Supporting Private Data on Hyperledger Fabric with Secure Multiparty Computation

F. Benhamouda, S. Halevi, T. Halevi

Hyperledger Fabric is a "permissioned" blockchain architecture, providing a consistent distributed ledger, shared by a set of "peers" that must all have same view of its state. For many applications it is desirable to enable keeping private data on the ledger, but the same-view principle makes it challenging to implement. In this work we explored supporting private data on Fabric using secure multiparty computation (MPC). In our solution, peers encrypt their private data before storing it on the chain, and use secure MPC whenever such private data is needed in a transaction. We created a demo of our solution, implementing a bidding system where sellers list assets on the ledger with a secret reserve price, and bidders publish their bids on the ledger but keep secret the bidding price. We implemented a smart contract that runs the auction on this secret data, using a simple secure-MPC protocol that was built using the EMP-toolkit library. We identified two basic services that should be added to Hyperledger Fabric to support our solution, inspiring follow-up work to implement and add these services to the Hyperledger Fabric architecture.

A preliminary version of this article appeared in the first IEEE Workshop on Blockchain Technologies and Applications, BTA 2018. This is a revised and expanded version.

1 Introduction

A blockchain is a distributed system for recording history of transactions on a shared ledger, providing *consistency* (i.e., all participants have the same view of the ledger) and *immutability* (i.e., once something is accepted to the ledger, it cannot change). First popularized for cryptocurrencies such as Bitcoin [10], blockchain technology today is gaining momentum in other areas as well and is touted by some as a disruptive change akin to open-source software [9] or even the Internet itself [12]. The Hyperledger Fabric [5] is a *permissioned* blockchain, namely a closed system where one has to obtain some credentials in order to read the ledger or write to it. The participants that are allowed to read/write to the ledger in Hyperledger Fabric are called *peers* (and typically there are only a few of them). This setting makes it easier to control the transactions on the ledger and is typically faster than public blockchains that are used in most cryptocurrencies.

Nearly all blockchain architectures support the notion of *smart contracts*, namely a programmable application logic that is invoked for every transaction. In Hyperledger Fabric, these smart contracts are implemented via a *chaincode*, which can be an arbitrary program (e.g., in Go), executed by (some of) the peers before a transaction can be recorded on the ledger. The chaincode has access to the current ledger and to the details of the new transaction, and it decides whether or not that transaction will go through, and what data to add to the ledger.

1.2 Blockchain with Private Data

In many application scenarios, we would like to use a blockchain architecture in a setting where some information is private to some participants and should not be seen by others. For example, imagine an insurance market in which insurers want to pool their data together to detect fraud. They may want to discover instances where the same person is buying policies with many insurers over a short period of time or submit multiple claims to different insurers for the same incident. Here too we could consider keeping policy and claim information on a joint ledger in a privacy-preserving format, and periodically running a fraud-detection smart contract to look for fraud patterns. As in the previous example, the smart contract should have logical access to all the data in order to look for suspicious patterns, but it should be implemented without revealing private data of one insurance company to any of its competitors.

Putting private data on the ledger comes with an inherent dilemma: If everyone sees the same ledger, how can we have private data that some can see but others cannot? A common solution in many systems is to put on the ledger only an encryption (or a hash) of the private data, while keeping the data itself under the control of the party that owns it. Of course, this solution on its own is not enough if the smart contracts depend in any way on the private data (as in the example above). Several existing systems offer partial solutions:

HYPERLEDGER FABRIC CHANNELS. Hyperledger Fabric implements *channels*, which are essentially separate ledgers. The data on a channel is only visible to the members of that channel, but not to other peers in the system. This solution provides some measure of privacy (from non-member peers), but it still requires that all members of a channel trust each other with all the data on this channel.

ZERO-KNOWLEDGE PROOFS. Zero-Knowledge proofs (ZKP) [4] allow a prover to convince others that a certain statement is true, without revealing any additional information. Using ZKPs is useful when the smart contract depends on the private data of a

single participant: The party who knows the secret can run the smart contract on its own, and then prove to everyone else that it did so correctly. For example, in a setting where participants have accounts with secret balance on the ledger, a participant wishing to buy a $100-item can use ZKP to prove that its balance is greater than $100. One examples of this approach is the Zcash currency [17], that supports a very general form of ZKPs.

