decrease their chance of financial default [30]. By contrast, these methods are not available to the large majority of citizens and businesses, so they do not benefit the real economy. An exception is Slovenia, which, starting in 1991, implemented MTCS on a country-wide level. MTCS is run on a monthly basis by the government agency AJPES, which charges a percentage fee on the cleared amount [29]. The participation spiked in 2012, after the global financial crisis, with 14000 firms participating (8.3% of the total registered companies) and 683 million Euro (1.89% of the GDP) cleared. In 2022, the participation dropped to 1500 firms (0.2% of total), and 108 million Euro (0.7% of the GDP) cleared [41, 4]. Currently, MTCS requires an agency to access the firms' sensitive data. Fleischman and Dini [28] argue that a lack of trust of the firms towards such an agency might discourage their participation. Indeed Bogetoft et al. [16], concluded that ordinary people do care about data confidentiality, especially when money is involved and even more so if parties getting access to these data have conflicting interests with theirs, as they experimented with Danish farmers and sugar beet contracts trading.

Our research thesis is that firms would be more incentivized to engage in MTCS if their trade credit data were not revealed to an external agency. A higher participation rate would also increase the size of the network and the chance of detecting and removing debt cycles from it, yielding greater benefits for all the participants.

In the proposed privacy-preserving[3] solution, the agency is replaced by a set of parties with conflicting interests that perform the MTCS in a secure environment. The protocol operates in a clients-to-servers setting similar to Damgaard

[3] privacy-preserving and secure are used interchangeably

et al. [26]. The servers (also referred to as parties or computing parties) receive private inputs from the firms in a secret-shared format, perform the MTCS in Secure Multi-party Computation (MPC), and send back the shares of the updated obligations with each firm. In the proposed solution, the servers are managed by institutions with conflicting interests, such as the chamber of commerce, the tax authority, and a payment processor. One key intuition to scale the protocol to tens of thousands of firms is to accept that even a sub-optimal MTCS process, in which not all the debt cycles are removed, yields benefits for the participants.

The correctness of the algorithm execution and the privacy over trade credit data (i.e. amount, source, and destination of individual obligations) is guaranteed under the security assumptions of the underlying MPC protocol. Constraint correctness is satisfied if the net balance of any individual firm does not change and no novel obligations among firms are created. We demonstrate that clients can verify constraint correctness even with all the servers deviating from the protocol specification, making the protocol secure against malicious computing parties.

1.1 Related Work

Multilateral Trade Credit Set-off Fleischman et al. [29] provides a rigorous analysis of the benefits of MTCS for small and medium-sized enterprises (SMEs). Fleischman and Dini [28] provide the mathematical foundations for the MTCS problem, including its formulation and solution based on the Minimum Cost Flow (MCF) problem. Our work can be synthesized as an effort to efficiently replicate the algorithm proposed by Fleischman and Dini [28] in a secure environment.

Secure Multilateral Netting Various papers address the problem of secure multilateral netting for inter-bank payments, which, similarly to MTCS, belongs to the category of liquidity saving mechanisms (LSMs). Cao et al. [19] and Galal and Youssef [31] leverage zero-knowledge proofs to design a distributed protocol in which participants compute their local optimal settlement and iteratively submit it to a smart contract until convergence is found. Such solutions require all the participants to be online during the protocol. Additionally, none of them is able to provide full privacy over the payment source, destination, and amount. The solution proposed by Atapoor et al. [10] runs in a three-party MPC. Nevertheless, the algorithm is designed only to provide binary netting (one transaction is settled in full or not settled at all). Lastly, Agarwal et al. [2] introduce an MPC-based solution that achieves an optimal solution in terms of amounts netted by allowing partial settlement. Their algorithm is based on a secure linear programming solution based on Karmarkar method that has runtime complexity equal to $O(n^6)$ [1] where n is the number of participating banks. Toft [43] observes the inefficiency of the Karmarkar method in a secure environment. Although the multilateral netting optimization problem is technically analogous for banks and firms, none of the existing solutions designed for the banking system can efficiently scale to support a network of firms, which would increase the factor n

by at least two orders of magnitude. To the best of our knowledge, no literature discusses the problem of secure MTCS.

Secure Minimum Cost Flow Algorithms Aly and Van Vyne [6] propose a secure protocol to solve the MCF problem on a graph in a way that hides any information related to the nodes and the edges of the graph. To achieve that, the protocol operates on a complete directed graph; namely, all the possible $n \cdot (n-1)$ edges are assumed to exist. This comes at a cost, since the algorithm runs in $O(n^{10} \log n)$ rounds of interactions, where n is the number of nodes in the graph. Similarly, other authors [15, 5] assume complete graph representation to securely compute network flow operations. Further application of secure minimum cost flow can be found in single-commodity multi-market auctions [7], rebalancing on payment channel networks [11] and trade chain detection [45]. The MCF problem can also be interpreted as a linear programming problem. Toft [43] introduces a protocol to securely solve linear problems via the simplex method. Finally, Avarikioti et al. [11] recommend using secure MCF over linear programming algorithms to solve privacy-preserving rebalancing on payment channel networks because of their better efficiency.

1.2 Our Contributions

Our main contribution is a secure MTCS protocol. To achieve the desired level of scalability, we leverage two novel techniques:

- A secure graph anonymization protocol that allows the computing parties to perturb a graph by relabelling the nodes and performing random deletion and addition of edges
- A secure network simplex protocol, which, for each pivot operation, requires $O(max(n, \log\log n + m))$ rounds of interactions in which $O(max(n^2, nm))$ comparisons and multiplications have to be performed. Such protocol is used as a subroutine to solve the MCF protocol and improves the state-of-the-art solution proposed by Aly and Van Vyne [6] by several orders of magnitude.

In the proposed secure MTCS protocol, the servers first anonymize the graph, then open its viable and perturbed m edges, and, lastly, run the MCF protocol securely without accounting for a complete graph. Experimental results based on synthetic datasets show that a meaningful level of anonymity (74%, 81% and 86% of nodes with changed degree, respectively for each network) can be achieved with a limited loss in terms of clearing percentage (63%, 63 and 66% of the optimal).

1.3 Overview

In Section 2 we formalize the MTCS as in Fleischamn and Dini [28], whereas in Section 3, we lay the foundations for the graph-related algorithms we use in our system, namely the Minimum Cost Flow problem and the Network Simplex as well as the Network Anonymization protocols. Section 4 contains the cryptographic components we use. We introduce our novel Secure Graph Anonymization via SecureRelabel and SecureRandomAdd/Del in Section 5. We show how to make SecureNetworkSimplex in Section 6 and its application to Secure MTCS in Section 7. Section 8 contains an experimental evaluation of the tradeoff between privacy and optimality of the proposed solution. We conclude our work and sketch future directions in Section 9.

2 Multilateral Trade Credit Set-off

Following the mathematical formalization by Fleischman and Dini [28], in a trade credit network with n firms, the obligation matrix O is a square $(n \times n)$ matrix where the entry O_{ij} represents the amount that firm i owes to firm j. Since firms do not issue obligations to themselves, the matrix O has zeroes on the diagonal. The sum of the elements of the column array i and the sum of the elements of the row array i, represent, respectively, the total credit and the total debt of firm i. The net balance b_i of such a firm is defined by the difference between its total credit and total debt.

Performing MTCS in a trade credit network is equal to finding the obligation matrix O' such that its Grandsum function μ is minimum:

$$\min \mu(O') = \min \sum_{i=1}^{n} \sum_{j=1}^{n} O'_{ij} \tag{1}$$

subject to the constraints:

$$\sum_{k=1}^{n} O'_{ki} - \sum_{k=1}^{n} O'_{ik} = b_i \qquad \forall i \in n \tag{2}$$

$$0 \leq O'_{ij} \leq O_{ij} \qquad \forall i, j \in n \tag{3}$$

O' is the obligation state after MTCS, (1) is the objective function, (2) enforces that the net balance of any individual firm remains consistent, while (3) guarantees that no novel obligations among firms are created.

We introduce two notions of MTCS algorithm correctness:

– *Constraint correctness*, which is satisfied if (2) and (3) are met. Such correctness can be verified locally by each firm i by comparing their state before and after MTCS
– *Optimality correctness*, which is satisfied if constraint correctness is met and (1) is met. Such correctness requires a global view of the original obligation state O and updated state O' to be verified

As long as constraint correctness is met, a firm can not be worse off after MTCS than when it started: in the worst case, the updated obligations are equal to the initial ones, so nothing changed.

Fleischman and Dini [28] suggest interpreting such optimization problem as a Minimum Cost Flow (MCF) problem over a directed graph where the nodes represent firms, the edges represent the obligations, and the cost and the capacity of any edge are, respectively, set to 1 and the obligation amount. The optimization problem can also be solved via Linear Programming [2, 23]. Nevertheless, interpreting the problem as a network flow problem provides more efficient strongly polynomial time algorithms [44, 3]. Király and Kovács [37] list most of the protocols used for MCF. The MCF-based solution, while not unique, is proven to yield the maximum possible amount of liquidity saved for the firms in the trade credit network [28]. This is the same result, in terms of optimality, achieved by Agarwal et al. using Linear Programming [2]. Our secure protocol is based on the formalization of [28].

Fleischman and Dini [28] extend the concept by introducing a liquidity source node to the obligation network that acts as a store of funds for firms, similar to a bank. Such an extension does not require any modification of the mathematical primitives; therefore, for the sake of simplicity, we do not consider it.

3 Graph Theory Preliminaries

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a directed graph such that $\mathcal{V}$ is the set of nodes and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is the set of edges such that $|\mathcal{V}| = n$ and $|\mathcal{E}| = m$. An edge can be represented as an ordered pair (i, j) where i is the index of the source node and j is the index of the destination node. Each edge (i, j) has non-negative capacity u_{ij} and cost c_{ij} associated with that. Each node i has a signed supply balance b_i associated with that. A node is defined as a *demand* node if its balance is negative or as a *supply* node if its balance is positive. The network is balanced if $\sum_{i\in\mathcal{V}} b_i = 0$. A cycle is a special path $(i_1, j_1), \ldots, (i_k, j_k) \in \mathcal{E}$ in which the edges are contiguous and no vertex is visited more than once except for $i_1 = j_k$.

3.1 Minimum Cost Flow Problem

The MCF problem involves finding the least expensive way to send a specified amount of flow from supply nodes to demand nodes through a directed network of edges with capacity constraints and costs associated with them. The optimization goal is to minimize the total transportation cost (4) while satisfying flow conservation constraints (5) over the nodes and capacity constraints (6) over the edges.

