Private Liquidity Matching using MPC

Shahla Atapoor[1] , Nigel P. Smart[1,2] , and Younes Talibi Alaoui[1]

[1] imec-COSIC, KU Leuven, Leuven, Belgium.
[2] University of Bristol, Bristol, UK.
Shahla.Atapoor@kuleuven.be, nigel.smart@kuleuven.be, younes.talibialaoui@kuleuven.be

Abstract. Many central banks, as well as blockchain systems, are looking into distributed versions of interbank payment systems, in particular the netting procedure. When executed in a distributed manner this presents a number of privacy problems. This paper studies a privacy preserving netting protocol to solve the gridlock resolution problem in such Real Time Gross Settlement systems. Our solution utilizes Multi-party Computation and is implemented in the SCALE MAMBA system, using Shamir secret sharing scheme over three parties in an actively secure manner. Our experiments show that, even for large throughput systems, such a privacy preserving operation is often feasible.

1 Introduction

Real Time Gross Settlement systems (RTGS systems for short) consist of fund transfer systems that permit banks to exchange transactions, where debiting the funds from the account of the source to credit them to the account of the destination (Settlement) is performed immediately (Real Time) if enough funds are available from the account of the source. The settlement of transactions happens individually (Gross), that is, the funds move (in electronical form as opposed to physical form) every time a transaction is performed.

RTGS systems are nowadays widely adopted as they reduce the risks associated to high-value payment settlements between participants. In particular, the use of RTGS systems mitigates the effects caused by a failing bank not being able to settle its payments, as opposed to netting systems where the settlements between banks take place at the end of the day. However, the immediate settlement of transactions increases the liquidity requirements for the participants, and as a consequence, banks will be exposed to the risk of not being able to settle transactions due to the lack of liquidity, and therefore settling their transactions will be postponed. This delay comes with a cost for both sources and destinations, especially for time-critical payments if they are not settled in due time. Moreover, such disturbances in the system may lead to *gridlock* situations, where basically a set of transactions cannot be settled as each transaction is waiting for another one to be settled (See Section 2).

To address such a situation, the parties involved need to inject more liquidity into the system, but this comes with a cost if the additional liquidity was borrowed. Alternatively, *Liquidity Saving Mechanisms* (LSM for short) [GS10] can be employed, such as *multilateral netting*, which consists of simultaneously offsetting multiple transactions, to result in a single net position for the participants involved, which will eventually permit to unblock some of the pending transactions if net positions of the respective sources are positive. This process is generally carried out by a central entity, as RTGS systems are traditionally managed by the central banks of countries. That is, each country has its own RTGS system, but we may also have a shared RTGS system between more than one country, as the Target 2 system [ECB] operated by the Eurosystem.

Running the RTGS requires the central entity in charge to have sight over the balances of the participants as well as payments instructions, which includes the source, destination, and the

amount being transferred. Thus, this entity is trusted to preserve these information confidential, and as a result, participants might be skeptical about submitting all their transactions to the RTGS. In order to eliminate this need of having a trusted entity, one could *decentralize* the RTGS system, by permitting the participants to exchange transactions, without having to disclose them to a central entity.

However, designing such an RTGS will arise two main challenges, namely how to guarantee (i) Correctness: while settling a transaction, the amount debited from the source is the same as the amount transferred to the destination; (ii) Fairness: The LSM process implemented should not favor a participant over the others. These two requirements are easy to achieve in a centralized system, as all the transactions are visible to the trusted central entity in charge of the RTGS, but in a decentralized payment system, one would expect that these tasks become a burden.

In this paper, we propose a *Multi-Party Computation* (MPC for short) based solution to perform the liquidity optimization for decentralized RTGS systems. In particular, we show that the task of managing the RTGS system can be assigned to a set of p entities, that will obliviously settle the incoming transactions, and update the (private) balances of the parties involved. The payments instructions and balances will remain hidden as long as those entities do not collude.

We also show that those entities will be capable of obliviously running a multilateral netting process. For this purpose we distributed the gridlock resolution algorithm of [BS01] (See Section 2), providing three main versions : (i) sodoGR where the amount being transferred in transactions is held secret, while the source and the destination are revealed to the p entities; (ii) sodsGR where the amount and the source are secret whereas the destination is revealed to the p entities; (iii) ssdsGR where the source, destination, and the amount are all held secret;

We implemented our algorithms using the SCALE-MAMBA framework [ACK+21], with $p = 3$, and we simulated an RTGS system running over time using the three aforementioned versions of gridlock resolution. For our simulation we drew transactions drawn from a distribution similar to what is seen in real life. For one variant of our clearing methodology we found that, in the sodoGR and sodsGR cases, that transactions could be cleared in effectively real time, with no delay due to the secure nature of the processing. Only in the case of ssdsGR that we find a significant delay being introduced, which depends on the number of banks, the number of transactions per hour, and the overall liquidity within the system.

Prior usage of MPC in financial applications has mainly focused in auctions, such as [BDJ+06], [BCD+08], [BCD+09], [PRST06] [BHSR19] for one shot auctions, and [CST18] [CSA20] [AHBPV20] for auctions running in Dark Markets. MPC was also used for privacy preserving financial data analysis, such as [BTW11] to conduct statistics over the performance of companies throughout the year, or to compute the systemic risk between financial institutions such as in [AKL11] and [HFT20]. Also [SvA+18] and [CSA21] used MPC for detecting fraud between financial institutions, and [BP20] used MPC for privacy preserving federated learning for financial applications.

Driven by the motivation of removing the central entity, various blockchain projects have also investigated the feasibility of a decentralized RTGS using blockchain combined with smart contracts, for example the Jasper [CaSHMM], Ubin [oS] and Stella [Eur] projects, In [WXF+18] the authors provide a blockchain based solution, however it still relied on a central entity that checks the correctness of the multilateral netting process, besides, while the source, destination, and amount of every transaction are hidden, the net positions during the multilateral netting process are revealed. Moreover, the multilateral process being an expensive operation, is only executed whenever a pending transaction approaches its due time (if any).
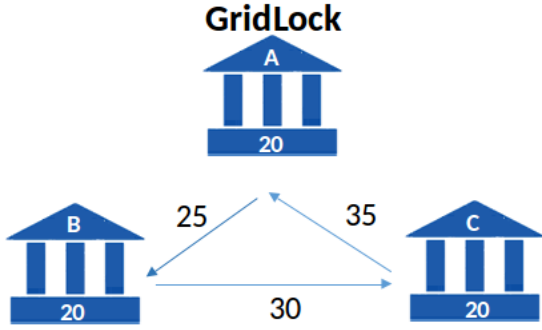
Fig. 1. The banks do not have enough liquidity to settle their transactions. However, a multilateral netting will result in positive net positions to all banks.
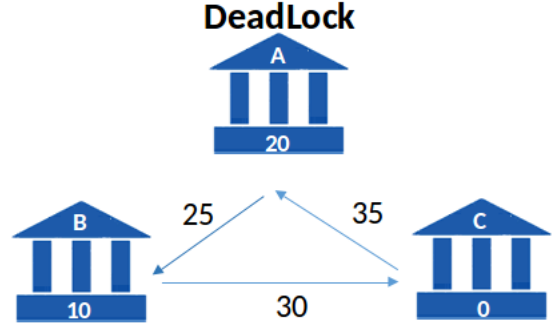

Fig. 2. The banks do not have enough liquidity to settle their transactions, and even a multilateral netting will not result in positive net positions for all banks

More recently, another blockchain-based work was proposed in [CYC+20], which is the work most closely related to ours. In [CYC+20] homomorphic Pedersen commitments and Zero-knowledge proofs are used to remove the need of a central entity, as well as the need to reveal the net positions during the multilateral netting process. However, the basic protocol reveals the source and destination of every transaction. Hiding them is possible, but it would induce a factor of n to the cost of the protocol where n is the number of participants in the RTGS. The paper [CYC+20] does not provide any concrete runtimes, but simply gives execution times of the underlying cryptographic primitives. In comparison to our algorithm, with 100 banks and 9000 transactions, and a liquidity measure of $\beta = 0.1$ (see Section 4) one would need to execute approximately 1750 gridlock computations per hour. Our solution could achieve this with a delay of zero seconds, assuming we only secure the values and not the source and destination addresses. The algorithm in [CYC+20] would require over 64 minutes to process the transactions, and thus is not real time with these relatively modest transaction levels.

2 Preliminaries

2.1 The Gridlock Resolution Problem

For a pictorial description of the problem see Figure 2. The problem is to decide on a multilateral netting which will result in transactions being settled, when this is possible. Such a multilateral netting may not be locally determinable by an individual entity, as it requires knowledge of the position of other entities and the transactions being processed. This problem can be modeled as a discrete optimization problem. In order to introduce it, we will base our definitions on [BS01] with some simplifications for ease of exposition.

