Private Liquidity Matching using MPC

Shahla Atapoor, Nigel P. Smart, and Younes Talibi Alaoui

imec-COSIC, KU Leuven, Leuven, Belgium.
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 with high-value payment settlements between participants. 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 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 could 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) [23] 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. This will eventually permit the unblocking some of the pending transactions if the 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 [21] 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 all payment instructions, which includes the source, destination, and the amount being transferred. Thus, this entity is trusted to preserve the confidentiality of this information. As a result, participants might be skeptical about submitting all their transactions to the RTGS. In order to eliminate this need for 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 gives rise to three 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. (iii) Privacy: Account balances and payment amounts, and possible source and destination identities should be kept private to the respective banks. These 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 fact, one could imagine a fully decentralized solution where in the case of a gridlock, each participant notifies the banks it is planning to send transactions to (with the respective amounts), and based on this, each participant would determine the expected balance it would hold. Then each participant removes transactions from its queue till the expected balance becomes positive, and notifies the corresponding receivers about the deleted transactions. This will allow the participants to resolve the gridlock situation without the interference of a third party, but does not necessarily address the aforementioned challenges, particularly the privacy guarantee, as the participants cannot move forward from the gridlock situation, until all participants confirm that they have a positive balance, and this obviously depends on the transactions they are exchanging. Thus, it may occur that correlations can be drawn over the sender and receiver of a transaction, from the sequence of the parties that signal having a negative balance. It is thus not trivial how to construct a protocol that avoids such leakages, without introducing heavy computation that would make the RTGS impractical.

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 instruc-

tions and balances will remain hidden as long as less than $t+1$ of these entities collude.

We show that these p entities will be capable of obliviously running the multilateral netting process. For this purpose, we distribute the gridlock resolution algorithm of [6] (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 [3], with $p = 3$ and $t = 1$, and we simulated an RTGS system running over time using the three aforementioned versions of gridlock resolution. For our simulation, we generated 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. In this case, we also allow a small amount of information leakage, since removing this leakage would cause an even greater delay being introduced.

Prior usage of MPC in financial applications has mainly focused on auctions, such as [5, 8, 9, 29], for one shot auctions, and [4, 13, 14] for auctions running in Dark Markets. MPC was also used for privacy-preserving financial data analysis, such as [7] to conduct statistics over the performance of companies throughout the year or to compute the systemic risk between financial institutions such as in [2] and [25]. Also, [30] and [19] used MPC for detecting fraud between financial institutions, and [11] 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 [17], Ubin [28] and Stella [22] projects, In [35] the authors provide a blockchain based solution, however it still relied on a central entity that checks the correctness of the multilateral netting process. In addition, 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).

More recently, another blockchain-based work was proposed in [12], which is the work most closely related to ours. In [12] 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. In addition,

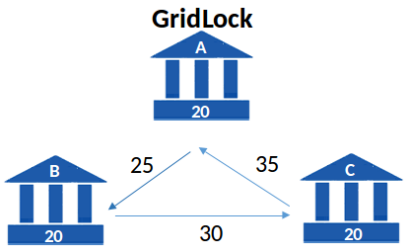

Fig. 1. The banks do not have enough liquidity to settle their transactions, as each bank is planning to send more than what it currently holds. However, a multilateral netting will result in positive net positions for all banks.

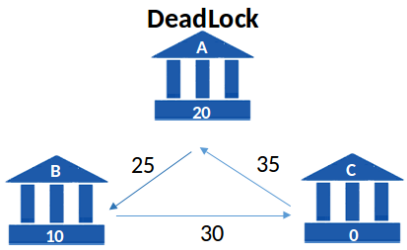

Fig. 2. The banks do not have enough liquidity to settle their transactions, as each bank is planning to send more than what it currently holds, and even multilateral netting will not result in positive net positions for all banks

the protocol would still suffer from the issue described earlier about participants inferring correlations between senders and receivers, from the expected balances of the participants during the multilateral netting process.

The paper [12] 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 1697 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 [12] would require around 63 minutes to process the transactions, and thus is not real time even with these relatively modest transaction levels.

2 Preliminaries

2.1 The Gridlock Resolution Problem

For a pictorial description of the problem see Fig. 1 and Fig. 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 [6] 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.

The transactions are assumed to come with an ordering $t < t'$ which implies, for a given source bank, that the bank prefers t to be executed before t'. 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.

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^i_U and R^i_U denote respectively the amounts sent and received by bank i after running simultaneously a set of U transactions. Let B^i_U denote the balance of this bank after running these transactions. i.e., $B^i_U = B^i - S^i_U + R^i_U$.