The MCF problem, where x_{ij} indicates the flow over the edge (i, j), is defined as follows, according to the definition of Király and Kovács [37]:

$$\min \sum_{(i,j)\in\mathcal{E}} c_{ij} x_{ij} \tag{4}$$

subject to the constraints:

$$\sum_{j:ij\in\mathcal{E}} x_{ij} - \sum_{j:ji\in\mathcal{E}} x_{ji} = b_i \quad \forall i \in \mathcal{V}, \tag{5}$$

$$0 \leq x_{ij} \leq u_{ij} \qquad \forall (i,j) \in \mathcal{E} \tag{6}$$

The MCF problem is equal to the optimization problem defined in §2 where x_{ij} is equal to O'_{ij}. Intuitively, from the MCF viewpoint, cycles are an inefficient way to carry flow through the network because they lengthen the path and increase the cost. Therefore they are excluded from the solution x.

Multiple algorithms have been devised to solve the MCF problem. The Minimum Mean Cycle-Cancelling (MMCC) [32, 33] algorithm, used in the secure implementation of Aly and Van Vyne [6], has a runtime complexity (in its non-secure/plain version) equal to $O(n^2m^2 \min\{\log(nC), m\log(n)\})$ where C is the largest edge cost in the network. The primal Network Simplex (NS) [39] has a worst-case running time complexity of $O(n^2m \min\{\log(nC), m\log(n)\})$, Results from Király and Kovács [37] show that the NS is the most efficient solver for the MCF protocol, being over four orders of magnitude faster than MMCC with a practical runtime complexity of $O(nm)$.

3.2 Network Simplex

The NS algorithm is a specialization of the general linear programming simplex method that leverages the MCF problem's network structure to achieve greater efficiency. The NS Algorithm operates by partitioning the edge set $\mathcal{E}$ into a spanning tree structure represented by the sets $\mathcal{T}$, $\mathcal{L}$ and $\mathcal{U}$ such that each edge in $\mathcal{L}$ has flow set to 0, each edge in $\mathcal{U}$ has flow saturating its capacity and the edges in $\mathcal{T}$ form an undirected spanning tree with unrestricted flow. A spanning tree represents a special subgraph that has exactly $n-1$ edges that connect all nodes without creating any cycles. The algorithm maintains a feasible spanning tree solution (i.e that satisfies the constraints of the MCF problem) at each step. It gradually converges towards the optimal solution by successive transformations, known as pivots, that reduce the total cost of the flow. More specifically, the optimality test is done by computing the reduced cost c^{π}_{ij} of each non-spanning tree edge from the node potentials π such that $c^{\pi}_{ij} = c_{ij} - \pi_i + \pi_j$. An optimal spanning tree structure satisfies the following two conditions:

$$\begin{aligned} c^{\pi}_{ij} &\geq 0 \quad \text{for every edge}(i,j) \in \mathcal{L}, \\ c^{\pi}_{ij} &\leq 0 \quad \text{for every edge}(i,j) \in \mathcal{U}. \end{aligned}$$

At each pivot iteration, an edge violating the optimality condition is added to the spanning tree $\mathcal{T}$ to form a negative-cost cycle. We augment the flow along the cycle until an edge reaches its upper or lower capacity bound. This edge is removed from the tree, canceling the cycle, and placed either in $\mathcal{L}$ or $\mathcal{U}$.

While moving from one spanning tree solution to the next, the algorithm always maintains the condition that the reduced cost of every edge (i, j) in the current spanning tree is equal to 0 (i.e. $c^{\pi}_{ij} = 0$)

Algorithm 1 Network Simplex

```
Input: graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
Output: minimum cost flow $x$ over the edges $\mathcal{E}$
1: Initialize a feasible structure $(\mathcal{T}, \mathcal{L}, \mathcal{U})$ with flow $x$ and node potentials $\pi$.
2: while some non-tree edge violates the optimality conditions do
3:     Find an edge $e_{in} \in \mathcal{L} \cup \mathcal{U}$ violating its optimality condition
4:     Add edge $e_{in}$ to the spanning tree and determine leaving edge $e_{out}$
5:     Update the spanning tree structure $(\mathcal{T}, \mathcal{L}, \mathcal{U})$ and the associated $x$ and $\pi$
6: end while
```

Given that MCF instances always have an optimal spanning tree solution [3], the NS algorithm exploits this property by focusing its search on the spanning tree solutions. Király and Kovács [37] illustrate the data structures that have been devised for efficient updates of the spanning tree structure, together with the different ways to find an initial feasible solution and entering edge selection policies. Ahuja et al. [3] provide a comprehensive overview of the algorithm, its analogies with the traditional simplex method and the leaving arc selection rule to avoid degeneracy (i.e. infinite looping) to guarantee finite termination.

3.3 Graph Anonymization

Graph Anonymization is a branch of research that studies techniques that a data owner can apply to their graph to anonymize the underlying data before publishing. Zhou et al. [46] and Casas-Roma [21] provide a comprehensive overview of the topic. Naively, one might think that relabelling the nodes' identifier would suffice. Backstrom et al. [12] demonstrated how this approach is vulnerable to targeted re-identification attacks from an adversary that knows the degree of a particular node. Overall, the goal of the study is to define perturbation algorithms to avoid such attacks while retaining the utility of the data as much as possible.

Randomized edge construction algorithms construct an anonymized graph $\mathcal{G}'$ from the original graph $\mathcal{G}$ through a sequence of ξ edge deletions followed by ξ edge insertions. In particular, Hey et al. [34] suggest a Random Add/Del algorithm such that the edges to be deleted are chosen uniformly at random from the set of edges of $\mathcal{G}$, while the edges to be inserted are chosen uniformly at random from the set of all non-existent edges of the interim graph. The nodes do not change, and the perturbation process, together with the perturbation parameter ξ, are assumed to be publicly known. As a result, an adversary that knows the degree of a target node from the original graph and tries to re-identify it in the perturbed published graph cannot easily exclude nodes from the candidate set, even if they do not match the structural characteristics of the target. Empirical results from Bonchi et al. [17] demonstrate how this Random Add/Del method is the best perturbation method to alter the degree sequence of the nodes in order to preserve users' privacy and hinder re-identification attacks. In particular, they observe that, on average, more than 90% of the nodes change their degree with 10% of perturbation ($\xi = 0.1m$). The algorithm is efficient, with a computational complexity of $O(n^2)$, and simple, which makes it a good candidate algorithm for building a secure graph anonymization protocol.

4 Cryptographic Preliminaries

4.1 Secure Multi-party Computation

Secure multiparty computation (MPC) allows parties to compute a function on their combined private input without disclosing it to each other [38]. To perform MPC, we require an arithmetic black-box $\mathcal{F}_{\mathsf{ABB}}$ that allows for secure integer computation of elements in a prime field $\mathbb{Z}_p$ or ring of the form $\mathbb{Z}_{2^k}$ [27]. The arithmetic modulus M must be sufficiently large such that no overflow occurs. We assume that the following basic functionalities are available: addition, multiplication, equality and inequality tests. More advanced functionalities are described in §4.2. Negative values such as $-a$ are represented as $M - a \in M$ such that addition and multiplication that involve negative values behave as expected. The concept of arithmetic black-box abstracts the algorithm design from the functionality specificities of the underlying MPC protocol, which can be chosen from different security assumptions [8, 24, 25, 35]. We use the traditional square brackets notation e.g. $[x]$, to represent secret shared elements in $\mathcal{F}_{\mathsf{ABB}}$.

Similarly to Aly and Van Vyne [6] and Toft [43], we measure the cost of our protocol by communication complexity and round complexity. Communication complexity is measured by the number of secure multiplications and secure comparisons performed, in which the former ones are more computationally efficient than the latter ones. Linear operations such as additions or multiplication by a public value are considered costless, because they can be performed locally without any communication. Round complexity is measured by the number of messages transmitted between the parties. As an example, an arithmetic circuit in which the multiplicative depth is one requires a single round of interaction to be computed, while a circuit in which the multiplicative depth is t requires t rounds of interactions. Informally, communication and round complexity affect, respectively, the required bandwidth and latency of a network. As a general rule of thumb for our protocol design, we try to reduce round complexity as much as possible.

4.2 Advanced Functionalities

This section defines more advanced functionalities that leverage the basic primitives available from the arithmetic black-box $\mathcal{F}_{\mathsf{ABB}}$.

Conditional Assignment Functionality denoted by $[c] \leftarrow_{[z]} [a] : [b]$, such that one of $[a]$ or $[b]$ is assigned to $[c]$ according to the bit $[z]$. More specifically, $[c] = [a]$ if $[z] = 1$ and $[c] = [b]$ otherwise and this can be performed with a single multiplication $c = b + z(a - b)$.

Conditional Exchange Functionality denoted as CondEx that securely swaps $[a]$ and $[b]$ if $[z] = 0$ and performs no swap if $[z] = 1$. Algorithm 2 describes the implementation of such functionality, which requires a single multiplication.

Algorithm 2 Secure Conditional Exchange

Input: binary expression $[z]$, values $[a]$ and $[b]$
Functionality: returns $[a']$ and $[b']$ such that $[a'] = [b]$ and $[b'] = [a]$ (swap) if $[z] = 0$ or $[a'] = [a]$ and $[b'] = [b]$ (no swap) if $[z] = 1$

```
1: $[d] \leftarrow (1 - [z]) \cdot ([b] - [a])$
2: $[a'] \leftarrow [a] + [d]$
3: $[b'] \leftarrow [b] - [d]$
```

Opening We invoke the function $z \leftarrow \mathsf{Open}([z])$ to open a secret shared value to the public and $\mathsf{Open}_{\mathsf{E}}([z], \mathrm{ID})$ to open a secret shared value to a specific subject identified by its ID.

Secure Shuffle We define a $\mathsf{Shuffle}$ protocol that allows to securely shuffle a vector based on a secret shared random permutation $[\pi]$ (which can be generated during the setup phase). We leverage the protocol proposed by Asharov et al. [9], designed specifically for replicated secret sharing, which requires each party to communicate $2n$ ring elements to the other parties, where n is the size of the input vector. Keller and Scholl [36] introduce a shuffling technique for malicious security secret-sharing-based schemes. Similarly, the $\mathsf{Unshuffle}$ protocol takes a securely shuffled vector as input, together with $[\pi]$, and reverses the shuffling operation. We extend the algorithm to support matrix shuffling $\mathsf{ShuffleMat}$, which requires $2n$ $\mathsf{Shuffle}$ over the n rows and the n columns of a matrix. $\mathsf{UnshuffleMat}$ is the corresponding unshuffling operation.

Compute the minimal of multiple values Given a secret shared array $[arr]$ the goal is to compute a secret share of the minimum value $[minval]$ together with a secret share of its index $[minidx]$. This operation can be done in a $O(\log \log k)$ rounds protocol using $O(k)$ comparisons and multiplications, as described by Toft [43], where k is the length of the array and k is assumed to be a power of two. We denote this functionality as Min.