Let n denote the number of banks using the RTGS system, with each bank i having an initial balance $B^i \geq 0$. A transaction is of the form $t = [s, a, r]$, where $s \in \{1, \ldots n\}$ is the source of t, $r \in \{1, \ldots n\}$ is the destination of t, and $a > 0$ is the amount of money that s is sending to r. For a transaction $t = [s, a, r]$, if $B^s \geq a$, i.e. the sending bank has a large enough balance to cover the transaction, the transaction is executed right away, otherwise, the transaction t is added to a queue Q. The goal of gridlock resolution is to deal with the queue Q that is formed in this way.

3

We define $\mathsf{Sen}_U \subseteq \{1, \ldots, n\}$ to be the set of sources of transactions in a set $U \subseteq Q$. We also denote by U^i the set of the transactions in U coming from bank i, i.e. $U^i = [t = (s, a, r) \in U \text{ s.t. } s = i]$, and let us denote by m the size of Q. Let S_U^i and R_U^i denote respectively the amounts sent and received by bank i after running simultaneously a set of U transactions. Let B_U^i denote the balance of this bank after running these transactions. i.e., $B_U^i = B^i - S_U^i + R_U^i$.

The key issues for the queue are those of gridlock and deadlock. A gridlock is where there is a subset of transactions which can be executed, the term 'gridlock' is used here to distinguish it from the situation where the transactions clear without resorting to a queue.

Definition 2.1 (Gridlock). *A gridlock is a situation where among the transactions hanging in Q, there exists $U \subset Q$ and $U \neq \emptyset$, such that:*

$$\forall i \in \mathsf{Sen}_U, \ B_U^i \geq 0 \tag{1}$$

$$\forall i \in \mathsf{Sen}_U, \ \nexists t' \in Q^i \setminus U^i \quad \text{such that} \quad t < t' \tag{2}$$

A deadlock is where no transactions can be executed, with the current queue state.

Definition 2.2 (Deadlock). *A deadlock is a situation where $\nexists U \subset Q$ that satisfies the conditions of 2.1*

The gridlock resolution problem is to find a subset of transactions in the queue which can be executed.

Definition 2.3 (The gridlock resolution problem). *The gridlock resolution problem consists of finding $\max_{U \subset Q} |U|$, such that (1) and (2) from 2.1 hold*

This problem is NP complete, and thus one needs to use approximate algorithms in order to solve it, such as the ones introduced in [SD06] and [GJL98]. However, the problem is not NP-complete if the transactions are augmented with a strict ordering $<$ of execution for every bank, and one can find an optimal solution in polynomial time. This ordering can be First-In-First-Out, that is, among the transactions hanging issued by bank i, no transaction can be addressed prior to the execution of all transactions that arrived before it.

The Gridlock Resolution Algorithm: The algorithm for gridlock resolution GR [BS01] proceeds in the following steps, with the formal definition of the algorithm being given in Figure 3. We first empty the queue Q by moving all the transactions from Q to U. We then call a sub-routine which computes the balances resulting from the set U. Thus we compute the amounts sent by the banks, i.e. $[S_U^1, \ldots, S_U^n]$, as well as the amounts received by the banks, i.e. $[R_U^1, \ldots, R_U^n]$. From this we can compute the balances as $[B_U^1, \ldots, B_U^n]$, from $B_U^i \leftarrow B^i + S_U^i - R_U^i$.

At this point, if all the balances of Sen_U are positive, the GR algorithm terminates with the set U. Otherwise, the last transaction for all sources i such that the balance B_U^i is negative is moved from U to the queue Q[3]. The balances are recomputed for this new set U, and this process is repeated. If at some point the algorithm terminates with a set $U \neq \emptyset$, the transactions in U are executed, and the actual balances B^i are updated. If however $U = \emptyset$ then we have reached deadlock and we need to wait for more transactions to come in before running the GR algorithm again.

---

[3] The algorithm from [BS01] removes just one transaction of a source with negative balance at this point, but it is equivalent to remove one transaction from each source which has a negative balance.

<div style="border:1px solid">

The GR algorithm

Algorithm: Balance(U, B)**:**
    On input of a list of transactions U and the current set of balances B this computes the new balances B_U^i for all $i \in [1, \ldots, n]$.
    (1) For $i \in [1, \ldots, n]$
        (I) $S^i \leftarrow \sum_{t=[s,a,r] \in U^i} a$
        (II) $R^i \leftarrow \sum_{t=[s,a,r] \in U \, s.t. \ r=i} a$
        (III) $B_U^i \leftarrow B^i - S^i + R^i$
    (2) Output $[B_U^1, \ldots, B_U^n]$

Algorithm : GR(Q):
    On input Q this algorithm finds is a solution $U \subset Q$ to the gridlock resolution problem, or it returns Deadlock.
    (1) $U \leftarrow Q$ and $Q \leftarrow \emptyset$
    (2) While $U \neq \emptyset$ do
        (I) $B_U \leftarrow$ Balance(U, B)
        (II) If all the balances in B_U are positive return U.
        (III) Remove from U the last transaction sent by all of the banks for which $B_U^i < 0$, and place these transactions in Q
    (3) Return Deadlock

</div>

Figure 3. The GR algorithm

2.2 Multiparty Computation (MPC)

MPC is a family of cryptographic techniques that allow a set of parties, to perform computation on their inputs, without having to reveal them. MPC was formally introduced by Yao in [Yao86], and was developed throughout the years. One family of MPC protocols, and the ones we will consider in this paper, are those based on secret sharing schemes. In these protocols secrets are split among the p parties participating in the protocol, and computation on the secrets is performed through communication between the parties. The function to be evaluated is expressed (to a first order approximation) through a circuit consisting of addition and multiplication gates over a field $\mathbb{F}_q$.

Corruption of parties can be either passive or active. A passive adversary follows the instructions of the protocol, but may try to learn information about other players inputs from its view of the protocol. An active adversary, is a party that may arbitrarily deviate from the protocol. In this work we will be considering protocols which are secure against active adversaries, in the sense that the protocol will abort with overwhelming probability if the adversarial parties deviate from the protocol.

An MPC protocol is also characterized by its adversarial structure, i.e., the coalitions of parties that may maliciously team up. An MPC protocol is said to be secure if the claimed security properties remain, even when a coalition of parties among the ones considered collude. MPC Protocols usually consider threshold adversarial structures, where a threshold t of corrupt parties is considered, that is, any number up to t parties may collude. The threshold considered affects how the MPC protocol is realized. Generally the protocols with honest majority, i.e., $p/2 > t$ are less complex than having a dishonest majority.

Shamir Secret Sharing based MPC: In Shamir Secret Sharing, the secret $x \in \mathbb{F}_q$ is encoded in the constant term of a polynomial $f_x(X)$ of degree t, i.e. $x = f_x(0)$, where t is the threshold

considered. The share of each party i of the secret x is $x_i \in \mathbb{F}_q$, such that $f_x(i) = x_i$. From now on we will denote such a secret shared value x by $\langle x \rangle$.

If $t + 1$ parties combine their shares by interpolation of the polynomial, they can recover the polynomial f_x and thus the secret x. Therefore, the secret sharing requires a honest majority, as otherwise, honest players will risk to not be able to recover the secret.

This so far shows only how to share a secret and how to recombine it. To build an MPC protocol out of Shamir secret sharing, we need to be able to perform computation on the shares. Linear operations, can be performed locally on the shares, so only non-linear operations (such as multiplication) require interaction. The basic methodology for such multiplications is the local-product-reshare-and-recombine method described in [Mau06]. The resulting protocol is a passively secure protocol.