However, ZKPs are not sufficient in settings where the smart contract depends on the secret information of more than one participant. For example, if we have one user with secret balance and another with secret reserve price for an item, ZKPs on their own are not enough for checking if the balance of the first user is bigger than the reserve price of the second.

BLOCKSTREAM CONFIDENTIAL ASSETS (CA). Blockstream CA [13] use simple ZKPs in conjunction with additive homomorphic commitments to manipulate secret data on the ledger. For example, two users whose secret account balances are encrypted with additively homomorphic commitments, can agree privately (off chain) on a price of an item. The first user can then subtract this amount from her balance and add it to the balance of the other user (using homomorphism) and prove to everyone (using ZKP) that the amount added to the second balance is equal to the amount subtracted from the first. But note that the transaction amount itself must be fully known to the first party, this combination of ZKP and additively homomorphic commitments is still not strong enough to compare two secret values.

SOLIDUS. Solidus [2] is a system for confidential transactions on public blockchains, aiming to hide not only the details of the different transactions but also the participants in those transactions. Designed for banking environments, it uses publicly-verifiable Oblivious-RAM (which combines ZKPs with Oblivious RAM) to hide the identities of the individual bank customers. Similar to other ZKP-based solutions, Solidus is designed for settings where each transaction depends only on secrets of one participant (i.e., one of the banks).

HAWK. Hawk [8] is an architecture for a blockchain that can support private data. It uses a trusted component (called a manager) to handle that secret data, which is realized using trusted hardware (such as Intel SGX). The Hawk paper remarks that secure-MPC protocols can also be used to implement the manager but chose not to explore that option in their context (with very many parties).

ENIGMA. The Enigma system [18, 19] uses secure-MPC protocols to implement support for private data on a blockchain architecture. The main difference between our solution and Enigma is that we integrate secure-MPC protocols within the blockchain architecture itself, while Enigma uses *off-chain computation* for that purpose. We discuss the pros and cons of these approaches in Section 1.2.1 below, here we just note that on-chain computation seems like a better match for a permissioned blockchain such as Hyperledger Fabric.

1.2 Our Work

In this work we investigated using secure-MPC protocols for supporting private data on Hyperledger Fabric, integrating the execution of the secure-MPC protocol as part of the smart contract.

Cryptographic secure-MPC techniques, developed since the 80's [3, 15], allow mutually suspicious parties to compute a joint function on their secret inputs, arriving at the right outcome without having to reveal the inputs to each other. A good way of thinking about such protocols is that they mimic the security guarantees that we could get by having a trusted party do the computation on behalf of the participants. But of course, this trusted party is merely virtual, replaced by cryptographic messages that are sent between the actual parties in the protocol. The last decade saw many advances in practical protocols for cryptographic secure computation, and this technology is now efficient enough to handle many real-life workloads.

In our solution, the parties store their private data on the ledger, encrypted with their own secret key (using symmetric-key encryption). When private data is needed in a smart contract, the party who has the key decrypts it and uses the decrypted value as its local input to the secure-MPC protocol. This allows the smart contract to depend on any combination of public and private data from the ledger.

1.2.1 On-Chain Secure-MPC

Differently than systems such as Enigma [19], our approach integrates secure-MPC protocols into the blockchain architecture itself rather than having separate nodes that run it off-chain. Our approach seems to be a better match for a permissioned blockchain such as Hyperledger Fabric, where the peers are often associated with "semantically meaningful" entities that have a stake in the data on the ledger. Indeed, the underlying trust model in a permissioned blockchain is essentially the same as the one used in secure-MPC protocols, i.e., mutually-suspicious parties that communicate to accomplish a common goal. For example, in the insurance scenario from above, it is likely that each peer in the system will belong to one company, and hence will have some data that it can see but the other peers cannot. Having the same peers that write to the ledger also execute the secure-MPC protocol allows us to align the trust models, resulting in a more manageable (and more secure) system.

Moreover, running the secure-MPC protocol on-chain allows us to use the blockchain facilities in the protocol itself. For example, we can use the blockchain facilities for identity management and communication (or even use an existing implementation of a consensus protocol to implement a broadcast channel that may be needed in the protocol). Delegating the secure-MPC protocol to an off-chain component would mean re-implementing these facilities for that new component.

The main argument against using on-chain secure-MPC protocols is that the inefficiencies of the protocol and the blockchain may compound each other, but this argument applies more to permissionless blockchain (that are typically slower than permissioned ones). In our (limited) experiments with a simple secure-MPC protocols, the cost of the secure-MPC protocol was quite small (and an optimized version can be made even much faster). See some details in Section 3.3.