The key two states 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 $U \subseteq Q$ and $U \neq \emptyset$, such that:*

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

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

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

Definition 2.2 (Deadlock). *A deadlock is a situation where $\nexists U \subseteq Q$ 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 if no strict global ordering is given, and thus one needs to use approximate algorithms in order to solve it, such as the ones introduced in [31] and [24]. However, the problem is not NP-complete if the transactions are augmented with a strict ordering $<$ of execution, and in such a situation one can find an optimal solution in polynomial time.

The Gridlock Resolution Algorithm: The algorithm for gridlock resolution GR [6] proceeds in the following steps, with the formal definition of the algorithm being given in Fig. 3. We first empty the queue Q by moving all the transactions from Q to U. We then call a subroutine 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[1]. 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 a deadlock and we need to wait for more transactions to come in before running the GR algorithm again.

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 returns 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

Fig. 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 introduced by Yao in [36], 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

[1] The algorithm from [6] removes just one transaction of a source with a negative balance at this point, but it is equivalent to removing one transaction from each source which has a negative balance.

participating in the protocol using a secret sharing scheme. The computation on the shared secret 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 $\mathbb{F}_q$, for a large prime q.

The specific underlying MPC protocol we will use is that of [32], this is a protocol which provides active security with abort. This means that an adversary may arbitrarily deviate from the protocol, but if he does so then with overwhelming probability the honest parties should abort the protocol. Whilst based on information-theoretic primitives, the protocol of [32] is computationally secure since it relies on hash functions and PRFs, etc for efficiency (as well as TLS in the implementation we use in order to obtain secure channels).

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 input from the environment. If this input is $\mathsf{Deliver}$ 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 $\mathsf{Deliver}$ then $\varnothing$ is output to all parties.

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

The protocol of [32] essentially allows the implementation of the arithmetic black box defined in Fig. 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 the players' shares of $\langle z \rangle$ to the external bank i.) This last operation is used to send data to external parties which is the output of the computation; this is identical as an opening to an internal player in terms of the ability for the environment to decide if the correct value is actually delivered.

The protocol of [32] is defined for arbitrary $\mathcal{Q}_2$ access structures, but in this paper, we will concentrate on the simpler subset of threshold access structures. In a threshold access structure on p players, with threshold t, up to t players can collude. A $\mathcal{Q}_2$ access structure in this threshold context is one for which

$t < p/2$. Such honest majority protocols allow efficient multiplication of secret shared data.

In such a situation (threshold access structures with $t < p/2$) we can (and do in this paper) define the underlying secret sharing scheme via Shamir Secret Sharing. 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. We will write $\langle x \rangle$ for the sharing of x with this sharing.

Linear operations on the secret shared values can be performed via local operations, whilst the multiplication operations in our arithmetic black box are performed in an offline-online manner (since our protocol is based upon [32]). In an "offline" phase is where data that does not depend on player inputs or the function is pre-computed, and then in the "online" phase, the actual computation takes place. The protocol of [32] provides an efficient online phase, although the combined cost of the offline plus online phases can be less efficient than other protocol choices.

Up until now, we have considered the arithmetic black box of Fig. 4. This can be extended to cope with non-arithmetic operations in a standard manner. For example, in this work, we will represent integer values $x \in [-2^{K-1}-1, \ldots, 2^K-1]$ as elements of F_q, where $k \ll \log_2 q$. Comparison, and non-arithmetic operations, on such encoded integers, can then be performed using special purpose statistically secure sub-protocols such as those in [15, 16, 20]. The reader should note that comparison is more expensive to execute than multiplication; which is the opposite of what happens when computing in the clear.

At many points, we will need to read and write to an array of secret values, with an index which is also secret. To do this we make use of the naive Oblivious RAM (ORAM) implementation from [27]. We only require the naive methodology as our arrays will be relatively small, meaning the naive methodology will be faster for our example than the more elaborate variants given in [27]. The naive ORAM has at its heart 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.

Given this arithmetic black-box extended with the protocols for comparison and ORAM-style array lookup, we can implement algorithms securely. If the algorithm is correct and utilizes only the operations described above, then the secure implementation will inherit the security properties of the underlying MPC protocol. Particularly, if a server running the computation does not perform the required computation, this will be detected by the honest servers. The only issue which needs to be verified would be if the secure implementation opens any intermediate result, at this point one simply needs to verify that the opened value is something which is inherent in the application; i.e. it is a public value

which we expect the algorithm to output to the parties. This is a topic which we will return to in the next section.

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 transaction, 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). We denote the fully oblivious version ssdsGR. In all situations, we ensure the privacy of the amount in the transactions and privacy of the banks' balances. The basic ideal functionality we aim to emulate is given in Fig. 5.