There are many ways of turning such a passively secure protocol into an actively secure one (with abort). We shall focus on a methodology outlined in [KRSW18], which splits the computation into an offline and online phase. The offline phase is where data that does not depend on players inputs is created, and the online phase where the actual computation takes place. The methodology in [KRSW18] provides a faster online computation than other methodologies, which aim to reduce the total amount of computation as opposed to the online time only. The methodology works by having a protocol such as that in [Mau06], where we produce the so-called multiplication triples, which are then consumed in the online phase. Assuming the offline phase has been made actively secure, the online phase can be made actively secure essentially for free.
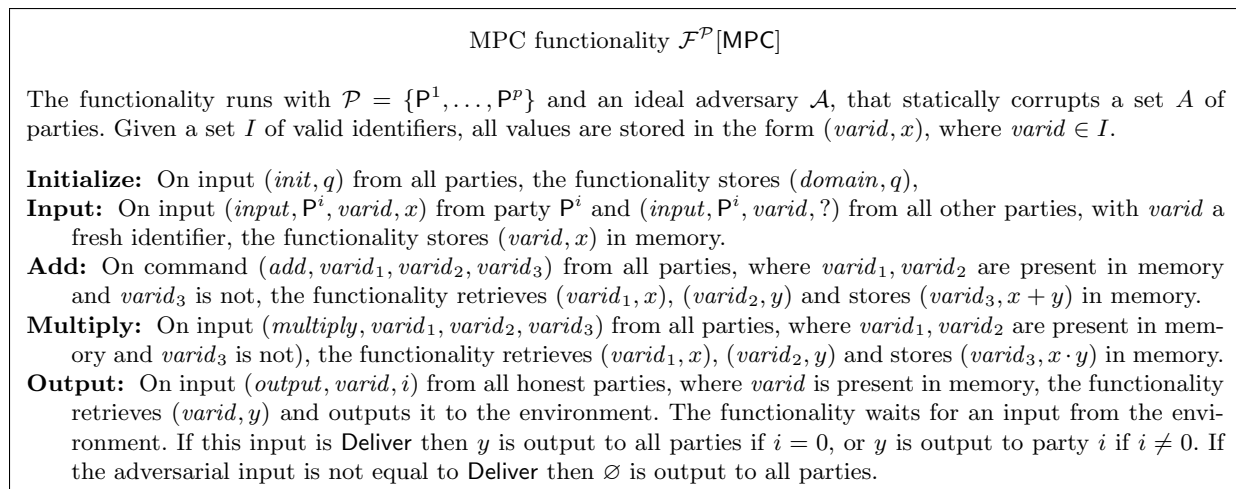

MPC functionality $\mathcal{F}^{\mathcal{P}}[\mathsf{MPC}]$

The functionality runs with $\mathcal{P} = \{\mathsf{P}^1, \ldots, \mathsf{P}^p\}$ and an ideal adversary $\mathcal{A}$, that statically corrupts a set A of parties. Given a set I of valid identifiers, all values are stored in the form $(varid, x)$, where $varid \in I$.

Initialize: On input $(init, q)$ from all parties, the functionality stores $(domain, q)$,
Input: On input $(input, \mathsf{P}^i, varid, x)$ from party P^i and $(input, \mathsf{P}^i, varid, ?)$ from all other parties, with $varid$ a fresh identifier, the functionality stores $(varid, x)$ in memory.
Add: On command $(add, varid_1, varid_2, varid_3)$ from all parties, where $varid_1, varid_2$ are present in memory and $varid_3$ is not, the functionality retrieves $(varid_1, x)$, $(varid_2, y)$ and stores $(varid_3, x + y)$ in memory.
Multiply: On input $(multiply, varid_1, varid_2, varid_3)$ from all parties, where $varid_1, varid_2$ are present in memory and $varid_3$ is not), the functionality retrieves $(varid_1, x)$, $(varid_2, y)$ and stores $(varid_3, x \cdot y)$ in memory.
Output: On input $(output, varid, i)$ from all honest parties, where $varid$ is present in memory, the functionality retrieves $(varid, y)$ and outputs it to the environment. The functionality waits for an input from the environment. If this input is Deliver then y is output to all parties if $i = 0$, or y is output to party i if $i \neq 0$. If the adversarial input is not equal to Deliver then $\varnothing$ is output to all parties.

Figure 4. MPC functionality $\mathcal{F}^{\mathcal{P}}[\mathsf{MPC}]$

This enables one to produce a standard MPC functionality as in Figure 4. Note, as a shorthand in our protocols we will refer to the addition and multiplication operations as $\langle z \rangle \leftarrow \langle x \rangle + \langle y \rangle$ and $\langle z \rangle \leftarrow \langle x \rangle \cdot \langle y \rangle$, and the output operations as $z \leftarrow \mathsf{Open}(\langle z \rangle)$ for open-to-all. We also define an operation $\mathsf{Open}_E(\langle z \rangle, i)$ which opens the shared value $\langle z \rangle$ to the external bank i (by sending all the players shares of $\langle z \rangle$ to the external bank i.)

For some non-linear operations, and especially the comparison operations which we will make extensive use of, the reduction to additions and multiplications over $\mathbb{F}_q$ is not efficient. Thus in this

case one utilizes special purpose sub-protocols such as those in [Cd10, CS10, DFK$^+$06]. Comparison in the online phase requires multiple rounds of communication, as opposed to multiplication that requires only one round, thus comparison is slower than multiplication. Another sub-protocol which we will be using is a form Oblivious RAM, i.e. a methodology to enable the MPC engine to access a secret location in an array, see [KS14] for more details. At the heart of the Oblivious RAM operation is a Demux function, which takes as input a secret x and a bound L, and outputs a list of bits $b_1, \ldots b_L$, such that $b_i = 1$ if $i = x$, and $b_i = 0$ otherwise.

3 The Gridlock Resolution Algorithm with MPC

Executing the GRP algorithm totally obliviously, i.e. where the MPC engines know neither the amounts, the source addresses of each transactions or the destinations, turns out to be very expensive. Thus as well as examining this situation we also examine two relaxed situations in which both the source and destination addresses are in the clear (i.e. open) (which we call sodoGR), and one in which the source addresses are in the clear, but the destination addresses are not (which we call sodsGR). The fully oblivious version we denote ssdsGR. In all situations we ensure privacy of the amount in the transactions, and privacy of the banks balances.

In Figure 5 we present the algorithm to compute the balances, upon processing a list of payments Q, in our three different scenarios. The algorithms, make use of the Demux operation. Recall $\langle \mathbf{v} \rangle \leftarrow \mathsf{Demux}(\langle r \rangle, n)$, where we are guaranteed that $1 \leq r \leq n$, produces a vector $\langle \mathbf{v} \rangle$ of size n such that each entry is a sharing of zero, apart from entry r where we have a sharing of one. The i-th element of $\langle \mathbf{v} \rangle$ will be denoted by $\langle \mathbf{v}_i \rangle$.

In Figure 7 we present the algorithm for executing the Gridlock Resolution algorithm in our three cases. This utilizes the algorithm for computing balances from Figure 5, as well as the algorithm, in Figure 6, which notifies the banks of their completed transactions. In the following paragraphs we highlight aspects of these algorithms in the three different situations.

Sources and Destinations are Open (sodoGR): For this version, computing the receipts and payments is straightforward. That is, we know the source and destination of every transaction, thus for every bank of Sen_Q, we know the amount of each transaction we need to add or subtract to its balance. Therefore, the algorithm for updating the balances (Figure 5) are relatively cheap. However, we face two main obstacles for this version: (1) How to determine whether we have a solution for the problem without leaking which banks have negative balances? (2) In the case where we still do not have a solution, how can we remove transactions without leaking which ones, and without leaking their corresponding banks?

To address the first obstacle, in Figure 7 we first compare the balances of the banks in an oblivious manner, by performing in step 5.II.A a secure comparison of all balances of Sen_Q, which will result in secret values $\langle h^i \rangle$, that are equal to one if the corresponding balance is negative, or equal to zero otherwise. Then in step 5.IV, we open $\langle z \rangle$ which is the product of $(1 - \langle h^i \rangle)$. Thus z will be one if all the balances are positive and therefore we found a solution, or z will be zero in which case we need to continue removing transactions.

For the second obstacle, basically we need to touch every transaction so as to not leak which ones are being removed. We ensure this using the flags $\langle h^i \rangle$, which guarantee that we will be modifying the indicators of only the banks with negative balances,

<div style="border">

<div align="center">Algorithms for Balances</div>

$\mathsf{Demux}_{\mathsf{sodsGR}}(Q)$

   On input of a list of transactions Q this produces the demux flags C for the destination addresses when the source addresses are in the clear

   (1) $C \leftarrow [\,]$.

   (2) For $t = [s, \langle a \rangle, \langle r \rangle]$ in Q :

       (I) $\langle \mathbf{c}^t \rangle \leftarrow \mathsf{Demux}(\langle r \rangle, n)$.

       (II) Append C by $\langle \mathbf{c}^t \rangle$.

   (3) Output C.

$\mathsf{Demux}_{\mathsf{ssdsGR}}(Q)$

   On input of a list of transactions Q this produces the demux flags for the recipient addresses C and the source addresses W

   (1) $C \leftarrow [\,]$, $W \leftarrow [\,]$.

   (2) For $t = [\langle s \rangle, \langle a \rangle, \langle r \rangle]$ in Q:

       (I) $\langle \mathbf{c}^t \rangle \leftarrow \mathsf{Demux}(\langle r \rangle, n)$.

       (II) $\langle \mathbf{w}^t \rangle \leftarrow \mathsf{Demux}(\langle s \rangle, n)$.

       (III) Append C by $\langle \mathbf{c}^t \rangle$ and C by $\langle \mathbf{w}^t \rangle$.