4.3 Secret Shared Arrays

Secret shared arrays are defined using the square bracket notation $[arr]$ such that its elements are secret shared elements $[arr] = ([a_0], [a_1], \ldots, [a_{k-1}])$. The shortcut notation $[arr] \leftarrow [x]$ assigns the value $[x]$ to all the elements of the array, and it is often used during initialization. The notation $[arr]_i$ indexes a secret shared array at a publicly known index. More involved operations are required when the index is private. To index an array at a secret shared index $[arr]_{[j]}$, we first need to construct a binary array $[b]$ consisting of all 0's except for the j-th position. Then, indexing is the result of the dot product between the two arrays.

Algorithm 3 Secure Array Indexing with Private Index

```
Input: secret shared array $[arr]$ of length $k$, secret shared index $[j]$
Output: secret shared value $[v] = [arr]_{[j]}$
1: Initialize $[b]$ as array of length $k$ with $[b] \leftarrow 0$
2: for $i \leftarrow 0$ to $k - 1$ do
3:     $[b]_i \leftarrow ([j] = i)$
4: end for
5: $[v] \leftarrow \sum_{i=0}^{k-1} [b]_i \cdot [arr]_i$
```

A similar logic applies when performing an oblivious assignment $[arr]_{[j]} \leftarrow [x]$, which leverages a binary array $[b]$ and the conditional assignment operation.

Algorithm 4 Secure Array Assignment with Private Index

```
Input: secret shared array $[arr]$ of length $k$, secret shared index $[j]$, secret shared value $[x]$
Output: updated array $[arr]$ with $[arr]_{[j]} = [x]$
1: Initialize $[b]$ as array of length $k$ with $[b] \leftarrow 0$
2: for $i \leftarrow 0$ to $k - 1$ do
3:     $[b]_i \leftarrow ([j] = i)$
4: end for
5: for $i \leftarrow 0$ to $k - 1$ do
6:     $[arr]_i \leftarrow_{[b]_i} [x] : [arr]_i$
7: end for
```

Both the operation requires $O(k)$ comparisons and $O(k)$ multiplications, which can be performed in parallel in a single round of communication. We do not assume the secret shared index $[j]$ to be within the array range. Picking a non-existing secret shared index such as $[-1]$ returns $[0]$ in the first scenario and performs no update in the second.

5 Secure Graph Anonymization

Traditionally, graph anonymization is performed by a single party (e.g., the data owner) to sanitize the data before publishing. Secure graph anonymization applies the same techniques in a setting where the graph data is split between several parties. In this scenario, the parties are the adversaries that must access an anonymized version of the graph.

Starting from a secret-shared representation of a graph (all the edges are assumed to exist), the parties get to an anonymized version of the graph (only perturbed edges are assumed to exist) that, when opened, prevents any node re-identification attacks from an adversary that has background knowledge of node degrees. Parties can now perform any subsequent graph operation, such as network flow algorithms, much more efficiently than if they had to assume a complete graph representation, as the number of edges has been reduced from $n \cdot (n-1)$ to m due to the secure graph anonymization. This comes at the cost of leaking some information about the topology of the graph and partially losing the utility of the data due to the perturbation. Note that any other information attached to an edge, such as cost or capacity, is assumed to remain private after the anonymization.

The SecureRelabel protocol receives as input a $n \times n$ secret shared binary viability matrix $[A]$, which stores a positive bit $[1]$ in $[A]_{ij}$ if the edge (i, j) exists or $[0]$ and a secret shared random permutation $[\pi_1]$ and returns its relabelled version $\underline{A}$ by performing MatShuffle on $[A]$.

The SecureRandomAdd-Del protocol receives the graph's secret shared viability matrix $[A]$, the public perturbation parameter ξ, and two secret shared random permutation $[\pi_2]$ and $[\pi_3]$ to return the perturbed version of the viability matrix $\overline{[A]}$. We denote $\mathsf{vec}(Z)$ to describe the flattening of a $n \times n$ matrix into a vector z of size n^2 and $\mathsf{mat}(z)$ to describe its reverse operation.

Algorithm 5 SecureRandomAdd/Del

Input: secret shared viability matrix $[A]$, public perturbation parameter ξ and secret shared random permutation $[\pi_2], [\pi_3]$
Output: secret shared perturbed viability matrix $\overline{[A]}$

```
1:  $[b] \leftarrow \mathsf{Shuffle}(\mathsf{vec}([A]), [\pi_2])$        ▷ Delete first $\xi$ randomized existing edges
2:  $[countDeleted] \leftarrow 0$
3:  for $i \leftarrow 0$ to $n^2 - 1$ do
4:      $[z_1] \leftarrow ([b]_i = 1) \cdot ([countDeleted] < \xi)$
5:      $[b]_i \leftarrow_{[z_1]} 0 : [b]_i$
6:      $[countDeleted] \leftarrow [countDeleted] + [z_1]$
7:  end for
8:  $[c] \leftarrow \mathsf{Shuffle}(b, [\pi_3])$        ▷ Add first $\xi$ randomized non-existing edges
9:  $[countAdded] \leftarrow 0$
10: for $i \leftarrow 0$ to $n^2 - 1$ do
11:     $[z_1] \leftarrow ([c]_i = 0) \cdot ([countAdded] < \xi)$
12:     $[c]_i \leftarrow_{[z_1]} 1 : [c]_i$
13:     $[countAdded] \leftarrow [countAdded] + [z_1]$
14: end for
15: $\overline{[A]} \leftarrow \mathsf{mat}(\mathsf{Unshuffle}(\mathsf{Unshuffle}(c, [\pi_3]), [\pi_2]))$
```

Each of the two algorithms is insufficient by itself, but composing the two by together allows us to obtain a relabelled and perturbed viability matrix $\overline{\underline{[A]}}$, which the parties can safely open. The main cost centers of the protocol are the Delete and Add operations, in which the complexity of multiplications and comparisons is quadratic to the number of nodes in the graph $O(n^2)$. Such operations can be fully parallelized, resulting in a constant round of $O(1)$ interactions. The cost of the algorithm does not depend on ξ.

6 Secure Network Simplex

The protocol for Secure Network Simplex is described in Algorithm 6. This algorithm allows a set of parties to compute the minimum cost flow over a semi-private graph. In particular, the topology of such graph (edges $\mathcal{E}$ and vertices $\mathcal{V}$) is public, and so are the costs associated with each edge. On the contrary, the capacities associated with the edges and the supply/demand of each node are private. Note that the algorithm can easily be extended to support private costs without increasing its asymptotic complexity. The first thing you will notice when comparing it with its plain version (Algorithm 1) is that the while loop based on the optimality condition check is replaced by a for loop running for w iterations. This *stopping rule* hides the actual number of pivots required to find an optimal solution, avoiding any data leakage. The risk is that the resulting flow x might not be optimal. To guarantee an optimal solution with no leakage, the algorithm can be run to its upper bound complexity (see §3.1). Alternatively, the implementer can accept the data leakage by opening the optimality condition check after every pivot iteration and stopping whenever optimality is met. This section analyses each sub-algorithm step by step. Appendix A provides a detailed description of each variable involved in the algorithm.

Algorithm 6 SecureNetworkSimplex

```
Input: semi-private graph $\mathcal{G}^* = (\mathcal{V}, \mathcal{E}, c, [u], [b])$ and target number of network simplex pivot iterations $w$
Output: flow $[x]$ over the edges $\mathcal{E}$ after $w$ iterations
1: $[\mathcal{S}], [\pi], [x], [u], c, \mathcal{V}', \mathcal{E}' \leftarrow \mathsf{Initialize}(\mathcal{G}^*)$
2: for $i = 0$ to $w$ do
3:     $[e_{in}] \leftarrow \mathsf{SelectEnteringEdge}([\mathcal{S}], [\pi], c, \mathcal{E}')$
4:     $[LCA] \leftarrow \mathsf{FindLCA}([\mathcal{S}], [e_{in}], \mathcal{V}')$
5:     $[\delta], [f], [side], [u_{out}], [s], [t], [g], [h] \leftarrow \mathsf{Find}\delta([\mathcal{S}], [e_{in}], [u], [LCA], \mathcal{V}')$
6:     $\mathsf{UpdateFlow}([\delta], [f], [x], \mathcal{E}')$
7:     $[u_{in}] \leftarrow \mathsf{UpdateSpanningTree}([\mathcal{S}], [e_{in}], [x], [u_{out}], [side], [s], [t], \mathcal{V}')$
8:     $\mathsf{UpdatePotentials}([\mathcal{S}], [e_{in}], c, [\pi], [g], [h], [u_{in}], \mathcal{V}')$
9: end for
```

Initialize To kick off the protocol, an initial feasible solution must be created. We build a spanning tree made of an artificial root node (with reserved index

n) and artificial edges [18, 3, 40]. All real edges are initially placed in the lower set with flow equal to zero (lines 5-6). For every existing node v with a non-negative supply (line 9), an artificial edge from v to the artificial root node n is added to the spanning tree (line 10) with a capacity equal to its supply and a flow saturating it. For every node v with a negative supply (namely, a positive demand), an artificial edge from v to n is added to the spanning tree with a capacity equal to the negative of its supply and a flow saturating the capacity. The artificial edges get assigned an infinite cost ac. This ensures that the algorithm will prefer real edges over artificial ones when optimizing the flow. In practice, this cost is computed as the sum of the absolute values of all edge costs plus one. The artificial edges are added to the set $\mathcal{E}'$ (line 14) with their index $m + v$, the two endpoints v and r, and a secret bit $[z]$ indicating whether the edge is directed from v to r or the opposite. Such a bit defaults to 1 for all the real edges. The node potentials (line 15) are calculated leveraging the formula for the reduced cost optimality condition for spanning tree edges and considering that the potential associated with the root node n equals 0. The parent of node v is set to r, the predecessor edge of node v is set to the newly created artificial edge with index $m+v$ together with a bit indicating whether the edge is oriented towards the root or the opposite and the children of r is set to v (lines 16-18). Note that the root has no parent node or predecessor edge, so these are set to default to -1 and $(-1, -1)$. This setup, performed only once, ensures that the algorithm starts with a valid partition over the tree sets $(\mathcal{T}, \mathcal{L}, \mathcal{U})$ and a feasible flow, ready for the network simplex method to optimize. The cost is dominated by the n comparisons to evaluate the sign of the balance of a node (line 9). All the loops can be run in parallel, so the round complexity is $O(1)$.