Gridlock Resolution Algorithm Functionality $\mathcal{F}^{\mathsf{type}}[\mathsf{GRP}]$

The functionality operates between n banks, p servers and an ideal adversary $\mathcal{A}$ that statically corrupts a set A of parties among the servers. We assume that the banks are acting honestly throughout the protocol.
The functionality is reactive in that it maintains a list of the unexecuted queue of transactions Q, and then updates this with the new incoming transaction I. The internal queue is updated with I and then the GRP algorithm described in figure Fig. 3 is executed. The set of executed transactions U is returned (or Deadlock), and these executed transactions are deleted from the internal queue Q for usage in the next call to $\mathcal{F}^{\mathsf{type}}[\mathsf{GRP}]$. We distinguish three versions of the functionality: where type is in $\{\mathsf{sodoGR}, \mathsf{sodsGR}, \mathsf{ssdsGR}\}$

Initialize: The system is initialized with
- An empty queue of transactions Q.
- The current balances b_i for the banks.

InputTrans: On input $t = (s, a, r)$ from bank s, the functionality takes a fresh identifier id_t and associates it to the transaction t. If type $\neq$ ssdsGR and this transaction can be instantly executed, i.e. the balance of bank s is larger than a, then this transaction is executed (i.e. the functionality informs banks s and r of the amount a), otherwise the functionality also adds t to Q.
In addition to this, the functionality may reveal to the servers some of the components of the transaction t depending on the type.
- If type = sodoGR, s and r are revealed to the servers.
- If type = sodsGR, s is revealed to the servers.
- Otherwise, i.e. type = ssdsGR, nothing is revealed to the servers.

Run: On input Run from the servers, the functionality processes Q, by executing the algorithm of Fig. 3. If the algorithm returns Deadlock then this is returned to the servers and the banks. If the algorithm returns a set of transactions U then the functionality informs the servers as to which transactions these values in U correspond to; and they are deleted from the current queue Q. For every transaction $t = (s, a, r)$ in U the functionality informs s and r about this transaction and the amount.

Fig. 5. Gridlock Resolution Algorithm Functionality $\mathcal{F}^{\mathsf{type}}[\mathsf{GRP}]$

In Fig. 6 we present the algorithm to compute the balances, upon processing a list of payments Q, in our three different scenarios. The algorithms, make use