1.2.2 Our Demo

To help drive our investigation, we implemented a demo of a simple bidding scenario, in which reserve prices and bids are secret (and all other auction details are public). The smart contract implements a 1st-price seal-bid auction mechanism, where the participants learn nothing but the result (and in particular do not learn the losing bids nor the reserve price of the seller).

In the rest of this report we give more details about our architecture and implementation. In Section 2 we describe the system architecture and how it is integrated into Hyperledger Fabric and discuss the changes that are needed to get a production system. In Section 3 we give more details of the demo itself, including the secure-MPC protocol that we used and the user-interface aspects.

2 On-Chain Secure-MPC in Hyperledger Fabric

2.1 Basic Concepts of Hyperledger Fabric

In Hyperledger Fabric, the nodes that have access to the ledger are called peers, and each peer belongs to some organization. Adding transactions to Fabric is a two-phase process: A client requesting a transaction first approaches one or more peers with a *transaction proposal* and asks them to execute and endorse the proposal. The endorsing peers then execute a smart contract — called a *chaincode* in Fabric — to determine whether or not to endorse the transaction, and if so, then how this transaction changes the state on the ledger. A relevant detail for our purposes is that all endorsers must see an identical transaction proposal (else it is rejected in the next phase). Since the "logical validity" of transactions is determined in the endorsement phase, we chose to run the secure-MPC protocols during that phase.

Once sufficiently many endorsements are obtained, the client sends the endorsed transaction to an ordering service, that imposes a linear order on the transactions and then actually adds them to the ledger. The number of required endorsements for a transaction is determined by an *endorsement policy*, which is set when the ledger is initialized. Some example policies are "at least one endorser," "at least two endorsers from two different organizations," etc. Usually, the ledger has only a single endorsement policy that applies to all the transactions in it.

2.2 Two Crucial Additional Components

To support transactions that depend on private data, we needed to add two components to Fabric:

LOCAL CONFIGURATION. To deal with data that is only visible to some peers but not others, the chaincode implementing the endorsement logic at the different peers should have access to local parameters that are not available to other peers. At the very least, the chaincode running at a certain peer needs access to the secret key of the organization of that peer.

INTER-PEER COMMUNICATION. Another component that we need is communication between peers during endorsement. Namely, the chaincode running at one peer must communicate with the same chaincode running at other peers, so that information about private data could impact the endorsement decision of peers who do not see that data.

In our demo, we implemented these components using a "helper server" that we developed in Go. The helper server stores the local parameters of each peer and facilitates setting up communication channels between instances of the chaincode at different peers. The chaincode running inside a peer communicates with the helper server, whose address we hard-coded in the chaincode itself. Communication between the chaincode instances and the helper server is done via gRPC, a remote procedure call framework which is used extensively in Fabric.

We stress that our helper server was a proof-of-concept construct that allowed us to implement the above two components over Fabric 1.0. This server cannot be used in a production system, as it is a trusted party with access to all the secrets. We opened a JIRA issue for adding these two components to Fabric [6]. Building on our demo experience, follow up work (by us and others) explored ways to integrate these two components into Fabric in a secure way, see [1].

2.3 Fabric-Specific Implementation Details

ENCRYPTED DATA ON THE LEDGER. As explained earlier, we keep private data on the ledger in encrypted form, under keys that are only available to the peers that are supposed to see it. We thus need to deal with the question of how to put such encrypted data on the ledger in the first place. Recall that the only way to put data on the ledger is for a client to send a transaction proposal to some peers, and all these peers must see an identical proposal.[1] If the ledger policy requires endorsement from peers in different organizations (cf. Section 2.4), then the only way to keep data hidden from some peers is for the client to encrypt the data before including it in the proposal. Hence this solution requires that (some) clients have access to the encryption keys.

One option to implement client-side encryption would be to use public-key encryption, where clients use the public encryption keys of the relevant organizations to encrypt the private data, and the endorsing peers use the corresponding secret decryption keys to recover the private data for use in the secure-MPC protocol. In our demo, however, we used a simpler solution, where per-organization "privileged clients" are given access to the symmetric keys that these organizations use to encrypt their private data, the same keys that the peers of those organization use to decrypt values during the endorsement phase. Either way, deploying this type of solution in a production system would require proper key-management, to ensure that only authorized components get access to cryptographic keys.