Algorithm 7 Initialize

Input: semi-private graph $\mathcal{G}^* = (\mathcal{V}, \mathcal{E}, c, [u], [b])$
Output: spanning tree structure $[\mathcal{S}] = ([par], [pred], [children], [t], [lu])$, node potential array $[\pi]$, flow array $[x]$, capacity array $[u]$, cost array c, updated set of nodes $\mathcal{V}'$ and updated set of edges $\mathcal{E}'$

```
 1: $ac \leftarrow \infty$, $r = n$
 2: $[par] \leftarrow -1$, $[pred] \leftarrow (-1, -1)$, $[\pi] \leftarrow 0$, $[b]_r \leftarrow 0$
 3: $[lu] \leftarrow 0$, $[t] \leftarrow 0$, $\mathcal{E}' \leftarrow \mathcal{E}$
 4: for $e \in \mathcal{E}'$ do
 5:     $[x]_e \leftarrow 0$
 6:     $[lu]_e \leftarrow -1$
 7: end for
 8: for $v \in \mathcal{V}$ do
 9:     $[z] \leftarrow [b]_v \geq 0$
10:     $[t]_{m+v} \leftarrow 1$
11:     $[u]_{m+v} \leftarrow_{[z]} [b]_v : -[b]_v$
12:     $[x]_{m+v} \leftarrow [u]_{m+v}$
13:     $c_{m+v} \leftarrow ac$
14:     $\mathcal{E}' \leftarrow \mathcal{E}' \cup (m+v, v, r, [z])$
15:     $[\pi]_v \leftarrow_{[z]} -c_{m+v} : c_{m+v}$
16:     $[par]_v \leftarrow r$
17:     $[pred]_v \leftarrow (m+v, [z])$
18:     $[children]_{rv} \leftarrow 1$
19: end for
20: $\mathcal{V}' \leftarrow \mathcal{V} \cup r$
```

SelectEnteringEdge After establishing an initial feasible solution, we enter the pivot phase: every algorithm listed from now has to be executed a constant number of times w. The first step is to find a candidate entering edge. Any edge that violates the optimality condition, as described in §3.2, is a potential entering edge. There are different rules to choose this edge [3]. For faster pivot iteration, usually the first eligible edge is picked as a candidate. Nevertheless, the data-independent nature of our secure algorithm forces us to loop over all the edges anyway. Therefore, Dantzig's pivot rule, which chooses the edge that maximally violates the optimality condition, is the most logical choice. Such a rule also guarantees a faster convergence towards the optimal solution. The algorithm proceeds by computing the reduced costs of the edges (lines 3-4) and calculating the minimum reduced costs $[minval]$ and the corresponding edge index $[minidx]$ (line 6). If the minimum reduced cost is not negative (line 7), no edge is violating the optimality condition, and the optimal solution has already been found. If that's the case, the index $[idx_{in}]$, the endpoints $[k], [l]$, and the direction $[d]$ of the entering edge are set to -1. Otherwise, these are set to the respective values of the edge with an index equal to $[minidx]$ (lines 11-14). The most expensive operation is the Min function, which requires a $O(\log \log n + m)$ rounds protocol using $O(n+m)$ comparisons and multiplications. Both the loops can be run in parallel in a constant amount of rounds.

Algorithm 8 SelectEnteringEdge

Input: spanning tree structure $[\mathcal{S}]$, node potential array $[\pi]$ and cost array c and edge set $\mathcal{E}'$
Output: entering edge $[e_{in}] = ([idx_{in}], [k], [l], [d_{in}])$ with the highest reduced cost

```
 1: $[c^{\pi}] \leftarrow 0$
 2: for $(idx, i, j, [d]) \in \mathcal{E}'$ do
 3:     $[\pi'_i], [\pi'_j] \leftarrow \mathsf{CondEx}([\pi]_i, [\pi]_j, [d])$
 4:     $[c^{\pi}]_{idx} \leftarrow -[lu]_{idx} \cdot (c_{idx} - [\pi'_i] + [\pi'_j])$
 5: end for
 6: $[minval], [minidx] \leftarrow \mathsf{Min}([c^{\pi}])$
 7: $[z_1] \leftarrow [minval] >= 0$
 8: $[idx_{in}] \leftarrow_{[z_1]} -1 : [minidx]$
 9: $[k], [l], [d] \leftarrow (-1, -1, -1)$
10: for $(idx, i, j, [d]) \in \mathcal{E}'$ do
11:     $[z_2] \leftarrow idx = idx_{in}$
12:     $[k] \leftarrow_{[z_2]} i : [k]$
13:     $[l] \leftarrow_{[z_2]} j : [l]$
14:     $[d_{in}] \leftarrow_{[z_2]} [d] : [d_{in}]$
15: end for
```

FindLCA As the chosen candidate edge is added to the spanning tree, a new cycle is formed. We aim to find the lowest common ancestor (LCA) of the endpoint nodes of the entering edge, which represents the apex of the cycle in the spanning tree rooted in node r, by traversing up the tree from both endpoints simultaneously until a point where their paths meet is found. This point is the LCA. As soon as the LCA is found ($[z_3] = 0$), no more updates on the placeholders $[g]$ and $[h]$ are performed. The comparisons on indexed arrays (lines 4-5) are to be performed $n-1$ times for each iteration in an equal number of rounds.

Algorithm 9 FindLCA

Input: spanning tree structure $[\mathcal{S}]$, entering edge $[e_{in}]$ and node set $\mathcal{V}'$
Output: lowest common ancestor $[LCA]$ node of the entering edge endpoint nodes, which is the apex of the newly generated cycle in the spanning tree

```
 1: $[g] \leftarrow [k]$
 2: $[h] \leftarrow [l]$
 3: for $v \in \mathcal{V}'$ do
 4:     $[z_1] \leftarrow [par]_{[g]} \neq -1$
 5:     $[z_2] \leftarrow [par]_{[h]} \neq -1$
 6:     $[z_3] \leftarrow g \neq h$
 7:     $[g_{next}] \leftarrow_{[z_1]} [par]_{[g]} : [l]$
 8:     $[h_{next}] \leftarrow_{[z_2]} [par]_{[h]} : [k]$
 9:     $[g] \leftarrow_{[z_3]} [g_{next}] : [g]$
10:     $[h] \leftarrow_{[z_3]} [h_{next}] : [h]$
11: end for                                  ▷ $[g]$ is the $[LCA]$ node
```

Findδ A further scan over the cycle is required to identify the maximum amount of flow $[\delta]$ that can be augmented along the cycle. The endpoints of the entering edge are ordered according to its direction (line 1). Second, we create the source $[s]$ and target $[t]$ variables to denote the orientation of the cycle (line 3): flow can be pushed in the direction of an edge if the edge is empty (chosen from the lower set), or in the opposite direction if the edge is saturated (chosen from the upper set). $[\delta]$ is initially set to the maximum value possible, which is the capacity of the entering edge. Starting from the source node, we loop from the source node to the LCA node and from the target node to the LCA node, visiting the edges of the cycle using $[u_{next}]$ as a node placeholder variable. The $[count]$ variable tracks the number of times the $[LCA]$ node is encountered. $[count] = 0$ indicates we are on the source-side branch of the cycle, while if $[count] = 1$, we are on the target-side branch. The flow $[\delta_{temp}]$ that can be pushed over the next edge of the cycle, identified by its index $[idx]$, is calculated according to its orientation (line 11). Figure 2 illustrates this process. Whenever a new minimum $[\delta_{temp}]$ is found, it updates the $[\delta]$ variable, the $[u_{out}]$ variable, representing the lower endpoint node corresponding the blocking edge, and the $[side]$ variable, which indicates whether the minimum has been found in the path from source to LCA $[side] = 0$ or in the path from target to LCA $[side] = 1$ (lines 16-18). To avoid degeneracy, the leaving edge must be selected as the last blocking edge encountered in traversing the cycle along its orientation starting from its apex node LCA. The proof is provided by [3]. We ensure that by choosing only the first blocking edge when we are on the source side of the cycle $[z_4] \cdot [z_6] = 1$ and the latest blocking edge when we are on the target side of the cycle $([z_4] + [z_5]) \cdot [z_2] = 1$. Lastly, we construct the binary array $[f]$ of size $m + n$ that stores a positive bit at the index of an edge whose flow has to be augmented, a negative bit if the flow has to be decreased, and a 0 if the flow is untouched. The main loop cannot be parallelized and, therefore, requires $n + 1$ communication rounds between the parties. The dominant costs inside the loop are the fetching operations over the secret shared arrays $[u]$, $[x]$, and $[f]$, which require $n + m$ comparisons and $3 * (n + m)$ multiplications to be performed for each loop iteration.

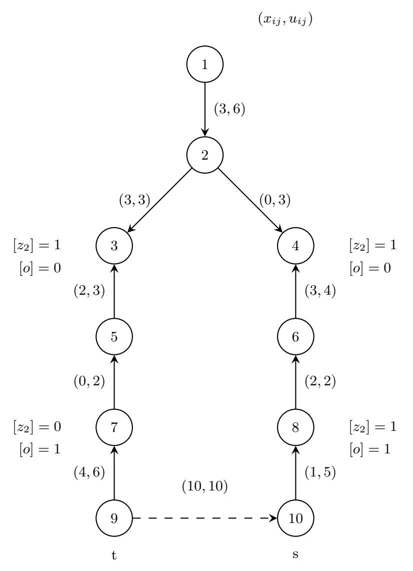

Fig. 2: Binary conditions over the newly created cycle

Algorithm 10 Findδ

Input: spanning tree structure $[\mathcal{S}]$, entering edge $[e_{in}]$, edge capacities $[u]$, $[LCA]$ node and node set $\mathcal{V}'$

Output: maximum quantity of flow $[\delta]$ that can be sent along the newly formed cycle, array $[f]$ indicating whether the flow over an edge has to be updated and in which direction, $[side]$ over which $[\delta]$ was found, lower endpoint node $[u_{out}]$ of the exiting edge, source $[s]$ and target $[t]$ nodes indicating the orientation of the cycle, $[g]$ and $[h]$ nodes such that the entering edge is directed from $[g]$ to $[h]$

```
1: $[g], [h] \leftarrow \mathsf{CondEx}([k], [l], [d_{in}])$
2: $[z_1] \leftarrow [lu]_{[idx_{in}]} = -1$
3: $[s], [t] \leftarrow \mathsf{CondEx}([k], [l], [z_1])$
4: $[\delta] \leftarrow [u]_{[idx_{in}]}, [u_{out}] \leftarrow -1, [side] \leftarrow -1, [count] \leftarrow 0, [u_{next}] \leftarrow [s], [f] \leftarrow 0$
5: $[f]_{[idx_{in}]} \leftarrow -[lu]_{[idx_{in}]}$
6: for $v \in \mathcal{V}'$ do
7:     $[z_2] \leftarrow [count] > 0$
8:     $[z_3] \leftarrow [u_{next}] \neq [LCA]$
9:     $[count] \leftarrow_{[z_3]} [count] : [count] + 1$
10:    $[idx], [o] \leftarrow [pred]_{[u_{next}]}$
11:    $[\delta_{temp}] \leftarrow_{[o]=[z_2]} [u]_{[idx]} - [x]_{[idx]} : [x]_{[idx]}$
12:    $[z_4] \leftarrow [\delta_{temp}] < [\delta]$
13:    $[z_5] \leftarrow [\delta_{temp}] = [\delta]$
14:    $[z_6] \leftarrow [z_2] = 0$
15:    $[z_7] \leftarrow [z_2] \cdot ([z_4] \cdot [z_6]) + (([z_4] + [z_5]) \cdot [z_2])) = 1$
16:    $[\delta] \leftarrow_{[z_7]} [\delta_{temp}] : [\delta]$
17:    $[u_{out}] \leftarrow_{[z_7]} [u_{next}] : [u_{out}]$
18:    $[side] \leftarrow_{[z_7]} [z_2] : [side]$
19:    $[u_{next}] \leftarrow_{[z_3]} [par]_{[u_{next}]} : [t]$
20:    $[sign] \leftarrow_{[o]=[z_2]} 1 : -1$
21:    $[f]_{[idx]} \leftarrow_{[z_3]} [sign] : [f]_{[idx]}$
22: end for
```

UpdateFlow The next step is to push $[\delta]$ amount of flow over the edges of the cycle. Such an algorithm requires $O(m+n)$ multiplications, which can be performed in a single round.