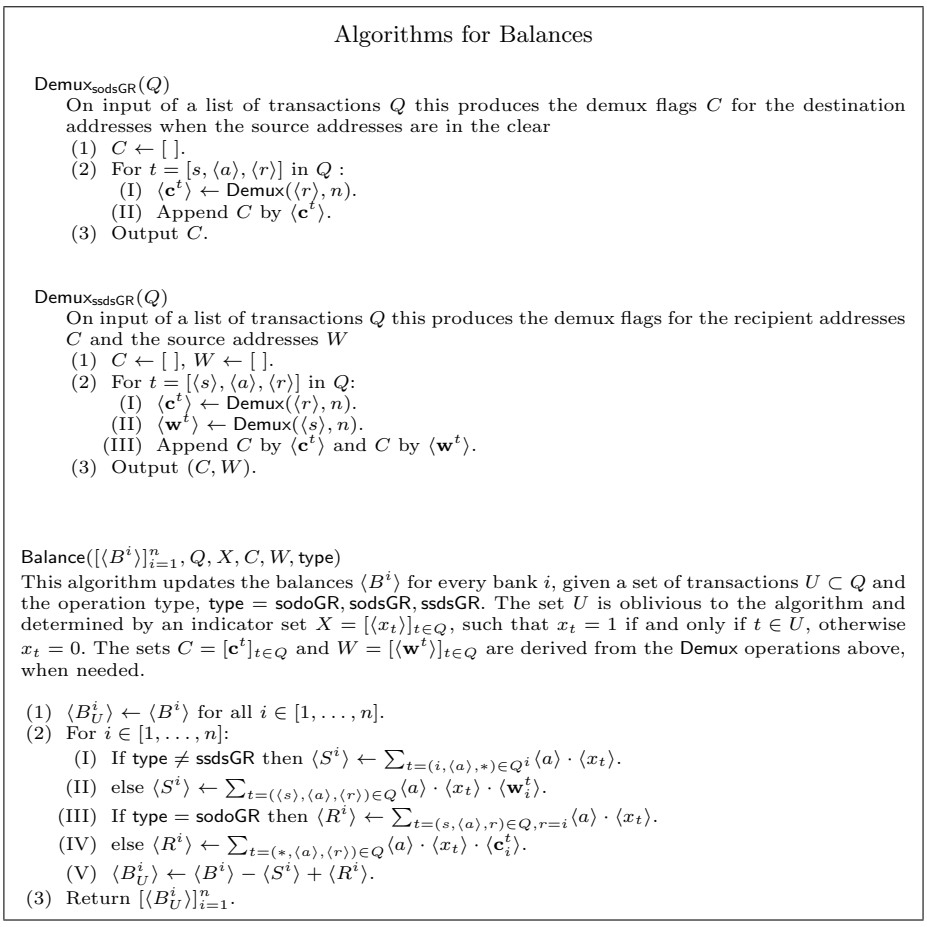

Fig. 6. Algorithms for Balances

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 Fig. 8 we present the algorithm for executing the Gridlock Resolution algorithm in our three cases. This utilizes the algorithm for computing balances from Fig. 6, as well as the algorithm, in Fig. 7, which notifies the banks of their completed transactions. In the following paragraphs, we highlight aspects of these algorithms in the three different situations.

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 Fig. 6.

(1) If type = 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 type = 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 type = 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 $\mathsf{Open}_E(\langle \mathbf{c}_i^t \rangle \cdot \langle r \rangle, i)$.

Fig. 7. Method to Notify Participants of Executed Transactions

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 (Fig. 6) is 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 Fig. 8 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 $\langle h^i \rangle$. Each of these values is 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.

MPC variant of the Gridlock Resolution Algorithm

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

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

Fig. 8. MPC variant of the Gridlock Resolution Algorithm

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,

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 Fig. 6. 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 with 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 from the system, the Gridlock Resolution algorithm needs 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 $\langle \mathsf{Sender} \rangle$ for which we remove the last transaction. Removing this last transaction is done in step 5.VII 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 $s_j = \mathsf{Sender}$. Ensuring that the indicators of other sources will not be modified while doing this operation is guaranteed by the term $\langle s_j \rangle = \langle \mathsf{Sender} \rangle$ 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 Leakage

We now discuss the leakage our algorithms have in comparison to the ideal functionality given in Fig. 5. Given the underlying MPC system implements securely any algorithm, the only leakage which can arise is when our algorithms reveal information via an Open command. We assume that the ordering between transactions is public knowledge, which will be the case if the ordering is done purely in a first-in/first-out manner.

We first discuss each such operation in Fig. 8. The openings in the notify algorithm in Fig. 7 are all identical to values revealed by the ideal functionality.

In step 5.V.A the identifiers of which transactions, which are executed, are opened but this is also something which is leaked in our ideal functionality; thus this line provides no additional leakage over what the ideal functionality will leak.

In step 5.VI.C the algorithm reveals to all servers whether Deadlock has been reached. Again this is something which happens in our ideal functionality.

This leaves us with the opening in step 5.IV, which reveals the value of $\langle z \rangle$. This reveals whether on this iteration we should terminate with a solution or

not, thus revealing this value reveals *the number of iterations* needed to solve the GRP problem or the number of iterations needed to reach Deadlock. In the first situation, this value is equal to the maximum number of transactions not executed by one of the banks when it is the source. Thus this information is always revealed by the functionality in the case where we have type = sodoGR or type = sodsGR. In the case of type = ssdsGR this value leaks to the p servers, but not to the n banks. In the case where Deadlock is output then the number of iterations reveals the maximum number of transactions with a given source. Thus this value does reveal some information about the distribution of transactions between banks at any given point in time, but we feel it is an acceptable leakage. Removing this leakage is possible, by essentially looping obliviously, in the case where we have type = ssdsGR, for a maximum number of iterations. This would make our implementation match the specifications of Fig. 5 for ssdsGR without inducing the aforementioned leakage, however, this would cause a huge performance penalty.

3.2 Experiments

We implemented the above three variants using the SCALE MPC system [3], which provides a convenient interface to access different secret sharing-based access structures, as well as the comparison and Demux operations discussed above. 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. Of course we can outsource the computation to more than $p = 3$ entities, which will allow the n servers to tolerate more malicious entities, but this will increase the cost of the computation. We used three machines to perform the experiments, that is one machine for each entity. The machines used are 128GB of RAM, with an Intel i-9900 CPU, with a ping time of 0.098 ms between them.

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 $\{8, 64, 128, 256, 512, 1024\}$, 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 $m/4$ iterations.

sodoGR: The runtimes for this variant are shown in Fig. 9. As one can notice, the runtimes keep increasing when n increases, and stabilize when n becomes bigger than m; reaching a maximum of 5.7 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 $n = m$.