SOFTWARE COMPONENTS. While the chaincode in Fabric is usually written in Go, most cryptographic libraries for secure-MPC protocols (including EMP-toolkit [14], the library we are using) are written in C++. To call EMP-toolkit from the Go

[1] Recent versions of Hyperledger Fabric weakened this restriction by allowing to not record the full proposal on the blockchain and to have transient data.

chaincode, we use SWIG, which allows calling C++ code from other languages.

To add support for SWIG and EMP-toolkit, we patched the FabricSDK for Node.js so that the SWIG files (`*.cpp, *.hpp, *.swigcxx`) are included in the chaincode package to be installed. We also use a customized build environment (i.e., a customized Docker container `fabric-ccenv`, specified by the environment variable `CORE_CHAINCODE_BUILDER`), that includes SWIG and EMP-toolkit.

COMMUNICATION CHANNELS. Another integration issue that we had to solve is that EMP-toolkit normally uses its own communication channels (using UNIX sockets). To use it within Fabric, however, we had to be able to use the Fabric communication mechanisms. We thus implemented new synchronous channels for EMP-toolkit on top of gRPC (currently using the helper server). Our channels are created in the chaincode in Go and passed to EMP-toolkit using SWIG.

THE ENDORSING PEERS. In the Fabric architecture, it is the client's responsibility to choose the endorsing peers for its transactions, and our application is no exception. In our case, it is important that the client chooses peers that collectively can decrypt all the private fields that the transaction depends on. Also, the client in our case must tell the peers about each other, since each peer must know the identities of the other peers in order to run a secure-MPC protocol with them. For the demo, this information is stored in the configuration file of the client.

2.4 Security Considerations

Below we discuss several security-related aspects that we did not address in our demo implementation, but that must be addressed in any production system.

ENDORSEMENT POLICIES. It may be important to align the trust model of the secure-MPC protocol with that of the endorsement policy in the ledger. For instance, if the trust model of the protocol assumes at most t adversarial parties, we may want to set a policy that requires more than t endorsers, ensuring that an invalid transaction will never be endorsed within the trust model. (The alignment of trust models is less important in settings where we can assume that parties are honest-but-curious, since an honest-but-curious party will not endorse an invalid transaction.)

As another example, we may want to set the endorsement policy to ensure that the bid records and items of an organization cannot be modified without endorsement of that organization. We may also want to ensure that items cannot be changed at all without the endorsement of a large enough number of organizations, to prevent the seller from changing the item from a very valuable one to a very cheap one just before the auction and to ensure that the result of an auction (and resulting change of ownership) cannot be faked. At the time the demo was developed, the policy language used in (the CLI interface of) Fabric could not specify such constraints, so the only "safe setting" that could be expressed in that policy language was requiring that every transaction be endorsed by all organizations (which may make the endorsement process very slow).

Fabric v1.3 introduced the notion of *key-level endorsement policies*, for specifying different endorsement policies for different variables (keys) on the ledger, which can be used also in our setting. For example, the default endorsement policy is set to require only a single endorsement, so that any organization can post an item or a bid record to the ledger without the help of anyone else. Each item is created with a specific policy that says that it can only be modified with endorsement from all (or many) organizations, the chaincode checks that the endorsement policy of an item requires endorsement from all (many) organizations before it places a bid for that item, and each bid is created with a policy that says that only the organization that placed it is allowed to modify it.

CLIENT AUTHORIZATION. A production system must implement appropriate authorization policies for clients. For example, in our demo setting, we may want to designate some per-organization privileged clients that can list new items and trigger auctions for existing items of that organization. Non-privileged clients may still issue queries for the state of the ledger, such as the description of all the items for sale.

ENFORCEMENT. Recall that Fabric transactions are added via a two-phase process, and that the secure-MPC protocol is run in the first phase to let peers decide whether or not to endorse the transaction. This setting, however, allows a rogue peer to first learn the result of the secure-MPC protocol, and then withhold its endorsement if it does not like this result. This is an issue of *fairness*, which is well studied in the literature. One way of addressing it includes using a threshold endorsement policy (so no single peer can block the transaction). Another variant is to implement a commit transaction in which the result is kept secret (e.g., encrypted under a key which is secret-shared by the organizations), followed by a reveal phase where sufficiently many organizations post their keys to the ledger, enabling everyone to decrypt the committed result.