Algorithm 11 UpdateFlow

Input: quantity of flow $[\delta]$, array $[f]$, flow array $[x]$ and edge set $\mathcal{E}'$
Functionality: pushes $[\delta]$ flow over the edges of the cycle updating $[x]$

```
1: for $e \in \mathcal{E}'$ do
2:     $[x]_e \leftarrow [x]_e + [f]_e \cdot [\delta]$
3: end for
```

UpdateSpanningTree After having updated the flow along the cycle, the spanning tree structure $[S]$, which includes the state of the edges ($[t]$ and $[lu]$ array) and the data structures of the spanning tree ($[children]$ matrix and $[par]$ and $[pred]$ arrays) have to be updated as well. Lines 1-5 handle an edge case; we will get to that later. The entering edge, identified by its index $[idx_{in}]$, is moved to the spanning tree (line 6), while the exiting edge, identified by its index $[idx_{out}]$, now moves either to the lower set if the edge is empty or to the upper set if the capacity of the edge is saturated (lines 7-8). The branch of the spanning tree affected by the pivot operation is identified by the $[side]$ variable. Figure **??**, on the left, illustrates a scenario in which the existing edge is found on the source side: we, therefore, pick the source node as the u_{in} node (line 9). The nodes marked in light blue on the right are the ones whose children are affected by the pivot operation. The ones in dark blue are the ones whose children, parent node, and predecessor edge are affected. To implement the update, we first traverse the tree from the $[u_{in}]$ node to the $[u_{out}]$, storing the nodes in the $[path]$ array (lines 11-15). Second, we reverse the parent-child relationships going down the $[path]$ (lines 16-25). In the process, we build a support array $[y]$, whose values are initiated at -1, that for each node $[el]$ keeps track of their new parent $[u_{next}]$. The vector is then used to update the $[children]$ matrix (lines 27-34). The dominating costs are the oblivious array indexing and assignments on lines 13, 21, 22, and 24, which have to be performed $n+1$ times in an equal number of rounds. The process to update the children matrix requires $(n+1)^2$ multiplication, which can be performed in parallel, resulting in a single round of communication.

Algorithm 12 UpdateSpanningTree

Input: spanning tree structure $[\mathcal{S}]$, entering edge $[e_{in}]$, flow array $[x]$, lower endpoint of the exiting node $[u_{out}]$, $[side]$, source and target nodes $[s]$ and $[t]$ and node set $\mathcal{V}'$
Functionality: updates the states of the entering and leaving edge and the spanning tree-related data structures for the nodes affected by the pivot
Output: endpoint of the entering edge which is on the side of the leaving edge $[u_{in}]$

```
1: $[z_1] \leftarrow [side] = -1$
2: $[lu]_{[idx_{in}]} \leftarrow_{[z_1]} -[lu]_{[idx_{in}]} : [lu]_{[idx_{in}]}$
3: $[s], [t], [idx_{in}] \leftarrow_{[z_1]} -1, -1, -1 : [s], [t], [idx_{in}]$
4: $[idx_{out}] \leftarrow_{[z_1]} -1 : [idx]$ s.t. $([idx], [o]) = [pred]_{[u_{out}]}$,
5: $[side] \leftarrow_{[z_1]} 0 : [side]$
6: $[lu]_{[idx_{in}]} \leftarrow 0$, $[t]_{[idx_{in}]} \leftarrow 1$
7: $[z_2] \leftarrow [x]_{[idx_{out}]} = 0$
8: $[lu]_{[idx_{out}]} \leftarrow_{[z_2]} -1 : 1$, $[t]_{[idx_{out}]} \leftarrow 0$
9: $[u_{in}], [v_{in}] \leftarrow \mathsf{CondEx}([t], [s], [side])$
10: $[u_{next}] \leftarrow [u_{in}], [path] \leftarrow -1, [y] \leftarrow -1$
11: for $v \in \mathcal{V}'$ do
12:     $[z_3] \leftarrow [u_{next}] \neq [u_{out}]$
13:     $[path]_v \leftarrow_{[z_3]} [u_{next}] : [path]_v$
14:     $[u_{next}] \leftarrow_{[z_3]} [par]_{[u_{next}]} : [u_{next}]$
15: end for
16: for $k \leftarrow n$ downto $0$ do
17:     $[u_{next}] \leftarrow [path]_k$
18:     $[z_4] \leftarrow [u_{next}] \neq -1$
19:     $[el] \leftarrow_{[z_4]} [par]_{[u_{next}]} : -1$
20:     $[y]_{[el]} \leftarrow [u_{next}]$
21:     $[par]_{[el]} \leftarrow [u_{next}]$
22:     $[idx], [o] \leftarrow [pred]_{[u_{next}]}$,
23:     $[z_5] \leftarrow [o] = 0$,
24:     $[pred]_{[el]} \leftarrow_{[z_5]} [idx], 1 : [idx], 0$
25: end for
26: $[y]_{[u_{in}]} \leftarrow [v_{in}]$
27: for $v \in \mathcal{V}'$ do
28:     $[z_6] \leftarrow [y]_v \neq -1$
29:     for $i \in \mathcal{V}'$ do
30:         $[z_7] \leftarrow [y]_v = i$
31:         $[children]_{iv} \leftarrow_{[z_6]} 0 : [children]_{iv}$
32:         $[children]_{iv} \leftarrow_{[z_6]\cdot[z_7]} 1 : [children]_{iv}$
33:     end for
34: end for
35: $[par]_{[u_{in}]} \leftarrow [v_{in}]$
36: $[z_8] \leftarrow [y] = [u_{in}]$
37: $[pred]_{[u_{in}]} \leftarrow_{[z_8]} [idx_{in}], 0 : [idx_{in}], 1$
```

UpdatePotentials The last step is to update the node potentials, which will be used in the following pivot iteration. Only the nodes in the subtree rooted at

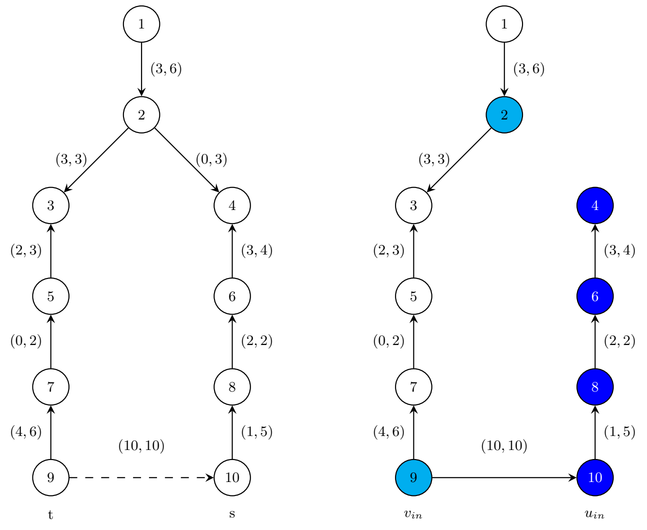

Fig. 3: Spanning tree before and after pivot

u_{in} (see Figure 3 for an example) require an update. The update is performed to maintain the reduced cost optimality condition for the spanning tree edges, as defined in §3.2. The algorithm is divided in two parts. In the first part (lines 3-32), we build the array $[e]$, which is activated at the indexes of the nodes that require potential updates. The second part (lines 33-39) performs the update. In the first part, we fetch the children of u_{in} and, recursively, the children of their children and activate the respective slot in the $[e]$ array. The first loop (lines 8-10) count the number of nodes in the queue, while the second loop (lines 13-18) activates the $[next]$ array at the index corresponding to the first positive value in $[q]$ (line 16) and deactivates the queue at the same index (line 17). $[next]$ is a binary array that indicates the index of the *next child node* to be added to $[e]$. We, then, fetch the children of this node by subtracting each value in the slot i of the $[next]$ vector from each value of the row i in the *clone* matrix. Only for the activated slot in $[next]$ its children (value 1 in the $[clone]$ matrix) will go to 0 after the update, while their 0 value (indicating a non-children) will go to $p-1$ (lines 19-23). The children are identified by comparing the matrix column sums before and after the update: all the columns whose sums decreased (line 28) indicate that the node corresponding to the column index is a child of the *next child node* and are therefore added to $[q]$. Lastly, the update is performed for every node whose index is activated in $[e]$. The nested loop at lines 20-22 only involves linear operation and, therefore, is costless. The dominating costs are the comparisons done in the nested loops (lines 9, 14, and 28), accounting for $(n+1)^2$ comparisons each. Note that each nested loop can be run in parallel without any round communication.

Algorithm 13 UpdatePotentials

Input: spanning tree structure $[\mathcal{S}]$ entering edge $[e_{in}]$, cost vector c, node potential vector $[\pi]$, $[g]$ and $[h]$ nodes, endpoint of the entering edge $[u_{in}]$ and node set $\mathcal{V}'$
Functionality: Update the potential of the nodes in the subtree rooted at u_{in} to maintain the reduced cost optimality conditions

```
1:  $[e] \leftarrow 0$, $[sums] \leftarrow 0$, $[q] \leftarrow 0$
2:  $[clone] \leftarrow [children]$, $[q]_{[u_{in}]} \leftarrow 1$,
3:  for $v \in \mathcal{V}'$ do
4:      $[sums]_v \leftarrow \sum_{l \in \mathcal{V}'} [clone]_{lv}$
5:  end for
6:  for $v \in \mathcal{V}'$ do
7:      $[count] \leftarrow 0$
8:      for $i \in \mathcal{V}'$ do
9:          $[z_1] \leftarrow [q]_i > 0$
10:         $[count] \leftarrow [count] + [z_1]$
11:     end for
12:     $[next] \leftarrow 0$
13:     for $k \leftarrow n$ downto $0$ do
14:         $[z_2] \leftarrow [count] = 1$
15:         $[count] \leftarrow [count] - [q]_k$
16:         $[next]_k \leftarrow_{[z_2] \cdot [q]_k} 1 : [next]_k$
17:         $[q]_k \leftarrow_{[z_2] \cdot [q]_k} 0 : [q]_k$
18:     end for
19:     for $i \in \mathcal{V}'$ do
20:         for $j \in \mathcal{V}'$ do
21:             $[clone]_{ij} \leftarrow [clone]_{ij} - [next]_i$
22:         end for
23:     end for
24:     $[sums'] \leftarrow 0$
25:     for $i \in \mathcal{V}'$ do
26:         $[e]_i \leftarrow [e]_i + [next]_i$
27:         $[sums']_i \leftarrow \sum_{l \in \mathcal{V}'} [clone]_{li}$
28:         $[z_3] \leftarrow [sums']_i < [sums]_i$
29:         $[q]_i \leftarrow [q]_i + [z_3]$
30:     end for
31:     $[sums] \leftarrow [sums']$
32: end for
33: $[c^{\pi}_{gh}] \leftarrow c_{[idx_{in}]} - [\pi]_{[g]} - [\pi]_{[h]}$
34: $[z_4] \leftarrow [g] = [u_{in}]$
35: $[\Delta\pi] \leftarrow_{[z_4]} -[c^{\pi}_{gh}] : [c^{\pi}_{gh}]$
36: for $v \in \mathcal{V}'$ do
37:     $[z_5] \leftarrow [e]_v > 0$
38:     $[\pi]_v \leftarrow [\pi]_v + [z_5] \cdot [\Delta\pi]$
39: end for
```