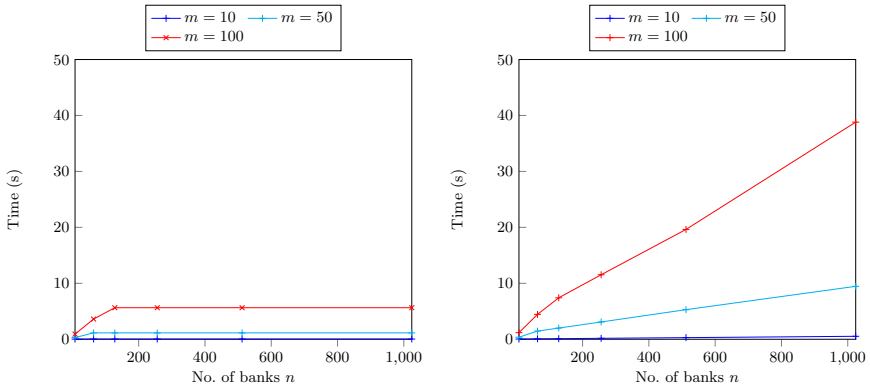

Fig. 9. Run times for sodoGR (left) and sodsGR (right).

sodsGR: The runtimes for this experiment are also presented in Fig. 9. One sees that they grow as n increases, irrespective of m, when $m = 100$ growing to around 39 seconds for $n = 1024$. 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 over a bigger set for n, 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 2.IV of the Balance algorithm in Fig. 6 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 2.III of the same algorithm.

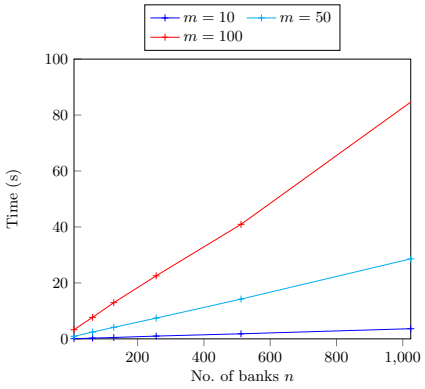

Fig. 10. Run times for ssdsGR.

ssdsGR: The runtimes for this variant are shown in Fig. 10. As one can see the runtimes keep increasing at the same rate even when $m \leq n$. This is related to the fact that sources are hidden from the algorithms. The runtimes are larger for this variant 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. Another thing that one can notice is that, the runtimes of ssdsGR are only twice the ones of sodsGR. This is actually related to how we chose transactions as explained earlier. That is, we chose transactions in such a way that we run $m/4$ iterations in step 5.VI/5.VII. We refer the reader to the next section for a more accurate comparison between the two algorithms.

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 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 the case that bank A sends to bank B multiple transactions, these are counted with only one link or weight. 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 have 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 [18] for more details about the structural properties of scale-free graphs). In particular [33] shows that the Fedwire system in the United States has a scale-free behavior, and similar observations have been made for the Japanese inter-bank payment system [26] and the Austrian inter-bank market [10]. Even the transaction graph formed by Bitcoin has been shown to be scale-free [1].

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 (which we take to be 0.1), and a value β controlling the amount of liquidity in the system.

For our work, we utilized the simulator of [34] 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 [34]. 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 is 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 $B^i = \beta \cdot (U_i - L_i) + L_i$.

In [34] 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 ap-

proximately $n = 7300$ participants[2] and the average daily volume of transfers in January 2021[3] was $803,413$ transactions. A smaller system is the Target2 system run by the European Central Bank, according to the annual report of 2019[4], 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 then add all transactions, which have arrived between the start of the previous execution of the GR algorithm and the end of the previous execution, 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 Fig. 11.

Leakage Again we need to consider if there is any leakage of information from the Open operations in Fig. 11. The two lines, in steps 2.IV.A.iii and 2.IV.B.iii, open a value which indicates whether (in the cases of type = sodoGR and type = sodsGR respectively) a transaction can be executed immediately, without needing to be passed into a gridlock resolution process. This is something which our ideal functionality also leaks.

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] https://www.federalreserve.gov/paymentsystems/fedfunds_about.htm
[3] https://www.frbservices.org/resources/financial-services/wires/volume-value-stats/monthly-stats.html
[4] https://www.ecb.europa.eu/pub/targetar/html/ecb.targetar2019.en.html

Simulation of RTGS

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 type = 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
        1. $\langle B^s \rangle \leftarrow \langle B^s \rangle - \langle a \rangle$.
        2. $\langle B^r \rangle \leftarrow \langle B^r \rangle + \langle a \rangle$.
        3. Execute $\mathsf{Notify}(\{1\}, \{t\}, \emptyset, \emptyset, \mathsf{sodoGR})$.
      (v) Else $Q \leftarrow Q \cup \{t\}$.
    (B) Else if type = 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 type = ssdsGR:
      (i) $Q \leftarrow Q \cup \{t\}$.
  (V) If type = 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 type = 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 type = 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.