   (3) Output (C, W).

$\mathsf{Balance}([\langle B^i \rangle]_{i=1}^n, Q, X, C, W, \mathsf{type})$

This algorithm updates the balances $\langle B^i \rangle$ for every bank i, given a set of transactions $U \subset Q$ and the operation type, $\mathsf{type} = \mathsf{sodoGR}, \mathsf{sodsGR}, \mathsf{ssdsGR}$. The set U is oblivious to the algorithm and determined by an indicator set $X = [\langle x_t \rangle]_{t \in Q}$, such that $x_t = 1$ if and only if $t \in U$, otherwise $x_t = 0$. The sets $C = [\mathbf{c}^t]_{t \in Q}$ and $W = [\langle \mathbf{w}^t \rangle]_{t \in Q}$ are derived from the Demux operations above, when needed.

(1) If $\mathsf{type} \neq \mathsf{ssdsGR}$

   (I) $\langle B_U^i \rangle \leftarrow \langle B^i \rangle$ for all $i \in [1, \ldots, n]$.

   (II) For $i \in [1, \ldots, n]$:

       (A) $\langle S^i \rangle \leftarrow \sum_{t=(i, \langle a \rangle, *) \in Q^i} \langle a \rangle \cdot \langle x_t \rangle$.

       (B) If $\mathsf{type} = \mathsf{sodoGR}$ then $\langle R^i \rangle \leftarrow \sum_{t=(s, \langle a \rangle, r) \in Q, r=i} \langle a \rangle \cdot \langle x_t \rangle$.

       (C) else $\langle R^i \rangle \leftarrow \sum_{t=(s, \langle a \rangle, \langle r \rangle) \in Q} \langle a \rangle \cdot \langle x_t \rangle \cdot \langle \mathbf{c}_i^t \rangle$.

       (D) $\langle B_U^i \rangle \leftarrow \langle B^i \rangle - \langle S^i \rangle + \langle R^i \rangle$.

(2) else

   (I) For $i \in [1, \ldots, n]$:

       (A) $\langle S^i \rangle \leftarrow \sum_{t=(s, \langle a \rangle, \langle r \rangle) \in Q} \langle a \rangle \cdot \langle x_t \rangle \cdot \langle \mathbf{w}_i^t \rangle$.

       (B) $\langle R^i \rangle \leftarrow \sum_{t=(s, \langle a \rangle, \langle r \rangle) \in Q} \langle a \rangle \cdot \langle x_t \rangle \cdot \langle \mathbf{c}_i^t \rangle$.

       (C) $\langle B_U^i \rangle \leftarrow \langle B^i \rangle - \langle S^i \rangle + \langle R^i \rangle$.

(3) Return $[\langle B_U^i \rangle]_{i=1}^n$.

</div>

Figure 5. Algorithms for Balances

<div style="border:1px solid">

Method to Notify Participants of Executed Transactions

$\mathsf{Notify}(X, Q, C, W, \mathsf{type})$

This algorithm notifies the participants about the executed transactions they sent/received. On input Q contains a set of transactions, X specifies (in the clear) the transactions in Q that were executed (signaled by a one bit), the sets C and W are derived from the Demux operation in Figure 5.

(1) If $\mathsf{type} = \mathsf{sodoGR}$
   (I) For $t = [s, \langle a \rangle, r] \in Q$ such that $x_t = 1$.
      (A) Notify s and r about t and execute $\mathsf{Open}_E(\langle a \rangle, r)$.
(2) If $\mathsf{type} = \mathsf{sodsGR}$
   (I) For $t = [s, \langle a \rangle, \langle r \rangle] \in Q$ such that $x_t = 1$
      (A) Notify s about t, who can then update bank r about (s, a, r).
(3) If $\mathsf{type} = \mathsf{ssdsGR}$
   (I) For $i \in [1, \ldots, n]$ and $t = [\langle s \rangle, \langle a \rangle, \langle r \rangle] \in Q$ such that $x_t = 1$
      (A) Execute $\mathsf{Open}_E(\langle \mathbf{w}_i^t \rangle \cdot \langle s \rangle, i)$, $\mathsf{Open}_E(\langle \mathbf{w}_i^t \rangle \cdot \langle a \rangle, i)$, $\mathsf{Open}_E(\langle \mathbf{w}_i^t \rangle \cdot \langle r \rangle, i)$, $\mathsf{Open}_E(\langle \mathbf{c}_i^t \rangle \cdot \langle s \rangle, i)$, $\mathsf{Open}_E(\langle \mathbf{c}_i^t \rangle \cdot \langle a \rangle, i)$, and $\mathsf{Open}_E(\langle \mathbf{c}_i^t \rangle \cdot \langle r \rangle, i)$.

</div>

Figure 6. Method to Notify Participants of Executed Transactions

Then in order to not reveal which transaction was removed for these banks, we use the indicators x_{t_j} to act as flags. Namely, step 5.VI.A.ii guarantees that we are setting to zero only the last indicator in the queue that has the value one, for the banks in question, i.e., the last transaction submitted to the RTGS among the ones that are still considered for the gridlock resolution. Note that within this step, we could have simply conducted secure comparisons in order to determine which indicator to modify. However, we implemented it as such purely for performance purposes, as multiplications are cheaper than comparisons. In order to detect a deadlock, we compute the product of $(1 - \langle x_t \rangle)$ in 5.VI.B and we open it, which will be equal to one if a deadlock occurs.

Finally, once a set of transactions are executed, we notify the corresponding sources and destinations using the algorithm Notify. For every transaction executed $t = [s, \langle a \rangle, r]$, this algorithm notifies s about the execution of transaction t, and notifies r of receiving a transaction by sending him the tuple (s, a). The amount a here was opened to r by having the p entities send their respective shares of $\langle a \rangle$ to him, who will reconstruct a from these shares. This way only r will learn a while the entities will not.

Sources are Open and Destinations are Secret (sodsGR): For this version, computing the payments needed by a party is straightforward as the sources of transactions are open, but computing the corresponding receipts we need to apply a Demux operation to compute the balances in Figure 5. This is done only once at the beginning of the algorithm, at the expense of having to store the vectors C in memory all along the computation. The rest of the computation proceeds as in the case sodoGR described above.

To notify participants about the execution of their transactions, we again use Notify. For every transaction executed $t = [s, \langle a \rangle, \langle r \rangle]$, this algorithm, in this case, notifies s of the execution of t, who can update r about the amount sent a.

Sources and Destinations are Secret (ssdsGR): This version is more challenging to address compared to the previous versions as we keep all the sources and destinations secret, thus we have to deal with some challenges. The alteration to compute the balances is similar to the case sodsGR. However, given that the sources are hidden to the system, the Gridlock Resolution algorithm needs

9

to update all the n balances when looking for a solution, as opposed to only the balances of the sources in sodoGR and sodsGR, which is quite expansive.

The fact that the sources and destinations are hidden obliged us to radically change our strategy to remove transactions with negative balances. In fact, obliviously removing the last transaction of *all* sources which have a negative balance is expensive, as it would require $n \cdot m$ equality checks; where n is the number of banks and m is the number of transactions in Q. Therefore, for this version we remove only one transaction at each iteration, as this requires only m equality checks. To determine which transaction to remove, we compute the flags $\langle h_i \rangle$ in a similar way to sodoGR and sodsGR, then we compute the flags $\langle f_i \rangle$ in step 5.III.B.ii, which are all equal to zero, except for the first source who has a negative balance, in which case it is equal to one. These flags will be used to determine the sender $\langle \text{Sender} \rangle$ for which we remove the last transaction. Removing this last transaction is done in step 5.VII.C where we iterate over all the transactions starting from the end and going backward, so as to switch the first encountered indicator x_j equal to one corresponding to a transaction issued by source $s_j = \text{Sender}$. Ensuring that the indicators of other sources will not be modified while doing this operation is guaranteed by the term $\langle s_j \rangle = \langle \text{Sender} \rangle$ in $\langle y \rangle$, thus when this equality test is equal to zero, y will be zero, and thus x_j will not be modified in step 5.VII.C.ii.b, and ensuring that only one indicator for Sender is switched from one to zero is guaranteed using the variable d, that is, once we reach the first indicator x_j for Sender that is equal to one, y will be equal to one in step 5.VII.C.ii.a, and x_j will be equal to zero in 5.VII.C.ii.b, and d will be equal to zero. Once d becomes zero, y will always have the value zero, and thus no other indicator will change in step 5.VII.C.ii.b

In addition, we introduce a counter $\mathsf{CNT}_{\mathsf{DeadLock}}$ into the algorithm in order to identify a deadlock. This counter is augmented by one each time we perform an iteration, and therefore we will have a deadlock when this counter reaches m.