VERIFIABILITY AND AUDIT. Including secret data in the endorsement process makes it harder to verify proper endorsement ex post facto. One way to address this concern is by recording on the ledger noninteractive zero-knowledge proofs of proper endorsement (together with the transaction itself). A cheaper alternative is to allow verification only by privileged auditors, through recording with the transaction also the protocol transcript (or its hash), and have peers keep their private data and randomness to show to the auditor.

3 Demo

Our demo implements a simple 1st-price auction scenario with secret reserve prices and bids. It includes three organizations, called Auctionite Ltd., BuyBuy Corp., and PurrChase Inc., each with a single peer in the system. Each organization can list items with secret reserve prices and can place sealed bids for items listed by the others. All the information about the items is recorded on the ledger, including a unique-ID, description, optional picture, etc. It also includes a cleartext minimum bid amount, and an encrypted reserve price under the key of the listing organization. Similarly, each bid record includes the ID of the item, the identity of the bidder, and an encrypted bid amount.

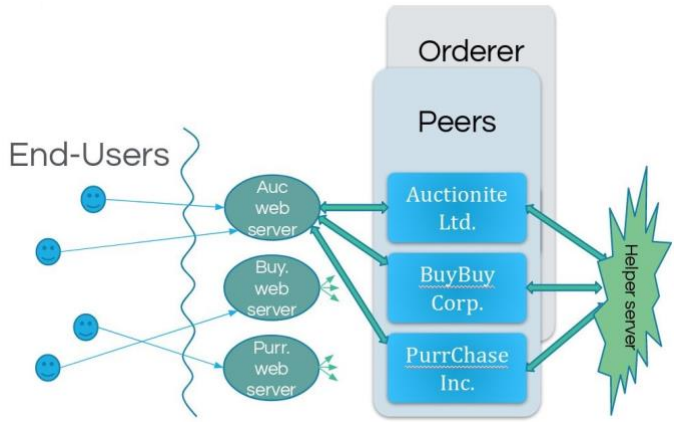

Figure 1 High-level demo architecture. End-users access the system via web servers, which play the role of the Fabric clients. The Fabric back end (peers and orderer) are assisted by our helper server.

When an auction transaction is invoked (by clicking a button in the user interface), all three peers are activated to endorse it, and each peer uses its organization key to decrypt its own secrets off of the ledger. They then run a secure-MPC protocol to determine the highest bid and whether or not it meets the reserve price, and the auction result is published to the ledger. Once the auction took place all the bid records for that auction are marked as "invalid," and if the auction succeeded then the item is marked as "sold," with a new owner and with the sell price. (If the reserve was not met then the item ownership does not change, and the peers are made aware of the failure.)

3.1 Demo Implementation

As illustrated in **Figure 1**, the demo has three layers: A Fabric back end (with our helper server), organization web servers (that play the Fabric clients), and the browser-based end-user interface.

Most details of the Fabric layer were described in Section 2, for the demo we used three organizations with one peer each (and IDs org⟨n⟩.example.com and peer⟨n⟩.org⟨n⟩.example.com, respectively, ⟨n⟩ ∈ {0,1,2}). We utilized a single orderer, as well as the helper server that we used to implement local state and communication channels (cf. Section 2.2).

The end-user interface is browser-based, implemented with the bootstrap framework using HTML 5, CSS, and Javascript. We describe more aspects of it in Section 3.2 below.

In between, we have a layer of web servers, one per organization. On one hand these servers serve the browser-based interface to the end users, and on the other hand they play the role of the Fabric clients, interacting with the peers. This layer was developed with Hyperledger Fabric SDK for Node.js, and uses the hapijs framework and Handlebars.js templates. To simplify coding, in our demo we implemented a single web server that serves the website of all three organizations (but of course a production system would have different web servers for the different organizations).

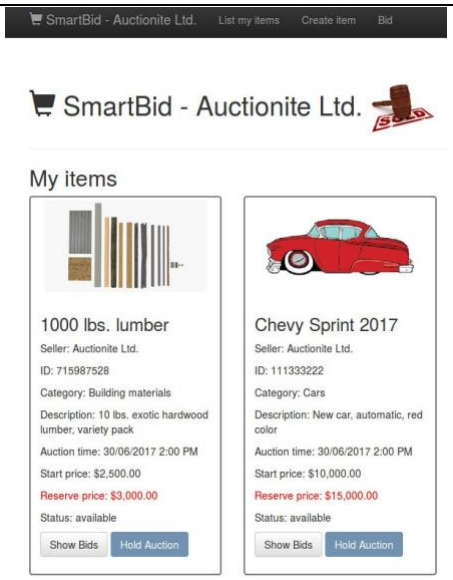

Figure 2 All items listed by one seller.