Fig. 11. Simulation of RTGS

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.

			Version 1		Version 2	
			E	D	E	D
sodoGR	$\beta = 0.1$	$n = 100$, $M = 900$, $o = 10$	0	0	0	0
		$n = 100$, $M = 9000$, $o = 100$	0	307	0	0
		$n = 100$, $M = 45000$, $o = 500$	-	-	0	0
		$n = 1000$, $M = 9900$, $o = 10$	-	-	0	1
	$\beta = 0.5$	$n = 100$, $M = 900$, $o = 10$	0	0	0	0
		$n = 100$, $M = 9000$, $o = 100$	0	0	0	0
		$n = 100$, $M = 45000$, $o = 500$	-	-	0	0
		$n = 1000$, $M = 9900$, $o = 10$	-	-	0	0
	$\beta = 0.9$	$n = 100$, $M = 900$, $o = 10$	0	0	0	0
		$n = 100$, $M = 9000$, $o = 100$	0	0	0	0
		$n = 100$, $M = 45000$, $o = 500$	-	-	0	0
		$n = 1000$, $M = 9900$, $o = 10$	-	-	0	0
sodsGR	$\beta = 0.1$	$n = 100$, $M = 900$, $o = 10$	0	0	0	0
		$n = 100$, $M = 9000$, $o = 100$	18707	12030	0	0
		$n = 100$, $M = 45000$, $o = 500$	-	-	0	2
		$n = 1000$, $M = 9900$, $o = 10$	-	-	0	7
	$\beta = 0.5$	$n = 100$, $M = 900$, $o = 10$	0	0	0	0
		$n = 100$, $M = 9000$, $o = 100$	0	163	0	0
		$n = 100$, $M = 45000$, $o = 500$	-	-	0	0
		$n = 1000$, $M = 9900$, $o = 10$	-	-	0	5
	$\beta = 0.9$	$n = 100$, $M = 900$, $o = 10$	0	0	0	0
		$n = 100$, $M = 9000$, $o = 100$	0	0	0	0
		$n = 100$, $M = 45000$, $o = 500$	-	-	0	0
		$n = 1000$, $M = 9900$, $o = 10$	-	-	0	0
ssdsGR	$\beta = 0.1$	$n = 100$, $M = 900$, $o = 10$	0	63	0	0
		$n = 100$, $M = 9000$, $o = 100$	164556	102046	0	13
		$n = 100$, $M = 45000$, $o = 500$	-	-	1175	835
		$n = 1000$, $M = 9900$, $o = 10$	-	-	7426	5427
	$\beta = 0.5$	$n = 100$, $M = 900$, $o = 10$	0	0	0	0
		$n = 100$, $M = 9000$, $o = 100$	28646	14442	0	2
		$n = 100$, $M = 45000$, $o = 500$	-	-	0	8
		$n = 1000$, $M = 9900$, $o = 10$	-	-	4482	1784
	$\beta = 0.9$	$n = 100$, $M = 900$, $o = 10$	0	0	0	0
		$n = 100$, $M = 9000$, $o = 100$	0	0	0	0
		$n = 100$, $M = 45000$, $o = 500$	-	-	0	0
		$n = 1000$, $M = 9900$, $o = 10$	-	-	0	105

Table 1. Experimental Results

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. 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.

Acknowledgments

We would like to thank Cedric Humbert of the European Central Bank for suggesting we look into this problem, and answering various questions we had along the way. This work has been supported in part by ERC Advanced Grant ERC-2015-AdG-IMPaCT, by the FWO under an Odysseus project GOH9718N, and by CyberSecurity Research Flanders with reference number VR20192203.