Finally, we need to notify participants of the execution of their transactions. For every transaction executed $t = [\langle s \rangle, \langle a \rangle, \langle r \rangle]$, this algorithm sends to every bank i the opening of the values, $\{(\langle \mathbf{w}_i^t \rangle \cdot \langle s \rangle, \langle \mathbf{w}_i^t \rangle \cdot \langle a \rangle, \langle \mathbf{w}_i^t \rangle \cdot \langle r \rangle), (\langle \mathbf{c}_i^t \rangle \cdot \langle s \rangle, \langle \mathbf{c}_i^t \rangle \cdot \langle a \rangle, \langle \mathbf{c}_i^t \rangle \cdot \langle r \rangle)\}$ which are equal to $\{(0,0,0), (0,0,0)\}$ if t was not sent to i or received by i; equal to $\{(s,a,r), (0,0,0)\}$ if $i = s$; and equal to $\{(0,0,0), (s,a,r)\}$ if $i = r$.

3.1 Experiments

We implemented the above three variants using the SCALE MPC system [ACK$^+$21], which provides a convenient interface to access different secret sharing based access structures, as well as the comparison and Demux operations discussed above. For this purpose we used a machine of 128GB of RAM, with an Intel i-9900 CPU. The experiments were performed using a setup using Shamir Secret Sharing between three parties, over a finite field of size 128 bits. This models the situation where the n banks outsource the computation to $p = 3$ entities, such that each bank trusts that only one of the three entities will act in a malicious manner. The reason for doing this is to avoid having to perform a secure computation with n servers, which will be prohibitively expensive for practical values of n.

We begin by examining the online phase runtimes of single runs of the algorithms of this section (without including the notification part of the participants about the execution of their transactions). In all experiments we varied the number of banks n to have the values $\{8, 64, 128, 256, 512, 1024\}$

<div style="border:1px solid">

<p align="center">MPC variant of the Gridlock Resolution Algorithm</p>

$\mathsf{GR}(\langle B^i \rangle_{i=1}^n, Q, \mathsf{type})$

Given a set of balances B_i and a queue of transactions this determines a subset of the transactions $U \subset Q$ which can be executed; where the set U is given by an indicator set X. If no such U exists it returns DEADLOCK, otherwise it returns the set of remaining transactions which have not been executed.

(1) $X \leftarrow [\langle x_t \rangle]_{t \in Q}$ with $\langle x_t \rangle \leftarrow 1$.

(2) $C, W \leftarrow \emptyset$.

(3) If $\mathsf{type} = \mathsf{sodsGR}$ then $C \leftarrow \mathsf{Demux}_{\mathsf{sodsGR}}(Q)$.

(4) If $\mathsf{type} = \mathsf{ssdsGR}$ then $(C, W) \leftarrow \mathsf{Demux}_{\mathsf{ssdsGR}}(Q)$ and $\mathsf{CNT}_{\mathsf{DeadLock}} \leftarrow 0$.

(5) Repeat

    (I) $[\langle B_U^i \rangle]_{i=1}^n \leftarrow \mathsf{Balance}([\langle B^i \rangle]_{i=1}^n, Q, X, C, W, \mathsf{type})$.

  (II) If $\mathsf{type} \neq \mathsf{ssdsGR}$

      (A) For $i \in \mathsf{Sen}_Q$ do $\langle h^i \rangle \leftarrow (\langle B_U^i \rangle < 0)$.

      (B) $\langle z \rangle \leftarrow \prod_{\{i \in \mathsf{Sen}_Q\}} (1 - \langle h^i \rangle)$.

  (III) else

      (A) $\langle \mathsf{Sender} \rangle \leftarrow 0$.

      (B) For $i \in \{1, \ldots, n\}$:

          (i) $\langle h^i \rangle \leftarrow (\langle B^i \rangle < 0)$

          (ii) $\langle f^i \rangle \leftarrow \langle h^i \rangle - \langle h^i \rangle \sum_{k=1}^{k=i-1} \langle f^k \rangle$

          (iii) $\langle \mathsf{Sender} \rangle \leftarrow \langle \mathsf{Sender} \rangle + i \cdot \langle f^i \rangle$

      (C) $\langle z \rangle \leftarrow \sum_{i=1}^n \langle f^i \rangle$

  (IV) $z \leftarrow \mathsf{Open}(\langle z \rangle)$.

  (V) If $z = 1$ then

      (A) $x_t \leftarrow \mathsf{Open}(\langle x_t \rangle)$ for $t \in Q$, and let $X' = [x_t]_{t \in Q}$.

      (B) Execute $\mathsf{Notify}(X', Q, C, W, \mathsf{type})$.

      (C) For $i \in [1, \ldots, n]$ execute $\langle B^i \rangle \leftarrow \langle B_U^i \rangle$.

      (D) $Q \leftarrow Q \setminus \{t\}_{t \in Q, x_t = 1}$.

      (E) Return Q.

  (VI) If $\mathsf{type} \neq \mathsf{ssdsGR}$

      (A) For $i \in \mathsf{Sen}_Q$

          (i) Let $t_1, \ldots, t_v$ denote the transactions in Q with source i ordered in time of receipt order.

          (ii) For $j = 1, \ldots, v - 1$ do

             (a) $\langle x_{t_j} \rangle \leftarrow (\langle x_{t_j} \rangle \cdot \langle x_{t_{j+1}} \rangle) \cdot \langle h^i \rangle + \langle x_{t_j} \rangle \cdot (1 - \langle h^i \rangle)$.

          (iii) $\langle x_{t_v} \rangle \leftarrow \langle x_{t_v} \rangle \cdot (1 - \langle h^i \rangle)$.

      (B) $\langle \mathsf{DeadLock} \rangle \leftarrow \prod_{i \in \mathsf{Sen}_Q, t \in Q^i} (1 - \langle x_t \rangle)$.

      (C) $\mathsf{DeadLock} \leftarrow \mathsf{Open}(\langle \mathsf{DeadLock} \rangle)$.

  (VII) else

      (A) $\mathsf{CNT}_{\mathsf{DeadLock}} \leftarrow \mathsf{CNT}_{\mathsf{DeadLock}} + 1$

      (B) If $(\mathsf{CNT}_{\mathsf{DeadLock}} = m - 1)$:

          (i) $\mathsf{DeadLock} \leftarrow 1$.

      (C) else

          (i) $\langle d \rangle \leftarrow 1$

          (ii) For j in $\{m, \ldots, 1\}$

             (a) $\langle y \rangle \leftarrow (\langle s_j \rangle = \langle \mathsf{Sender} \rangle) \cdot \langle x_j \rangle \cdot \langle d \rangle$.

             (b) $\langle x_j \rangle \leftarrow \langle x_j \rangle - \langle y \rangle$.

             (c) $\langle d \rangle \leftarrow \langle d \rangle - \langle y \rangle$.

(6) Until $\mathsf{DeadLock} = 1$.

(7) Return DEADLOCK.

</div>

Figure 7. MPC variant of the Gridlock Resolution Algorithm

, and the number of transactions m to have the values $\{10, 50, 100\}$. The m transactions were chosen in such a way that a solution will be found after $m/2$ iterations.
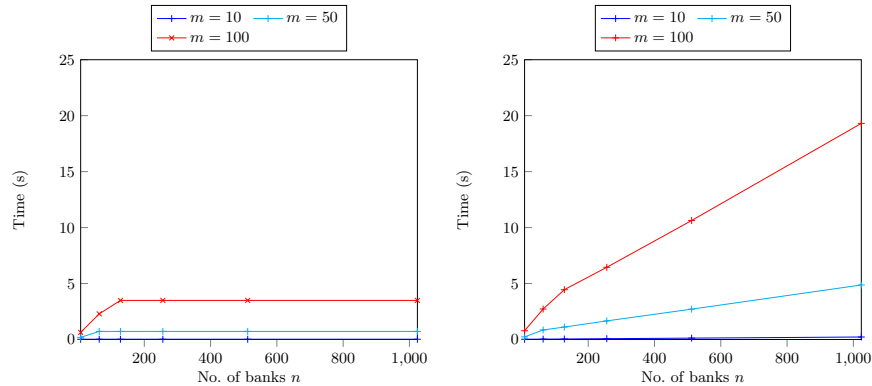

Fig. 8. Run times for sodoGR (left) and sodsGR (right).

sodoGR: The runtimes for this variant are shown in Figure 8. As one can notice, the runtimes keep increasing when n increases, and stabilize when n becomes bigger than m; reaching a maximum of 5 seconds when considering $m = 100$ and $n \geq m$. This is because when $m \leq n$ we set the transactions in such a way that we only process the m banks that will be sources of the transactions. Therefore, when $m \leq n$ the algorithm will perform exactly the same amount of work as when $n = m$.

sodsGR: The runtimes for this experiment are also presented in Figure 8. The runtimes for this algorithm grow as n increases, irrespective of m, when $m = 100$ growing to around 20 seconds for $n = 1000$. When $n \geq m$ the runtime increases, but at a slower pace than for $n < m$. This is due to the fact that once n reaches m, the only extra computation is the calculation of the Demux flags C, as we can set in this case m banks to be the sources of the transactions when $m \leq n$.