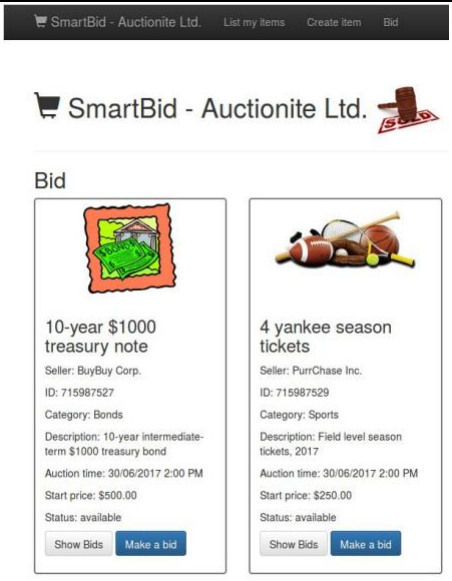

Figure 3 Listings of available items for sale.

3.2 User Interface

In our demo we have identical user interface for the three organizations. The UI lets the end-users create new items, list all the available items, bid on an item belonging to another organization, list all bids for an item, and run auction for an item. Some screenshots are illustrated in **Figures 2 though 4**. The normal flow of an auction is as follows:

First, a seller connects to the website of its organization and creates a new item record, specifying things such as category, description, start price, and reserve price. The reserve price is confidential and is only sent encrypted to the chaincode, and no other party has access to it.

Second, interested buyers connect to the website of their organization, see the list of items and place bids on them. The bid price is also confidential and sent encrypted to the

chaincode.

Third, owner of an item connects to the website to trigger the auction. The web server then contacts one peer from each organization, and they all endorse that transaction, running the secure-MPC protocol to get the result of the auction. The buyer that offered the highest bid will be the winner, as long as this bid is above the reserve price. Otherwise, an appropriate error is returned. The result of the auction is finally committed to the ledger.

3.3 The secure-MPC Protocol

Our demo only handles up to three parties, namely a seller and up to two buyers. As the EMP-toolkit library did not yet support protocols with more than two parties when we did this work, we designed a simple three-party protocol, building on EMP-toolkit's implementation of two-party semi-honest protocols. Our protocol is secure in the semi-honest model, assuming honest majority (i.e., at most one adversarial party).

The seller's private input in the protocol is the reserve price s for the item, and the private inputs of the two bidders are the bid amounts b_1, b_2, respectively. These numbers are all 32-bit integers. The output of the protocol is a ternary value: either 0 if the reserve price was not met, 1 if Bidder 1 won the auction, or 2 if Bidder 2 won. (If the two bids are equal, we arbitrarily let Bidder 1 win the auction.) In more detail, the function that they compute is:

$$f(s, b_1, b_2) = \begin{cases} (0,0) & \text{if } s > \max(b_1, b_2) \quad \text{(Reserve not met)} \\ (1, b_1) & \text{if } b_1 \geq \max(s, b_2) \quad \text{(Bidder 1 won)} \\ (2, b_2) & \text{if } b_2 \geq s,\ b_2 > b_1 \quad \text{(Bidder 2 won)} \end{cases}$$

We built our protocol over the following two tools that are implemented in EMP-toolkit:

- A 1-out-of-2 string oblivious transfer protocol [11] between a receiver with input choice-bit b and a sender with two input strings s_0, s_1. At the conclusion of the protocol, the receiver gets s_b but learns nothing about s_{1-b}, and the sender learns nothing at all.
- Yao's protocol for arbitrary two-input functions [15]. For any two-input function $f: \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$, Yao's protocol enables two parties with respective inputs $a, b \in \{0,1\}^n$, to jointly compute the result $\sigma = f(a, b)$. Both partied learns σ, but nothing else.