Edge Cases Since the number of pivot iterations w is defined a priori, there is a chance that an optimal solution is found earlier. Nevertheless, the algorithm is designed in a data-independent way to keep running without leaking this information to the computing parties. In such a scenario, there is no edge violating the optimality condition: the entering edge will therefore be chosen as $(-1, -1, -1, -1)$. The analysis of how the following algorithms (9, 10, 11, 12, 13) will not further modify any data structure as soon as optimality is met is left as an exercise to the reader. Another scenario appears whenever the entering edge, identified by its index $[idx_{in}]$, corresponds to the leaving edge $[idx_{out}]$. In that case, the entering edge is the edge of the cycle over which the maximum quantity of flow $[\delta]$ can be sent, so $[side]$, which is initiated at -1, does never get updated. After the flow update, the edge is moved from the lower set to the upper set (or vice-versa), and no further modification to the spanning tree is required. This edge case (no pun intended) is handled in Algorithm 12 (lines 1-5): after the state update (line 2), the data related to the entering edge and the leaving edge (lines 3-4) are set to -1, which means no further updates are performed during that pivot iteration.

Overall Table 1 synthesizes the complexity analysis for each algorithm. Overall, the Secure Network Simplex requires $O(max(n, \log\log n + m))$ rounds of interactions in which $O(max(n^2, nm))$ comparisons and multiplications have to be performed for each pivot operation. The number of pivot iterations is determined by the public parameter w. In §8, we will further analyze how to set w in order to get a close-to-optimal solution.

Algorithm	Round Complexity	Communication Complexity
7	$O(1)$	$O(n)$ comparisons
8	$O(\log\log n + m)$	$O(n + m)$ comparisons and multiplications
9	$O(n)$	$O(n^2)$ comparisons and multiplications
10	$O(n)$	$O(max(n^2, nm))$ comparisons and multiplications
11	$O(1)$	$O(n + m)$ multiplications
12	$O(n)$	$O(n^2)$ comparisons and multiplications
13	$O(n)$	$O(n^2)$ multiplications

Table 1: Complexity Analysis for Secure Network Simplex Algorithms

7 Secure Multilateral Trade Credit Set-off

State-of-the-art MTCS protocols, as introduced by Fleischman and Dini [28], rely on an agency that receives the obligations data from the firms, performs the set-off, and returns the updated data to the firms. This setup compromises the privacy over the firms' trade credit data, specifically revealing the amount over an obligation and the trade credit links between firms. Secure MTCS overcomes this limitation by replacing the trusted agency with a set of parties performing an MPC protocol. Figure 4 describes the different setups.

Given the scale of our problem, we are interested in the clients-to-servers MPC setting in which some clients supply input to a set of servers that perform the computation and return the output to the firms. A similar setting is implemented by Damgård et al. [26], and Baldimtsi et al. [13]. In secure MTCS, the clients are the n firms, while the servers are institutions with conflicting interests, such as the chamber of commerce, the tax authority, and a payment processor. A different setting in which all the firms join the MPC would be infeasible, given that the communication scales quadratically to the number of computing parties. Furthermore, this would require the firms to be always online throughout the protocol, which is not the case in the clients-to-servers model. The correctness of the protocol execution and the privacy of the data provided by the firms is guaranteed under the trust assumptions of the chosen MPC protocol. The number of parties can also vary across protocols but usually does not exceed four for efficiency reasons.

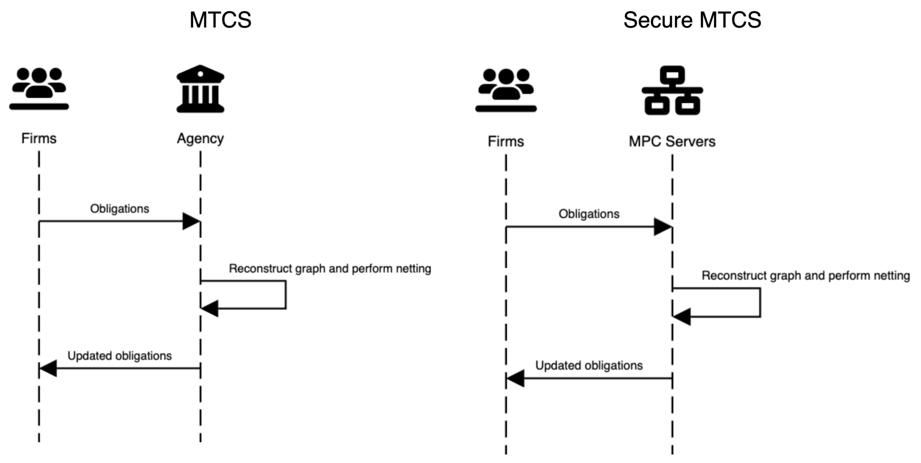

Fig. 4: MTCS and Secure MTCS procedures

Input Delivery Assuming that the identities of the firms joining the MTCS in known by all the participants, each i-th firm secret shares two n-sized vectors to the servers: $[o_i]$ and $[a_i]$. The former is a list of the IOUs (or obligations) that the firm i owes to the other firms such that $o_i(j)$ is how much i owes j. The latter is an auxiliary viability binary vector set to 1 for an existing obligation and 0 otherwise such that $a_i(j) = 1$ if $o_i(j) > 0$ and $a_i(j) = 0$ otherwise. The servers aggregate these vectors into an obligation matrix $[O]$ and a viability matrix $[A]$ such that the vectors $[o_i]$ and $[a_i]$ represent the i-th row, respectively, in the matrices $[O]$ and $[A]$. We assume that the inputs provided by the firms are valid.

Graph Anonymization The servers proceed with the graph anonymization to obtain a relabelled and perturbed version of the viability matrix $\overline{\underline{[A]}}$. The secret shared random permutations $[\pi_1]$, $[\pi_2]$ and $[\pi_3]$ can be generated during the MPC setup phase. The perturbation parameter ξ is public. This procedure involves feeding the viability matrix $[A]$ and $[\pi_1]$ into the SecureRelabel protocol and then feeding its output $\underline{[A]}$, together with ξ, $[\pi_2]$ and $[\pi_3]$ into the SecureRandomAdd/Del protocol to obtain a secret shared relabelled and perturbed viability matrix $\overline{\underline{[A]}}$. We also shuffle the obligation matrix $[O]$ with $[\pi_1]$ to obtain $\underline{[O]}$ such that its entries are correctly paired with the relabelled viability matrix.

GraphOpening After being anonymized, the graph can be safely opened. The anonymization protects from an adversary that might leverage prior knowledge of node degrees to perform re-identification attacks. This protocol yields a semi-private and anonymized version of the original graph $\mathcal{G}^*$ in which the nodes $\mathcal{V}$ are public and relabelled, and the edges are public and perturbed. The cost associated with each viable edge is public and set to 1 given the problem definition defined in §2. Note that the capacities of the edges (i.e., obligation amount) and the balances of nodes (i.e., net balance of each firm) remain private throughout the Secure MTCS process.

Algorithm 14 GraphOpening

Input: secret shared perturbed and relabelled viability matrix $\overline{\underline{[A]}}$, secret shared random permutation $[\pi_1]$, secret shared relabelled obligation matrix $\underline{[O]}$
Output: semi-private graph $\mathcal{G}^* = (\mathcal{V}, \mathcal{E}, c, [u], [b])$

```
 1: $\mathcal{V} \leftarrow \{0, \ldots, n-1\}$, $\mathcal{E} \leftarrow \{\}$, $c \leftarrow \{\}$, $[u] \leftarrow \{\}$, $[b] \leftarrow \{\}$
 2: for $i \leftarrow 0$ to $n-1$ do
 3:     for $j \leftarrow 0$ to $n-1$ do
 4:         $v \leftarrow \mathsf{Open}(\overline{\underline{[A]}}_{ij})$
 5:         if $v = 1$ then
 6:             $\mathcal{E} \leftarrow \mathcal{E} \cup \{(i,j)\}$
 7:             $c \leftarrow c \cup \{1\}$
 8:             $[u] \leftarrow [u] \cup \{\underline{[O]}_{ij}\}$
 9:         end if
10:     end for
11: end for
12: for $i \leftarrow 0$ to $n-1$ do
13:     $[b_i] \leftarrow 0$
14:     for $j \leftarrow 0$ to $n-1$ do
15:         $[b_i] \leftarrow [b_i] + \underline{[O]}_{ji} \cdot \mathsf{Open}(\overline{\underline{[A]}}_{ji})$
16:         $[b_i] \leftarrow [b_i] - \underline{[O]}_{ij} \cdot \mathsf{Open}(\overline{\underline{[A]}}_{ij})$
17:     end for
18:     $[b]_i \leftarrow [b_i]$
19: end for
```

Minimum Cost Flow Finally, the servers can perform the multilateral trade credit set-off procedure in a secure environment. As illustrated in §2 this is equivalent to solving the minimum cost flow problem over the graph describing the trade credit relations between the firms. We can leverage the SecureNetworkSimplex algorithm using the semi-private graph describing the obligation network $\mathcal{G}^*$ and the public target number of network simplex pivot iterations w as inputs to obtain the minimized flow $[x]$ over the edges, representing the updated obligation.

OutputDelivery Lastly, the servers will share the update obligations to the firms. This procedure requires them to reassemble an updated version of the relabelled obligation matrix $\underline{[O']}$, undo the relabelling to obtain $[O']$, and return the updated obligations to the designated firms. In the process, we need to account for viable edges that have been deleted as part of the perturbation process. For those edges, the updated state resembles the original state (lines 5-10). The updated obligation $[O]_{ij}$ is secret-shared to firms i and j (lines 13-18). Note that the client can evaluate constraint correctness, as defined in §2, starting from the inputs provided to the parties and the output received at the end of the process. This makes the protocol secure even if all the servers are malicious.