Since this variant needs to calculate the Demux flags, as well as performing more multiplications to compute the received values, for all values of n and m it is slower than sodoGR. For example in step 1.II.C of the Balance algorithm in Figure 5 we need to compute a multiplication by the Demux flag $\langle \mathbf{c}_t^i \rangle$, as opposed to performing a conditional multiplication (conditioned on where $r = i$) in step 1.II.B of the same algorithm.

ssdsGR: The runtimes for this variant are shown in Figure 9. The first observation that one would make, is that the runtimes keep increasing with the same rate even when $m \leq n$. This is related to the fact that sources are hidden to the algorithms. For $m = 100$ the algorithm requires 80 seconds to process $n = 1000$ banks data. This is due to the fact that, as explained in Section 3, the algorithm cannot exclude the banks that are not sources of the transactions hanging, therefore whenever a transaction is removed, the balances of all banks are calculated.
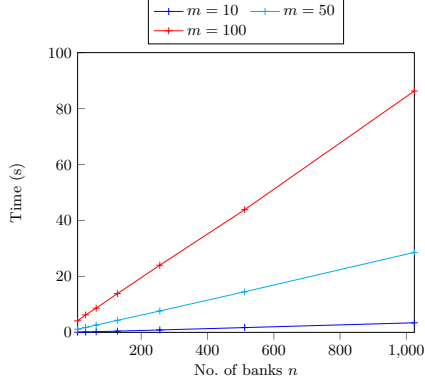
Fig. 9. Run times for ssdsGR.

4 Simulating an RTGS

Having so far provided latency values for single executions of our algorithms, we now turn to discussing are these latencies 'fast enough' in a real application. The latency depends not only on the number of banks n, but also (in the case where we try to keep more than just the transaction amounts private) on the number of unmet transfers in the queue Q, i.e. m. In a real system the value of m can both increase and decrease over time, depending on the frequency of incoming transactions and the amount of liquidity in the system. In this section we aim to simulate a realistic system and determine the cost of performing the liquidity matching in a privacy preserving manner.

Simulating Transactions: Our first task is to simulate the banks and the transactions between them. The transactions exchanged between banks over a period of time L can be modeled as a directed graph, where nodes represent banks and edges among them represent the flow of the transactions, i.e. a link from bank A to bank B is formed if bank A sent a transaction to bank B over L. This graph can be either unweighted, that is, in case bank A sends to bank B multiple transactions, these are counted with only one link; or weighted, that is, the link from A to B has a weight which is determined by either the number of transactions sent from A to B over L, or the volume of these transactions, i.e., the total amount of money sent from A to B.

The structure and properties of such graphs has been extensively studied in the literature, and it has been shown that scale free graphs (graphs for which the degree distribution follows the power law) are the closest ones to model transaction graphs (see [CHBA03] for more details about the structural properties of scale free graphs). In particular [SBA+07] shows that the Fedwire system in the United States has a scale free behavior, and similar observations have been made for the Japanese interbank payment system [INT+04] and the Austrian interbank market [BEST04]. Even the transaction graph formed by Bitcoin has been shown to be scale free [ALVL19].

Our simulation is controlled by a number of parameters: n the number of banks in the network, n_0 the number of 'central nodes' (see below), the total number of transactions M to be simulated, the time interval L over which we simulate the transactions, a parameter o giving the average number of transactions for each of the $n - n_0$ non-central nodes, a value for modifying the preferential attachment α, and a value β controlling the amount of liquidity in the system.

For our work, we utilized the simulator of [SC13] to generate the transaction graph. This simulator uses a tweaked version of the Barabasi-Albert model to generate scale free graphs. In this

simulation the graph is built by setting first n_0 central nodes among the n nodes, these central nodes are intended to send and receive transactions more than the remaining nodes; they correspond in the real world to the important banks in a network. This preference is guaranteed by at the beginning setting the preferential attachment v_i for these banks to be one, and for the remaining banks to be zero. The algorithm then proceeds to find a total number of $M = o \cdot (n - n_0)$ transactions $t_1, \ldots, t_M$.

These transactions $t_i = (s, a, r)$ are generated as follows: The source bank $s \in \{1, \ldots, n\}$ is selected with probability $v_s / \sum_{j=1}^{j=n} v_j$, the destination bank $r \in \{1, \ldots, n\}$ is selected with probability $v_r / \sum_{j=1}^{j=n} v_j$. If we obtain $s = r$ then a new value of r is sampled in the same way, until $s \neq r$. Whenever a transaction is generated, we update the preferential attachment for both the source and destination, by adding $\alpha = 0.1$ to v_s and v_r, and whenever o transactions are generated, we set the preferential attachment to be 1 for one of the banks that were not yet considered for sending or receiving transactions, i.e., a bank i that still has $v_i = 0$. The amount a is sampled by taking a value v from the normal distribution with mean 1 and standard deviation 0.2, and then setting $a = d \cdot \exp(v)$, where d is the minimum of the in-out degrees of the source and destination nodes s and r; this follows the methodology in [SC13]. It also means that transactions are likely to be larger from banks which are more central to the graph, a fact which is born out in practice.

To each transaction we assign a time of occurrence; for this we assume that the M transactions are uniformly distributed over the time interval L. The order of the transactions are as in the simulation above, with transaction t_i entering the system at time τ_i, where we have $\tau_{i+1} - \tau_i \approx L/M$ for all i.

For the initial balances of the banks, these are set according to a parameter $\beta \in [0, 1]$ that determines the amount of liquidity available in the system. That is, we calculate the lower and upper bounds of liquidity for each bank, where the lower bound L_i for bank i refers to the minimal initial balance that will allow the bank to settle all its transactions at the end of the time window, and the upper bound U_i refers to an initial balance that will allow the bank to settle immediately all its transactions without having to be placed in the queue U for the gridlock execution. Then the initial balance of bank i we set equal to $B^i = \beta \cdot (U_i - L_i)$.

In [SC13] it is claimed that the above algorithm is a good model for transaction graphs between banks. In particular they discuss the Fedwire transaction graph. To get some idea of the sizes of the graph we note that Fedwire in 2008 had approximately $n = 7300$ participants[4] and the average daily volume of transfers in January 2021[5] was $803,413.89$ transactions. A smaller system is the Target2 system run by the European Central Bank, according to the annual report of 2019[6], the number of direct participant is $n = 1050$, with each participant originating on average $344,120$ transactions per day.

The Simulation: In our simulation we assume that for sodoGR and sodsGR, two banks will settle an incoming transaction if it can be settled immediately, with unsettled transactions being passed to the queue Q for application of the Gridlock Resolution algorithm. For the case of ssdsGR, the only way a transaction can be settled is through the Gridlock Resolution algorithm. For the remaining transactions, we process them in one of two manners: In Version 1 each transaction is processed one at a time, it is cleared immediately if it can be, otherwise it is added to the current queue Q.

For the cases of sodsGR and ssdsGR, we run the GR algorithm if Q has more than one entry. For the case of sodoGR, we run the GR algorithm if the last incoming transaction (either settled or not) has as the destination one of the sources of the transactions in Q. In Version 2 we simulate the current time (according to how long previous executions of the GR algorithm take), we thus add all transactions, which have arrived between the start of the previous execution of the GR algorithm and the end of the algorithm, to the queue Q, after clearing all which can be executed immediately. For the cases of sodsGR and ssdsGR, we then execute the GR algorithm if Q has more than one entry. For the case of sodoGR, we run the GR algorithm if at least one of the last incoming transactions (either settled or not) has as the destination one of the sources of the transactions in Q. These two variants are described in Figure 10.

Experimental Results We executed the above simulation, again with an MPC system of three players, over a simulation time window L of one hour, considering only the online phase, with three different levels of liquidity $\beta \in \{0.1, 0.5, 0.9\}$, and with $n_0 = 10$. We tested on different values of n and o, with n either 100 or 1000; a value of 1000 is approximately the number of direct participants in Target 2. Our values of o (which controls the number of transactions $M = o \cdot (n - n_0)$) where also chosen to approximate the number of transactions Target 2 deals with in an hour assuming transactions are uniformly distributed through the day (in this case only with $n = 100$).