Our protocol consists of three main steps:

STEP 1. First the two bidders compare their bids using Yao's protocol for the Millionaires problem [15], where the output is secret-shared among them. Namely at the conclusion of this step they get two output bits x_1 (for Bidder 1) and x_2 (for Bidder 2) that are individually uniform and satisfy:

$$x_1 \oplus x_2 = \{0 \text{ if } b_1 < b_2, \text{ or } 1 \text{ if } b_1 \geq b_2\}.$$ [2]

STEP 2. Next the bidders run two instances of 1-out-of-2 string Oblivious Transfer (OT), with the goal of getting an XOR-sharing of the value $\max(b_1, b_2)$:

In the first OT instance Bidder 2 plays the OT-receiver, using x_2 as her input choice bit. Bidder 1 chooses a random 32-bit string r_1, then plays the OT-sender with r_1 and $r_1 \oplus b_1$ as his two strings, ordered according to x1. Namely if $x_1 = 0$ then Bidder 1 uses the pair $(r_1, r_1 \oplus b_1)$, and otherwise he uses $(r_1 \oplus b_1, r_1)$. The output of Bidder 1 from the protocol is r_1, and the output of Bidder 2 is the received OT string (which we denote r_2). It is easy to check that:

$$r_1 \oplus r_2 = \{0 \text{ if } x_1 \oplus x_2 = 0, \text{ or } b_1 \text{ if } x_1 \oplus x_2 = 1\}.$$

The second instance is symmetric, with the two bidders having output strings r_1', r_2' satisfying

$$r_1' \oplus r_2' = \{b_2 \text{ if } x_1 \oplus x_2 = 0, \text{ or } 0 \text{ if } x_1 \oplus x_2 = 1\}.$$

The two bidders XOR their respective outputs from the two instances, getting $y_1 = r_1 \oplus r_1'$ and $y_2 = r_2 \oplus r_2'$, and indeed $y_1 \oplus y_2 = \max(b_1, b_2)$.

STEP 3. Next, Bidder 1 sends its shares x_1 and y_1 to the seller over a private channel. Then the seller and Bidder 2 engage in another Yao protocol, computing whether the reserve price was met, i.e., whether $(y_1 \oplus y_2) \geq s$. If the reserve was met, then Bidder 2 sends x_2, y_2 to the seller and Bidder 1. They can recover the winning bid $y_1 \oplus y_2$, as well as the winner (which is Bidder 1 if $x_1 = x_2$ and Bidder 2 otherwise). Then they send these two values back to Bidder 2 so everyone learns the result.

SECURITY. Assuming the security of the underlying two-party protocols, the three parties in our protocol indeed learn only the auction result and nothing else. This is true since all the intermediate results that are obtained from the various sub-protocols are always masked by random values. Hence for each individual party, these values appear random and independent of the protocol's true inputs and outputs.

PERFORMANCE. We ran our demo on a Lenovo Carbon X1 machine (4th generation), with Intel Core i5-6300U CPU and 8GB of RAM, running Ubuntu 16.04, where the peers and

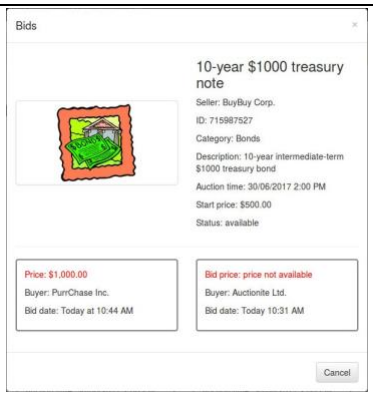

Figure 4 All bids that were made for an item. Bid amounts (in red) are secret, only the bidder can see its own amount.

~~the parties just output the LSB's of the respective output labels~~ they know.

[2] To gain some efficiency we use a feature of the EMP-toolkit implementation that implements free-XOR and half-gates [7, 16]: After running Yao's protocol for the comparison function,

servers were all running on separate docker containers on the same machine. The time of the execution and endorsement of a transaction proposal involving the secure-MPC protocol was about 0.4s, which seems fast enough for the application at hand. This time is measured from the moment the transaction proposal is sent by the client to the three peers up to the moment the transaction responses are received by the client. We speculate that most of this time is due to buffering effects in our communication infrastructure but did not explore this further. There is no doubt that this execution can be made much faster, in particular by improving the communication channels.

4 Conclusion

In this work we investigated supporting private data on Hyperledger Fabric using on-chain secure-MPC protocols. We designed an architecture that supports such private data and implemented a demo auction application that uses it. Our investigation identified two components that should be added to Fabric to enable execution of smart contracts that depend on such private data.