References

1. A, A., L, F., V, M., LY, C.: Fitness preferential attachment as a driving mechanism in bitcoin transaction network. PLoS One 14 (2019)
2. Abbe, E., Khandani, A.E., Lo, A.W.: Privacy-preserving methods for sharing financial risk exposures. CoRR abs/1111.5228 (2011)
3. Aly, A., Cong, K., Keller, M., Orsini, E., Rotaru, D., Scherer, O., Scholl, P., Smart, N.P., Tanguy, T., Wood, T.: SCALE and MAMBA v1.14: Documentation (2021), https://homes.esat.kuleuven.be/~nsmart/SCALE/Documentation.pdf
4. Asharov, G., Balch, T.H., Polychroniadou, A., Veloso, M.: Privacy-preserving dark pools. In: Seghrouchni, A.E.F., Sukthankar, G., An, B., Yorke-Smith, N. (eds.) Proceedings of the 19th International Conference on Autonomous Agents and Multiagent Systems, AAMAS '20, Auckland, New Zealand, May 9-13, 2020. pp. 1747–1749. International Foundation for Autonomous Agents and Multiagent Systems (2020)
5. Bag, S., Hao, F., Shahandashti, S.F., Ray, I.G.: SEAL: sealed-bid auction without auctioneers. IEEE Trans. Inf. Forensics Secur. 15, 2042–2052 (2020)
6. Bech, M., Soramaki, K.: Gridlock resolution in payment systems. Danmarks Nationalbank Monetary Review (07 2001)
7. Bogdanov, D., Talviste, R., Willemson, J.: Deploying secure multi-party computation for financial data analysis - (short paper). In: Keromytis, A.D. (ed.) FC 2012: 16th International Conference on Financial Cryptography and Data Security. Lecture Notes in Computer Science, vol. 7397, pp. 57–64. Springer, Heidelberg, Germany, Kralendijk, Bonaire (Feb 27 – Mar 2, 2012)
8. Bogetoft, P., Christensen, D.L., Damgård, I., Geisler, M., Jakobsen, T., Krøigaard, M., Nielsen, J.D., Nielsen, J.B., Nielsen, K., Pagter, J., Schwartzbach, M.I., Toft, T.: Secure multiparty computation goes live. In: Dingledine, R., Golle, P. (eds.) FC 2009: 13th International Conference on Financial Cryptography and Data Security. Lecture Notes in Computer Science, vol. 5628, pp. 325–343. Springer, Heidelberg, Germany, Accra Beach, Barbados (Feb 23–26, 2009)

9. Bogetoft, P., Damgård, I., Jakobsen, T., Nielsen, K., Pagter, J., Toft, T.: A practical implementation of secure auctions based on multiparty integer computation. In: Di Crescenzo, G., Rubin, A. (eds.) FC 2006: 10th International Conference on Financial Cryptography and Data Security. Lecture Notes in Computer Science, vol. 4107, pp. 142–147. Springer, Heidelberg, Germany, Anguilla, British West Indies (Feb 27 – Mar 2, 2006)
10. Boss, M., Elsinger, H., Summer, M., Thurner, S.: Network topology of the interbank market. Quantitative Finance 4(6), 677–684 (2004)
11. Byrd, D., Polychroniadou, A.: Differentially private secure multi-party computation for federated learning in financial applications. In: ICAIF '20: The First ACM International Conference on AI in Finance, New York, NY, USA, October 15-16, 2020. pp. 16:1–16:9 (2020)
12. Cao, S., Yuan, Y., De Caro, A., Nandakumar, K., Elkhiyaoui, K., Hu, Y.: Decentralized privacy-preserving netting protocol on blockchain for payment systems. In: Bonneau, J., Heninger, N. (eds.) FC 2020: 24th International Conference on Financial Cryptography and Data Security. Lecture Notes in Computer Science, vol. 12059, pp. 137–155. Springer, Heidelberg, Germany, Kota Kinabalu, Malaysia (Feb 10–14, 2020)
13. Cartlidge, J., Smart, N.P., Alaoui, Y.T.: Multi-party computation mechanism for anonymous equity block trading: A secure implementation of turquoise plato uncross. Cryptology ePrint Archive, Report 2020/662 (2020), https://eprint.iacr.org/2020/662
14. Cartlidge, J., Smart, N.P., Talibi Alaoui, Y.: MPC joins the dark side. In: Galbraith, S.D., Russello, G., Susilo, W., Gollmann, D., Kirda, E., Liang, Z. (eds.) ASIACCS 19: 14th ACM Symposium on Information, Computer and Communications Security. pp. 148–159. ACM Press, Auckland, New Zealand (Jul 9–12, 2019)
15. Catrina, O., de Hoogh, S.: Improved primitives for secure multiparty integer computation. In: Garay, J.A., Prisco, R.D. (eds.) SCN 10: 7th International Conference on Security in Communication Networks. Lecture Notes in Computer Science, vol. 6280, pp. 182–199. Springer, Heidelberg, Germany, Amalfi, Italy (Sep 13–15, 2010)
16. Catrina, O., Saxena, A.: Secure computation with fixed-point numbers. In: Sion, R. (ed.) FC 2010: 14th International Conference on Financial Cryptography and Data Security. Lecture Notes in Computer Science, vol. 6052, pp. 35–50. Springer, Heidelberg, Germany, Tenerife, Canary Islands, Spain (Jan 25–28, 2010)
17. Chapman, J., ans Scott Hendry, R.G., McCormack, A., McMahon, W.: Project Jasper: Are distributed wholesale payment systems feasible yet? https://www.finextra.com/finextra-downloads/newsdocs/fsr-june-2017-chapman.pdf
18. Cohen, R., Havlin, S., Ben-Avraham, D.: Structural properties of scale-free networks, pp. 85–110. Wiley (2003)
19. Cozzo, D., Smart, N.P., Alaoui, Y.T.: Secure fast evaluation of iterative methods: With an application to secure PageRank. In: Paterson, K.G. (ed.) Topics in Cryptology – CT-RSA 2021. Lecture Notes in Computer Science, vol. 12704, pp. 1–25. Springer, Heidelberg, Germany, Virtual Event (May 17–20, 2021)
20. Damgård, I., Fitzi, M., Kiltz, E., Nielsen, J.B., Toft, T.: Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In: Halevi, S., Rabin, T. (eds.) TCC 2006: 3rd Theory of Cryptography Conference. Lecture Notes in Computer Science, vol. 3876, pp. 285–304. Springer, Heidelberg, Germany, New York, NY, USA (Mar 4–7, 2006)
21. ECB: Target 2. https://www.ecb.europa.eu/paym/target/target2/html/index.en.html