We considered two main metrics to evaluate the performance: (i) How much time we exceed the time window L, i.e. was the value τ output by the simulation larger than the time window L, i.e. $E = \tau - L$. (ii) The average delay time for transactions. Each transaction (t, τ_i) is supposed to enter to the system in time τ_i, however, the system may not be able to address this transaction at time τ_i, but only at a later time τ_i', after finishing an ongoing gridlock computation. Thus the average delay time, in seconds, will be $D = \sum_{i=1}^{i=M}(\tau_i' - \tau_i)/M$. In a real system one would desire that E is zero, or as close as possible (to ensure real time settlement of all transactions in a day) and that D is also as small as possible (to ensure individual transactions are not delayed too much). The results of our simulation are given in Table 1.

We can notice from the runtimes that the more liquidity we have, the faster the runtimes are. This is because liquidity affects how many times the gridlock computation takes place, as well as the size of the queues on which this computation happens. We can also notice that the slowest runtimes correspond (unsurprisingly) to the case where sources and destinations are hidden. The Version 2 of the algorithm for clearing is much better, and indeed seems to meet the operational requirements of E and D being close to zero, for all cases bar that of both sources and destinations being secret.

One thing to mention is that we obtained with the ssdsGR variant for the fourth setting ($n = 1000, M = 9900$) slower runtimes for $\beta = 0.5$ than for $\beta = 0.1$, which is somewhat counter intuitive. This happened because the longest gridlock executions for $\beta = 0.5$ and $\beta = 0.1$ were the two last ones, which were on queues of sizes 4877 and 4070 for $\beta = 0.5$ and queues of sizes 1501 and 8255 for $\beta = 0.1$. However, for both cases the second to last gridlock execution happened before exceeding L, while the last ones after exceeding L. Such a gridlock execution should be fast, in our simulation, as all the transactions must be settled at the end, and thus no transaction will be excluded.

<div align="center">Simulation of RTGS</div>

RTGSSimulation: Given a set of transaction $\mathsf{T} = \{(t_i, \tau_i)\}$ where t_i is the transaction and τ_i is the time stamp of this transaction execute:

(1) $Q \leftarrow \emptyset$, $T \leftarrow \emptyset$, $\tau = 0$. The variable τ will define the current time in the simulation.

(2) While $\mathsf{T} \neq \emptyset$ do

   (I) If Version 1

      (A) Let (t_i, τ_i) denote element in T with the minimal value of τ_i.

      (B) $\tau \leftarrow \tau_i$ if $\tau < \tau_i$.

      (C) $\mathsf{T} \leftarrow \mathsf{T} \setminus \{(t_i, \tau_i)\}$.

      (D) $\mathcal{U} \leftarrow \{(t_i, \tau_i)\}$.

   (II) Else if Version 2

      (A) Let $\mathcal{U}$ denote all transactions in T with time $\tau_i < \tau$.

      (B) If $\mathcal{U} = \emptyset$ execute the above four steps for Version 1.

      (C) Else $\mathsf{T} \leftarrow \mathsf{T} \setminus \mathcal{U}$.

   (III) Start-timer; this is used to ensure we simulate time correctly.

   (IV) For all $(t, \tau_i) \in \mathcal{U}$

      (A) If $\mathsf{type} = \mathsf{sodoGR}$:

         (i) Write $t = [s, \langle a \rangle, r]$.

         (ii) $\langle z \rangle \leftarrow \langle B^s \rangle \geq \langle a \rangle$

         (iii) $z \leftarrow \mathsf{Open}(\langle z \rangle)$

         (iv) If $z = 1$ then $\langle B^s \rangle \leftarrow \langle B^s \rangle - \langle a \rangle$ and $\langle B^r \rangle \leftarrow \langle B^r \rangle + \langle a \rangle$ and execute $\mathsf{Notify}(\{1\}, \{t\}, \emptyset, \emptyset, \mathsf{sodoGR})$.

         (v) Else $Q \leftarrow Q \cup \{t\}$.

      (B) Else if $\mathsf{type} = \mathsf{sodsGR}$:

         (i) Write $t = [s, \langle a \rangle, \langle r \rangle]$.

         (ii) $\langle z \rangle \leftarrow \langle B^s \rangle \geq \langle a \rangle$

         (iii) $z \leftarrow \mathsf{Open}(\langle z \rangle)$

         (iv) If $z = 1$ then

            (a) $\mathbf{v} \leftarrow \mathsf{Demux}(\langle r \rangle, n)$.

            (b) $\langle B^s \rangle \leftarrow \langle B^s \rangle - \langle a \rangle$

            (c) $\langle B^i \rangle \leftarrow \langle B^i \rangle + \langle a \rangle \cdot \mathbf{v}_i$ for $i \in [1, \ldots, n]$

            (d) Execute $\mathsf{Notify}(\{1\}, \{t\}, \mathbf{v}, \emptyset, \mathsf{sodsGR})$.

         (v) $Q \leftarrow Q \cup \{t\}$.

      (C) Else if $\mathsf{type} = \mathsf{ssdsGR}$:

         (i) $Q \leftarrow Q \cup \{t\}$.

   (V) If $\mathsf{type} = \mathsf{sodoGR}$ and $\exists t_j = [s_j, \langle a_j \rangle, r_j] \in \mathcal{U}$ and $\exists t_k = [s_k, \langle a_k \rangle, r_k] \in Q \setminus \mathcal{U}$ such that $r_j = s_k$ then execute $\mathsf{sodoGR}(Q)$.

   (VI) Else if $\mathsf{type} = \mathsf{sodsGR}$ then execute $\mathsf{sodsGR}(Q)$, running $\mathsf{Demux}_{\mathsf{sodsGR}}$ only on the new transactions in $\mathcal{U} \subset Q$, by keeping track of the prior executions.

   (VII) Else if $\mathsf{type} = \mathsf{ssdsGR}$ then execute $\mathsf{ssdsGR}(Q)$, running $\mathsf{Demux}_{\mathsf{sodsGR}}$ and $\mathsf{Demux}_{\mathsf{ssdsGR}}$ only on the new transactions in $\mathcal{U} \subset Q$, by keeping track of the prior executions.

   (VIII) Stop-timer; let the elapsed time be denoted by τ_0.

   (IX) $\tau \leftarrow \tau + \tau_0$.

(3) Output the elapsed time τ needed to execute all the transactions.

Figure 10. Simulation of RTGS

			Version 1		Version 2	
			E	D	E	D
sodoGR	$\beta = 0.1$	$n = 100, M = 900$	0	0	0	0
		$n = 100, M = 9000$	0	106	0	0
		$n = 100, M = 45000$	-	-	0	0
		$n = 1000, M = 9900$	-	-	0	1
	$\beta = 0.5$	$n = 100, M = 900$	0	0	0	0
		$n = 100, M = 900$	0	0	0	0
		$n = 100, M = 45000$	-	-	0	0
		$n = 1000, M = 9900$	-	-	0	0
	$\beta = 0.9$	$n = 100, M = 900$	0	0	0	0
		$n = 100, M = 900$	0	0	0	0
		$n = 100, M = 45000$	-	-	0	0
		$n = 1000, M = 9900$	-	-	0	0
sodsGR	$\beta = 0.1$	$n = 100, M = 900$	0	1	0	0
		$n = 100, M = 9000$	17382	11357	0	0
		$n = 100, M = 45000$	-	-	0	1
		$n = 1000, M = 9900$	-	-	85	47
	$\beta = 0.5$	$n = 100, M = 900$	0	0	0	0
		$n = 100, M = 900$	302	346	0	0
		$n = 100, M = 45000$	-	-	0	1
		$n = 1000, M = 9900$	-	-	47	41
	$\beta = 0.9$	$n = 100, M = 900$	0	0	0	0
		$n = 100, M = 900$	0	0	0	0
		$n = 100, M = 45000$	-	-	0	0
		$n = 1000, M = 9900$	-	-	0	0
ssdsGR	$\beta = 0.1$	$n = 100, M = 900$	49	102	0	0
		$n = 100, M = 9000$	149559	90404	0	17
		$n = 100, M = 45000$	-	-	707	490
		$n = 1000, M = 9900$	-	-	5507	3874
	$\beta = 0.5$	$n = 100, M = 900$	0	0	0	0
		$n = 100, M = 900$	26147	13708	0	2
		$n = 100, M = 45000$	-	-	319	212
		$n = 1000, M = 9900$	-	-	10500	4983
	$\beta = 0.9$	$n = 100, M = 900$	0	0	0	0
		$n = 100, M = 900$	0	0	0	0
		$n = 100, M = 45000$	-	-	0	0
		$n = 1000, M = 9900$	-	-	3965	1877

Table 1. Experimental Results

Acknowledgments

References

ACK+21.   Abdelrahaman Aly, Kelong Cong, Marcel Keller, Emmanuela Orsini, Dragos Rotaru, Oliver Scherer, Peter Scholl, Nigel P. Smart, Titouan Tanguy, and Tim Wood. SCALE and MAMBA v1.12: Documentation, 2021.