Acknowledgment

We thank Angelo De Caro and Yacov Manevich for all their help with integrating our solution into Fabric. We also thank the anonymous reviewers for their helpful comments.

References

1. F. Benhamouda, A. DeCaro, S. Halevi, T. Halevi, C. jutla, Y. Manevich, and Q. Zhang. Initial public offering (IPO) on permissioned blockchain using secure multiparty computation. https://shaih.github.io/pubs/bdh+18.html, 2018.

2. E. Cecchetti, F. Zhang, Y. Ji, A. E. Kosba, A. Juels, and E. Shi. Solidus: Confidential distributed ledger transactions via PVORM. In Proceedings of ACM-CCS 2017, pages 701–717. ACM, 2017.

3. O. Goldreich, S. Micali, and A. Wigderson. Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. JACM, 38(3):691–729, 1991.

4. S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. SIAM J. on Computing, 18(1):186–208, 1989.

5. Welcome to Hyperledger Fabric. https://hyperledger-fabric.readthedocs.io/, accessed Jan 2018.

6. Supporting private data in Hyperledger. https://jira.hyperledger.org/browse/FAB-5131, 2017.

7. V. Kolesnikov and T. Schneider. Improved garbled circuit: Free XOR gates and applications. In Proceedings of ICALP 2008, Part II, volume 5126 of Lecture Notes in Computer Science, pages 486– 498. Springer, 2008.

8. A. E. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In 2016 IEEE Symposium on Security and Privacy, pages 839–858. IEEE Computer Society Press, May 2016.

9. L. Mearian. What is blockchain? the most disruptive tech in decades. Computerworld, Dec 2017, https://www.computerworld.com/article/3191077/security/what-is-blockchain-themost-disruptive-tech-in-decades.html.

10. S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2008.

11. M. O. Rabin. How to exchange secrets by oblivious transfer. Technical Report TR-81, Aiken Computation Laboratory, Harvard University, 1981.

12. G. Rapier. From Yelp reviews to mango shipments: IBM's CEO on how blockchain will change the world. Business Insider, June 2017, https://www.businessinsider.com/ibm-ceoginni-rometty-blockchain-transactionsinternet-communications-2017-6.

13. A. van Wirdum. "Confidential assets" brings privacy to all blockchain assets: Blockstream. Bitcoin Magazine, April 2017, https://bitcoinmagazine.com/articles/ confidential-assets-brings-privacy-allblockchain-assets-blockstream/.

14. X. Wang, A. J. Malozemoff, and J. Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.

15. A. C.-C. Yao. Protocols for secure computations (extended abstract). In 23rd FOCS, pages 160–164. IEEE Computer Society Press, Nov. 1982.

16. S. Zahur, M. Rosulek, and D. Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In EUROCRYPT 2015, Part II, volume 9057 of LNCS, pages 220–250. Springer, Heidelberg, 2015.

17. Zcash - all coins are created equal. https://z.cash/. Accessed Dec 2017.

18. G. Zyskind, O. Nathan, and A. Pentland. Decentralizing privacy: Using blockchain to protect personal data. In IEEE Symposium on Security and Privacy Workshops, pages 180–184. IEEE Computer Society, 2015.

19. G. Zyskind, O. Nathan, and A. Pentland. Enigma: Decentralized computation platform with guaranteed privacy. CoRR, abs/1506.03471, 2015.

Fabrice Benhamouda *IBM Research, Yorktown Heights, NY 10598, USA (fabrice.benhamouda@normalesup.org).* Dr. Benhamouda obtained a PhD in Computer Science in 2016 from ENS, France. He is a researcher in the cryptography group in IBM T.J. Watson Research Center.

Shai Halevi *IBM Research, Yorktown Heights, NY 10598, USA (shaih@alum.mit.edu).* Dr. Halevi obtained a PhD in Computer Science in 1997 from MIT, MA, USA. He is a researcher in the cryptography group in IBM T.J. Watson Research Center.

Tzipora Halevi *Brooklyn College, Brooklyn, NY 11210, USA (thalevi@nyu.edu).* Prof. Halevi obtained a PhD in Computer Science in 2012 from NYU-Poly, NY, USA. She is an Assistant Professor in the Computer Science department in CUNY Brooklyn College, NY, USA. Part of this work was done while she was a visiting scientist in the cryptography group in IBM T.J. Watson Research Center.