22. 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
23. Galbiati, M., Soramaki, K.: Liquidity-saving mechanisms and bank behaviour. Bank of England, Bank of England working papers (07 2010)
24. Guntzer, M.M., Jungnickel, D., Leclerc, M.: Efficient algorithms for the clearing of interbank payments. European Journal of Operational Research 106(1), 212–219 (1998)
25. Hastings, M., Falk, B., Tsoukalas, G.: Privacy-preserving network analytics (08 2020)
26. Inaoka, H., Ninomiya, T., Taniguchi, K., Shimizu, T., Takayasu, H.: 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 (Apr 2004)
27. Keller, M., Scholl, P.: Efficient, oblivious data structures for MPC. In: Sarkar, P., Iwata, T. (eds.) Advances in Cryptology – ASIACRYPT 2014, Part II. Lecture Notes in Computer Science, vol. 8874, pp. 506–525. Springer, Heidelberg, Germany, Kaoshiung, Taiwan, R.O.C. (Dec 7–11, 2014)
28. Monetary Authority of Singapore: Project Ubin: Central bank digital money using distributed ledger technology. https://www.mas.gov.sg/schemes-and-initiatives/Project-Ubin
29. Parkes, D.C., Rabin, M.O., Shieber, S.M., Thorpe, C.: Practical secrecy-preserving, verifiably correct and trustworthy auctions. Electron. Commer. Res. Appl. 7(3), 294–312 (2008)
30. Sangers, A., van Heesch, M., Attema, T., Veugen, T., Wiggerman, M., Veldsink, J., Bloemen, O., Worm, D.: Secure multiparty PageRank algorithm for collaborative fraud detection. In: Goldberg, I., Moore, T. (eds.) FC 2019: 23rd International Conference on Financial Cryptography and Data Security. Lecture Notes in Computer Science, vol. 11598, pp. 605–623. Springer, Heidelberg, Germany, Frigate Bay, St. Kitts and Nevis (Feb 18–22, 2019)
31. Shafransky, Y.M., Doudkin, A.A.: An optimization algorithm for the clearing of interbank payments. Eur. J. Oper. Res. 171(3), 743–749 (2006)
32. Smart, N.P., Wood, T.: Error detection in monotone span programs with application to communication-efficient multi-party computation. In: Matsui, M. (ed.) Topics in Cryptology – CT-RSA 2019. Lecture Notes in Computer Science, vol. 11405, pp. 210–229. Springer, Heidelberg, Germany, San Francisco, CA, USA (Mar 4–8, 2019)
33. Soramaki, K., Bech, M.L., Arnold, J., Glass, R.J., Beyeler, W.E.: The topology of interbank payment flows. Physica A: Statistical Mechanics and its Applications 379(1), 317 – 333 (2007)
34. Soramaki, K., Cook, S.: Sinkrank: An algorithm for identifying systemically important banks in payment systems. Economics: The Open-Access, Open-Assessment E-Journal 7 (06 2013)
35. Wang, X., Xu, X., Feagan, L., Huang, S., Jiao, L., Zhao, W.: 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. pp. 614–621. IEEE Computer Society (2018)
36. Yao, A.C.C.: How to generate and exchange secrets (extended abstract). In: 27th Annual Symposium on Foundations of Computer Science. pp. 162–167. IEEE Computer Society Press, Toronto, Ontario, Canada (Oct 27–29, 1986)