Algorithm 15 OutputDelivery

Input: opened perturbed and relabelled viability matrix $\underline{\overline{A}}$, secret shared relabelled obligation matrix $\underline{[O]}$, minimized flow $[x]$, edge set $\mathcal{E}$, random secret $[\pi_1]$
Functionality: secret share the updated obligation state to respective firms

```
1: $\underline{[O']} \leftarrow 0$
2: for $e = (i, j) \in \mathcal{E}$ do
3:     $\underline{[O']}_{ij} \leftarrow [x]_e$
4: end for
5: for $i \leftarrow 0$ to $n-1$ do
6:     for $j \leftarrow 0$ to $n-1$ do
7:         if $\underline{\overline{A}}_{ij} = 0$ then
8:             $\underline{[O']}_{ij} \leftarrow \underline{[O]}_{ij}$
9:         end if
10:    end for
11: end for
12: $[O'] \leftarrow \mathsf{UnshuffleMat}(\underline{[O']}, [\pi_1])$
13: for $i \leftarrow 0$ to $n-1$ do
14:    for $j \leftarrow 0$ to $n-1$ do
15:        $\mathsf{Open}_{\mathsf{E}}([O']_{ij}, i)$
16:        $\mathsf{Open}_{\mathsf{E}}([O']_{ij}, j)$
17:    end for
18: end for
```

Overall The cost of the Secure MTCS protocol is dominated by the SecureNetworkSimplex algorithm such that $O(max(n, \log\log n + m))$ rounds of interactions between the servers in which $O(max(n^2, nm))$ comparisons and multiplications have to be performed for each pivot operation. Note that if we were to skip the Graph Anonymization procedure, we would need to assume, similarly to Aly and Van Vyne [6], a complete graph in which $m = n^2$ in order to guarantee privacy, increasing the complexity of the algorithm. The choice of the public variables ξ and w determines a tradeoff between privacy and optimality and between efficiency and optimality, which will be investigated in the next paragraph.

8 Experiments

This section empirically evaluates the tradeoff between privacy and optimality within the secure MTCS protocol. Additionally, we leverage synthetic datasets to estimate the number of network simplex iterations w required.

Privacy indicates the information leaked to the computing parties during the protocol, which can help them fetch information about firms' sensitive trade credit data. In particular, leakage appears once the graph is opened after the anonymization. We consider an attacker with degree-based knowledge looking to re-identify nodes in the anonymized graph to discern trade credit relations between firms. Note the obligations' amounts or firms' balances always remain private. Using the framework proposed by Casas-Roma [20], we measure the number of nodes that changed their degree during the perturbation process. The higher the percentage of vertices that changed their degree, the harder the re-identification process, the better the privacy guarantee for the participating firms.

Optimality indicates the percentage of excess obligations cleared from the network after the MTCS process. We define this metric as "Clearing Percentage", similar to Agarwal et al., [2]. The clearing percentage is defined as:

$$\text{Clearing Percentage} = \frac{\text{Actual Cleared Amount}}{\text{Optimal Cleared Amount}} \times 100$$

Any percentage above zero indicates that the firms are better off than how they started. This also ties to the definition of *optimality correctness*, which is achieved when Clearing Percentage = 100%

8.1 Evaluation Framework

The experiments leverage three synthesized datasets (where n=28975, 35739, and 40743) kindly provided by Tomaž Fleischman, formerly of BeSolutions, which replicate statistical properties of confidential real-world Slovenian and Italian trade credit data. Such data confirm the scale-free properties of inter-firm payment networks data discovered by Tamura et al. [42], and by Berloco et al. [14]. Since ξ and w are the only variables that can be tuned, we are interested in investigating the following:

1. ξ impact on privacy by measuring the degree node change after the graph anonymization
2. ξ impact on optimality by measuring the clearing percentage after the graph anonymization
3. how to choose w based on the network magnitude

For graph anonymization, we consider a perturbation parameter ξ in ranging between 0% (original graph) and 20% of the total number of edges m in the graph. Given that the process involves randomness, we always run five independent executions, with different seeds, for each experiment and show the average value. The experiments use a custom-designed library for graph anonymization [4] and the NetworkX library [5] for minimum cost flow and its subroutine network simplex.

8.2 Results

The experiment illustrated in Figure 5 shows the percentage of nodes that changed their degree due to different perturbation levels. The curves demonstrate a similar asymptotic behavior for each dataset. Figure 6 illustrates that the clearing percentage decreases linearly with the perturbation percentage. This is because some existing obligations might be deleted due to the perturbation process. Overall, a meaningful level of anonymity (74%, 81% and 86% of nodes with changed degree, respectively for each network) and sub-optimal clearing percentage (63%, 63 and 66% of the optimal) can be achieved with a perturbation of 20% of the edges in the graph ($\xi = 0.2m$).

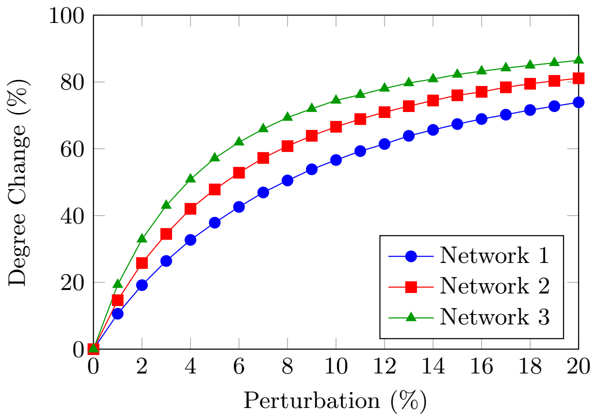

Fig. 5: Percentage of nodes that change their degree after the perturbation process

[4] https://github.com/enricobottazzi/mtcsPy
[5] https://networkx.org/

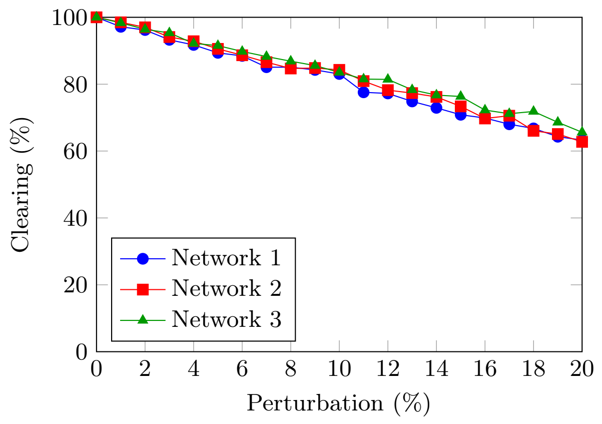

Fig. 6: Clearing percentage after the perturbation process

Finally, we measure the number of network simplex iterations required to obtain an optimal solution to the minimum cost flow problem. The results in Table 2 show a roughly linear relationship between nm and the number of iterations. This confirms the results from Király and Kovács [37] and provides a starting point to choose w starting from the trade credit network data. Note that the network simplex algorithm maintains a feasible solution at each pivot iteration, so any choice of w would result in a solution that satisfies *constraint correctness*, although possibly suboptimal in terms of Clearing Percentage.

Nodes (n)	**Edges (m)**	**n*m**	**Iterations**	**(n*m)/Iterations**
28,975	100,471	2,911,147,725	128,114	22,723
35,739	179,586	6,418,134,554	246,984	25,986
40,743	284,739	11,601,155,877	410,978	28,228

Table 2: Analysis of network size and Network Simplex iterations

9 Conclusions and Future Work

In this paper, we proposed a secure MTCS protocol that firms can leverage without relying on a single agency having access to all the sensitive trade credit data of the firms. The efficiency of the proposed solution comes from two novel techniques introduced in the paper: a secure graph anonymization protocol and a secure network simplex protocol. While the secure network simplex protocol can be leveraged as a standalone technique to increase the efficiency of secure MCF solvers, we believe the secure graph anonymization line of work holds the most general-purpose potential. Previous state-of-the-art secure network flow techniques required to consider a complete graph with $n \cdot (n-1)$ edges to preserve privacy over the network topology. We prove that graph anonymization techniques allow parties to operate on a graph in which the m viable and perturbed edges are revealed, significantly improving the efficiency of any following graph-based optimization algorithm. A possible future line of work is to handle secure anonymization algorithms that do not compromise the utility of a graph, reducing the optimality of its solution.

The protocol achieves security in a client-to-server setting, in which the computation is split across three servers. We choose these servers as three parties that have conflicting interests with each other, such that a collusion between them is unlikely. To further increase the security of the system, the servers might wrap their computation inside a Trusted Execution Environment (TEE): to maliciously assemble their shares; the servers must first break the TEE security. More far out in the future, the whole computation can be wrapped into an indistinguishable program, able to return the outputs of the program to the designated firm without requiring any further interaction.

9.1 Acknowledgement

We thank Tomaž Fleischmann for his detailed and passionate explanations regarding MTCS. We thank Ying Tong Lai, Andrew Miller, and Mikerah Quintyne-Collins for the helpful discussions on the design of the MPC protocol.

A Secure Network Simplex Variables Table

Table 3 describes the variables used in the Secure Network Simplex algorithm described in §6. The variables in the table are organized in logical groups:

- Graph structure variables: Define the topology and size of the network
- Edge properties: Describe characteristics of edges like cost, capacity, and flow
- Node properties: Track node-specific information like supply/demand and potentials
- Spanning tree (ST) data structures: Maintain the tree structure during iterations
- Algorithm-specific variables: Used in pivot operations and cycle management
- Support variables: Facilitate updates to tree structure and node potentials

References

1. Agarwal, A.: Privacy Preserving Decentralized Netting. `https://www.youtube.com/watch?v=kvD_ERfpUEI` (2022), accessed: 2024-09-17

Table 3: Variable Descriptions for Secure Network Simplex Implementation

Variable	**Size**	**Description**
$\mathcal{V}$	n	Set of vertices in the input graph
$\mathcal{V}'$	$n+1$	Set of vertices including artificial root node r
$\mathcal{E}$	m	Set of edges in the input graph
$\mathcal{E}'$	$m+n$	Set of edges including artificial edges from nodes to root
c	$m+n$ [1]	Public cost associated with each edge
$[u]$	$m+n$ [1]	Private capacity constraints for each edge
$[x]$	$m+n$	Current flow value for each edge
$[b]$	$n+1$ [2]	Private supply/demand for each node
$[\pi]$	$n+1$	Node potentials used for reduced cost calculations
$[par]$	$n+1$	Parent node for each vertex in spanning tree
$[pred]$	$n+1$	Tuple (idx, o) of predecessor edge index and orientation bit
$[children]$	$(n+1)\times(n+1)$	Binary matrix indicating parent-child relationships
$[t]$	$m+n$	Binary vector indicating if edge is in spanning tree
$[lu]$	$m+n$	State of non-tree edges (-1 for lower, 1 for upper)
w	1	Public number of pivot iterations to perform
$[e_{in}]$	4	(idx_{in}, k, l, d_{in}) describing entering edge and orientation
$[LCA]$	1	Lowest common ancestor node in newly formed cycle
$[\delta]$	1	Maximum flow that can be pushed along cycle
$[f]$	$m+n$	Direction of flow updates (-1, 0, or 1) for each edge
$[u_{out}]$	1	Lower endpoint node of leaving edge
$[side]$	1	Binary indicator for which side of cycle contains leaving edge
$[clone]$	$(n+1)\times(n+1)$	Working copy of children matrix during updates
$[path]$	$n+1$	Stores nodes in path during tree structure updates
$[y]$	$n+1$	Maps nodes to their new parents during tree updates
$[e]$	$n+1$	Binary vector tracking nodes requiring potential updates
$[q]$	$n+1$	Queue for processing children during potential updates
$[next]$	$n+1$	Binary vector indicating next child node to process
$[sums]$	$n+1$	Tracks sum of children for each node during updates
$[\Delta\pi]$	1	Change in potential to apply to affected subtree