AHBPV20.   Gilad Asharov, Tucker Hybinette Balch, Antigoni Polychroniadou, and Manuela Veloso. Privacy-preserving dark pools. In *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS 20, pages 1747–1749, Richland, SC, 2020. International Foundation for Autonomous Agents and Multiagent Systems.

AKL11.   Emmanuel A. Abbe, Amir E. Khandani, and Andrew W. Lo. Privacy-preserving methods for sharing financial risk exposures, 2011.

ALVL19.   Aspembitova A, Feng L, Melnikov V, and Chew LY. Fitness preferential attachment as a driving mechanism in bitcoin transaction network. *PLoS One*, 14, 2019.

BCD+08.   Peter Bogetoft, Dan Lund Christensen, Ivan Damgard, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael Schwartzbach, and Tomas Toft. Multiparty computation goes live. Cryptology ePrint Archive, Report 2008/068, 2008. http://eprint.iacr.org/2008/068.

BCD+09.   Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In Roger Dingledine and Philippe Golle, editors, *FC 2009*, volume 5628 of *LNCS*, pages 325–343. Springer, Heidelberg, February 2009.

BDJ+06.   Peter Bogetoft, Ivan Damgård, Thomas Jakobsen, Kurt Nielsen, Jakob Pagter, and Tomas Toft. A practical implementation of secure auctions based on multiparty integer computation. In Giovanni Di Crescenzo and Avi Rubin, editors, *FC 2006*, volume 4107 of *LNCS*, pages 142–147. Springer, Heidelberg, February / March 2006.

BEST04.   Michael Boss, Helmut Elsinger, Martin Summer, and Stefan Thurner. Network topology of the interbank market. *Quantitative Finance*, 4(6):677–684, 2004.

BHSR19.   Samiran Bag, Feng Hao, Siamak F. Shahandashti, and Indranil G. Ray. SEAL: Sealed-bid auction without auctioneers. Cryptology ePrint Archive, Report 2019/1332, 2019. https://eprint.iacr.org/2019/1332.

BP20.   David Byrd and Antigoni Polychroniadou. Differentially private secure multi-party computation for federated learning in financial applications. Papers, arXiv.org, 2020.

BS01.   Morten Bech and Kimmo Soramaki. Gridlock resolution in payment systems. *Danmarks Nationalbank Monetary Review*, 07 2001.

BTW11.   Dan Bogdanov, Riivo Talviste, and Jan Willemson. Deploying secure multi-party computation for financial data analysis. Cryptology ePrint Archive, Report 2011/662, 2011. http://eprint.iacr.org/2011/662.

CaSHMM.   James Chapman, Rodney Garratt ans Scott Hendry, Andrew McCormack, and Wade McMahon. Project jasper: Are distributed wholesale payment systems feasible yet? https://www.finextra.com/finextra-downloads/newsdocs/fsr-june-2017-chapman.pdf.

Cd10.   Octavian Catrina and Sebastiaan de Hoogh. Improved primitives for secure multiparty integer computation. In Juan A. Garay and Roberto De Prisco, editors, *SCN 10*, volume 6280 of *LNCS*, pages 182–199. Springer, Heidelberg, September 2010.

CHBA03.   Reuven Cohen, Shlomo Havlin, and Daniel Ben-Avraham. *Structural properties of scale-free networks*, pages 85–110. Wiley, 2003.

CS10.   Octavian Catrina and Amitabh Saxena. Secure computation with fixed-point numbers. In Radu Sion, editor, *FC 2010*, volume 6052 of *LNCS*, pages 35–50. Springer, Heidelberg, January 2010.

CSA20.   John Cartlidge, N. Smart, and Y. Alaoui. Multi-party computation mechanism for anonymous equity block trading: A secure implementation of turquoise plato uncross. *IACR Cryptol. ePrint Arch.*, 2020:662, 2020.

CSA21.   Daniele Cozzo, N. Smart, and Y. Alaoui. Secure fast evaluation of iterative methods: With an application to secure pagerank. *IACR Cryptol. ePrint Arch.*, 2021, 2021.

CST18.   John Cartlidge, Nigel P. Smart, and Younes Talibi Alaoui. MPC joins the dark side. Cryptology ePrint Archive, Report 2018/1045, 2018. https://eprint.iacr.org/2018/1045.

CYC⁺20.   Shengjiao Cao, Yuan Yuan, Angelo De Caro, Karthik Nandakumar, Kaoutar Elkhiyaoui, and Yanyan Hu. Decentralized privacy-preserving netting protocol on blockchain for payment systems. In Joseph Bonneau and Nadia Heninger, editors, *Financial Cryptography and Data Security - 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10-14, 2020 Revised Selected Papers*, volume 12059 of *Lecture Notes in Computer Science*, pages 137–155. Springer, 2020.

DFK⁺06.   Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In Shai Halevi and Tal Rabin, editors, *TCC 2006*, volume 3876 of *LNCS*, pages 285–304. Springer, Heidelberg, March 2006.

ECB.   ECB. Target 2. https://www.ecb.europa.eu/paym/target/target2/html/index.en.html.

Eur.   European Central Bank and Bank of Japan. Payment systems: liquidity saving mechanisms in a distributed ledger environment. https://www.ecb.europa.eu/pub/pdf/other/ecb.stella_project_report_september_2017.pdf.

GJL98.   Michael M. Guntzer, Dieter Jungnickel, and Matthias Leclerc. Efficient algorithms for the clearing of interbank payments. *European Journal of Operational Research*, 106(1):212–219, 1998.

GS10.   Marco Galbiati and Kimmo Soramaki. Liquidity-saving mechanisms and bank behaviour. *Bank of England, Bank of England working papers*, 07 2010.

HFT20.   Marcella Hastings, Brett Falk, and Gerry Tsoukalas. Privacy-preserving network analytics, 08 2020.

INT⁺04.   Hajime Inaoka, Takuto Ninomiya, Ken Taniguchi, Tokiko Shimizu, and Hideki Takayasu. Fractal Network derived from banking transaction – An analysis of network structures formed by financial institutions –. Bank of Japan Working Paper Series 04-E-4, Bank of Japan, April 2004.

KRSW18.   Marcel Keller, Dragos Rotaru, Nigel P. Smart, and Tim Wood. Reducing communication channels in MPC. In Dario Catalano and Roberto De Prisco, editors, *SCN 18*, volume 11035 of *LNCS*, pages 181–199. Springer, Heidelberg, September 2018.

KS14.   Marcel Keller and Peter Scholl. Efficient, oblivious data structures for MPC. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 506–525. Springer, Heidelberg, December 2014.

Mau06.   Ueli M. Maurer. Secure multi-party computation made simple. *Discrete Applied Mathematics*, 154(2):370–381, 2006.

oS.   Monetary Authority of Singapore. Project ubin: Central bank digital money using distributed ledger technology. https://www.mas.gov.sg/schemes-and-initiatives/Project-Ubin.

PRST06.   D. C. Parkes, M. O. Rabin, S. M. Shieber, and C. A. Thorpe. *Practical Secrecy-Preserving, Verifiably Correct and Trustworthy Auctions*, pages 70–81. Association for Computing Machinery, New York, NY, USA, 2006.

SBA⁺07.   Kimmo Soramaki, Morten L. Bech, Jeffrey Arnold, Robert J. Glass, and Walter E. Beyeler. The topology of interbank payment flows. *Physica A: Statistical Mechanics and its Applications*, 379(1):317 – 333, 2007.

SC13.   Kimmo Soramaki and Samantha Cook. Sinkrank: An algorithm for identifying systemically important banks in payment systems. *Economics: The Open-Access, Open-Assessment E-Journal*, 7, 06 2013.

SD06.   Yakov Shafransky and Alexandr Doudkin. An optimization algorithm for the clearing of interbank payments. *European Journal of Operational Research*, 171:743–749, 02 2006.

SvA⁺18.   Alex Sangers, Maran van Heesch, Thomas Attema, Thijs Veugen, Mark Wiggerman, Jan Veldsink, Oscar Bloemen, and Daniël Worm. Secure multiparty PageRank algorithm for collaborative fraud detection. Cryptology ePrint Archive, Report 2018/917, 2018. https://eprint.iacr.org/2018/917.

WXF⁺18.   Xin Wang, Xiaomin Xu, Lance Feagan, Sheng Huang, Limei Jiao, and Wei Zhao. Inter-bank payment system on enterprise blockchain platform. In *11th IEEE International Conference on Cloud Computing, CLOUD 2018, San Francisco, CA, USA, July 2-7, 2018*, pages 614–621. IEEE Computer Society, 2018.

Yao86.   Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167. IEEE Computer Society Press, October 1986.