2. Agarwal, A., De Caro, A., Miller, A.: Short paper: Privacy preserving decentralized netting. In: International Conference on Financial Cryptography and Data Security. pp. 280–298. Springer (2022)
3. Ahuja, R.K., Magnanti, T.L., Orlin, J.B.: Network Flows: Theory, Algorithms, and Applications. Prentice-Hall, Inc., Englewood Cliffs, NJ (1993)
4. AJPES: Poročila o Rezultatih VečStranskega Pobota, Agency of the Republic of Slovenia for Public Legal Records and Related Services. `https://www.ajpes.si/Bonitetne_storitve/Vecstranski_pobot/Porocila#b671` (2002-2024), accessed: 2024-09-13
5. Aly, A., Cuvelier, E., Mawet, S., Pereira, O., Van Vyve, M.: Securely solving simple combinatorial graph problems. In: Financial Cryptography and Data Security: 17th International Conference, FC 2013, Okinawa, Japan, April 1-5, 2013, Revised Selected Papers 17. pp. 239–257. Springer (2013)
6. Aly, A., Van Vyve, M.: Securely solving classical network flow problems. In: Information Security and Cryptology-ICISC 2014: 17th International Conference, Seoul, South Korea, December 3-5, 2014, Revised Selected Papers 17. pp. 205–221. Springer (2015)
7. Aly, A., Van Vyve, M.: Practically efficient secure single-commodity multi-market auctions. In: International Conference on Financial Cryptography and Data Security. pp. 110–129. Springer (2016)
8. Araki, T., Barak, A., Furukawa, J., Lichter, T., Lindell, Y., Nof, A., Ohara, K., Watzman, A., Weinstein, O.: Optimized honest-majority mpc for malicious adversaries—breaking the 1 billion-gate per second barrier. In: 2017 IEEE Symposium on Security and Privacy (SP). pp. 843–862. IEEE (2017)
9. Asharov, G., Hamada, K., Ikarashi, D., Kikuchi, R., Nof, A., Pinkas, B., Takahashi, K., Tomida, J.: Efficient secure three-party sorting with applications to data analysis and heavy hitters. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. pp. 125–138 (2022)
10. Atapoor, S., Smart, N.P., Alaoui, Y.T.: Private liquidity matching using mpc. In: Cryptographers' Track at the RSA Conference. pp. 96–119. Springer (2022)
11. Avarikioti, Z., Pietrzak, K., Salem, I., Schmid, S., Tiwari, S., Yeo, M.: Hide & seek: Privacy-preserving rebalancing on payment channel networks. In: International Conference on Financial Cryptography and Data Security. pp. 358–373. Springer (2022)
12. Backstrom, L., Dwork, C., Kleinberg, J.: Wherefore art thou r3579x? anonymized social networks, hidden patterns, and structural steganography. In: Proceedings of the 16th international conference on World Wide Web. pp. 181–190 (2007)
13. Baldimtsi, F., Kiayias, A., Zacharias, T., Zhang, B.: Crowd verifiable zero-knowledge and end-to-end verifiable multiparty computation. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 717–748. Springer (2020)
14. Berloco, C., De Francisci Morales, G., Frassineti, D., Greco, G., Kumarasinghe, H., Lamieri, M., Massaro, E., Miola, A., Yang, S.: Predicting corporate credit risk: Network contagion via trade credit. PLoS One **16**(4), e0250115 (2021)
15. Blanton, M., Steele, A., Alisagari, M.: Data-oblivious graph algorithms for secure computation and outsourcing. In: Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security. pp. 207–218 (2013)
16. Bogetoft, P., Christensen, D.L., Damgård, I., Geisler, M., Jakobsen, T., Krøigaard, M., Nielsen, J.D., Nielsen, J.B., Nielsen, K., Pagter, J., et al.: Secure multiparty computation goes live. In: International Conference on Financial Cryptography and Data Security. pp. 325–343. Springer (2009)

17. Bonchi, F., Gionis, A., Tassa, T.: Identity obfuscation in graphs through the information theoretic lens. Information Sciences **275**, 232–256 (2014)
18. Brunovsky: [tutorial] network simplex. `https://codeforces.com/blog/entry/94190` (2024), accessed: 2024-10-10
19. Cao, S., Yuan, Y., De Caro, A., Nandakumar, K., Elkhiyaoui, K., Hu, Y.: Decentralized privacy-preserving netting protocol on blockchain for payment systems. In: Financial Cryptography and Data Security: 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10–14, 2020 Revised Selected Papers 24. pp. 137–155. Springer (2020)
20. Casas-Roma, J.: An evaluation of vertex and edge modification techniques for privacy-preserving on graphs. Journal of Ambient Intelligence and Humanized Computing **14**(11), 15109–15125 (2023)
21. Casas-Roma, J., Herrera-Joancomartí, J., Torra, V.: A survey of graph-modification techniques for privacy-preserving on networks. Artificial Intelligence Review **47**, 341–366 (2017)
22. CoFi, I.S.: Frequently Asked Questions. `https://cofi.informal.systems/FAQ` (2024), accessed: 2024-09-13
23. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to algorithms. MIT press (2022)
24. Cramer, R., Damgård, I., Escudero, D., Scholl, P., Xing, C.: Spd: efficient mpc mod for dishonest majority. In: Annual International Cryptology Conference. pp. 769–798. Springer (2018)
25. Dalskov, A., Escudero, D., Keller, M.: Fantastic four:{Honest-Majority}{Four-Party} secure computation with malicious security. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 2183–2200 (2021)
26. Damgård, I., Damgård, K., Nielsen, K., Nordholt, P.S., Toft, T.: Confidential benchmarking based on multiparty computation. In: International Conference on Financial Cryptography and Data Security. pp. 169–187. Springer (2016)
27. Damgård, I., Nielsen, J.B.: Universally composable efficient multiparty computation from threshold homomorphic encryption. In: Annual international cryptology conference. pp. 247–264. Springer (2003)
28. Fleischman, T., Dini, P.: Mathematical foundations for balancing the payment system in the trade credit market. Journal of Risk and Financial Management **14**(9), 452 (2021)
29. Fleischman, T., Dini, P., Littera, G.: Liquidity-saving through obligation-clearing and mutual credit: An effective monetary innovation for smes in times of crisis. Journal of Risk and Financial Management **13**(12), 295 (2020)
30. Gaffeo, E., Gobbi, L., Molinari, M.: The economics of netting in financial networks. Journal of Economic Interaction and Coordination **14**, 595–622 (2019)
31. Galal, H.S., Youssef, A.M.: Privacy preserving netting protocol for inter-bank payments. In: Data Privacy Management, Cryptocurrencies and Blockchain Technology: ESORICS 2020 International Workshops, DPM 2020 and CBT 2020, Guildford, UK, September 17–18, 2020, Revised Selected Papers 15. pp. 319–334. Springer (2020)
32. Goldberg, A., Tarjan, R.: Solving minimum-cost flow problems by successive approximation. In: Proceedings of the nineteenth annual ACM symposium on Theory of computing. pp. 7–18 (1987)
33. Goldberg, A.V., Tarjan, R.E.: Finding minimum-cost circulations by canceling negative cycles. Journal of the ACM (JACM) **36**(4), 873–886 (1989)
34. Hay, M., Miklau, G., Jensen, D.D., Weis, P., Srivastava, S.: Anonymizing social networks. In: arXiv (2007)

35. Keller, M., Orsini, E., Scholl, P.: Mascot: faster malicious arithmetic secure computation with oblivious transfer. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 830–842 (2016)
36. Keller, M., Scholl, P.: Efficient, oblivious data structures for mpc. In: Advances in Cryptology–ASIACRYPT 2014: 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, ROC, December 7-11, 2014, Proceedings, Part II 20. pp. 506–525. Springer (2014)
37. Király, Z., Kovács, P.: Efficient implementations of minimum-cost flow algorithms. arXiv preprint arXiv:1207.6381 (2012)
38. Lindell, Y.: Secure multiparty computation. Communications of the ACM **64**(1), 86–96 (2020)
39. Orlin, J.B.: A polynomial time primal network simplex algorithm for minimum cost flows. Mathematical Programming **78**(2), 109–129 (Aug 1997). https://doi.org/10.1007/BF02614365, `https://doi.org/10.1007/BF02614365`
40. Righini, G.: The network simplex algorithm. `https://homes.di.unimi.it/righini/Didattica/OttimizzazioneCombinatoria/MaterialeOC/9b%20-%20NetworkSimplex.pdf`, combinatorial Optimization course material, Università degli Studi di Milano
41. SURS: GDP and National Accounts. Statistical Office of the Republic of Slovenia. `https://www.stat.si/statweb` (2024), accessed: 2024-09-13
42. Tamura, K., Miura, W., Takayasu, M., Takayasu, H., Kitajima, S., Goto, H.: Estimation of flux between interacting nodes on huge inter-firm networks. In: International Journal of Modern Physics: Conference Series. vol. 16, pp. 93–104. World Scientific (2012)
43. Toft, T.: Solving linear programs using multiparty computation. In: International Conference on Financial Cryptography and Data Security. pp. 90–107. Springer (2009)
44. Wayne, K.D.: A polynomial combinatorial algorithm for generalized minimum cost flow. In: Proceedings of the thirty-first annual ACM symposium on Theory of computing. pp. 11–18 (1999)
45. Wüller, S., Breuer, M., Meyer, U., Wetzel, S.: Privacy-preserving trade chain detection. In: Data Privacy Management, Cryptocurrencies and Blockchain Technology: ESORICS 2018 International Workshops, DPM 2018 and CBT 2018, Barcelona, Spain, September 6-7, 2018, Proceedings 13. pp. 373–388. Springer (2018)
46. Zhou, B., Pei, J., Luk, W.: A brief survey on anonymization techniques for privacy preserving publishing of social network data. ACM Sigkdd Explorations Newsletter **10**(2), 12–22 (2008)