Supporting Private Data on Hyperledger Fabric with Secure Multiparty Computation

(Traditional Paper)

Fabrice Benhamouda
IBM Research
Yorktown Heights, NY, USA
https://www.normalesup.org/∼fbenhamo/

Shai Halevi
IBM Research
Yorktown Heights, NY, USA
https://shaih.github.io/

Tzipora Halevi*
Brooklyn College
Brooklyn, NY, USA
thalevi@nyu.edu

***Abstract*—Hyperledger Fabric is a "permissioned" blockchain architecture, providing a consistent distributed ledger, shared by a set of "peers." As with every blockchain architecture, the core principle of Hyperledger Fabric is that all the peers must have the same view of the shared ledger, making it challenging to support *private data* for the different peers. Extending Hyperledger Fabric to support private data (that can influence transactions) would open the door to many exciting new applications, in areas from healthcare to commerce, insurance, finance, and more.**

In this work we explored adding private-data support to Hyperledger Fabric using secure multiparty computation (MPC). Specifically, in our solution the peers store on the chain encryption of their private data, and use secure MPC whenever such private data is needed in a transaction. This solution is very general, allowing in principle to base transactions on any combination of public and private data.

We created a demo of our solution over Hyperledger Fabric v1.0, implementing a bidding system where sellers can list assets on the ledger with a secret reserve price, and bidders publish their bids on the ledger but keep secret the bidding price itself. We implemented a smart contract (aka "chaincode") that runs the auction on this secret data, using a simple secure-MPC protocol that was built using the EMP-toolkit library. The chaincode itself was written in Go, and we used the `swig` library to make it possible to call our protocol implementation in C++.

We identified two basic services that should be added to Hyperledger Fabric to support our solution, and are now working on implementing them.

Keywords. Blockchain, Hyperledger Fabric, Implementation, Secure Multiparty Computation

I. 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 crypto-currencies such as Bitcoin [8], 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 [7] or even the Internet [9]. The Hyperledger Fabric [5] is a *permissioned* blockchain, where writing to the ledger requires some credentials. The participants that are allowed to 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 transaction on the ledger, and is typically faster than public blockchains that are used in most crypto-currencies.

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 *chaincode*, which can be an arbitrary program (in Go), executed by (some of) the peers. The chaincode has access to the current ledger as well as the details of the new transaction, and it decides whether or not that transaction will go through, and the data to add to the ledger.

A. 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. Some examples of applications that require dealing with private data include the following:

Running Medical Studies. Consider multiple hospitals that want to jointly run statistics on patient treatment data, e.g., the success rate of a treatment option for patients with some rare condition. The data is private so the hospitals cannot share it, and the condition is sufficiently rare that a simple redaction of PII is not sufficient. We may want to keep all the treatment data on a joint ledger (e.g., to facilitate logging and audit requirements), but we must do it in a privacy-preserving manner. Hospitals that want to run a study on their joint data should be able to get the results of that study without violating patient privacy requirements. Namely, we would like to have a smart contract that can logically see all the data, compute the relevant statistics, and publish them to the ledger. But this must be done in a privacy-preserving manner, without any of the individual hospitals being able to see in the clear the private data held by other hospitals.

Detecting Insurance Fraud. For another example, imagine an insurance market in which insurers want to pool their data together to detect fraud. For example they may want to discover instance where the same person is buying policies with many insurers over a short period of time, or

[*] Work partly done while at IBM Research

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.

BIDDING FOR SHIPPING SLOTS. In a setting with multiple ships and multiple delivery companies, we may want to auction off "container slots" on the various ships to the delivery companies. Imagine creating a unified ledger where each ship can record its schedule and rates, and each delivery company can record its needs and how much it is willing to pay. We then could have a smart contract that would logically treat all this information as a big multi-unit auction, and applying some resolution strategy to assign containers to ships and generate the appropriate invoices to the delivery companies. Using the consistency and immutability of the ledger, this yields a price resolution mechanism that carries with it enforcement and audit. Of course the various rates and payment options are very sensitive, so they need to be stored in a manner that will not compromise their secrecy. For the same reason, the smart contract must be implemented in a way that will not divulge the secrets of one company to another.

B. Previous Work on Blockchain with Private Data

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 use-cases 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 channels, 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.

USING 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 enough 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 [12], 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. (Indeed ZKPs are not sufficient for any of the use cases that we sketched before.)

BLOCKSTREAM CONFIDENTIAL ASSETS (CA). Blockstream CA [10] 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, or for any of the use-cases from above.

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 [6] 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 [13], [14] 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 I-C1 below, here we just note that on-chain computation seems like a better match for a permissioned blockchain such as Hyperledger Fabric.

C. 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], [11], 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. 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) On-chain Secure-MPC: Differently than systems such as Enigma [14], 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 typically 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 medical data use-case from above, it is likely that each peer in the system will belong to some hospital, 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 allow us to align the trust models, resulting in a more manageable (and more secure) system.

Moreover, running the secure-MPC protocol on-chain allow 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 III-C.

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 note we give more details about our architecture and implementation. In Section II we describe the system architecture and how it is integrated it into Hyperledger Fabric (v1.0), and discuss the changes that are needed to get a production system. In Section III we give more details of the demo itself, including the secure-MPC protocol that we used and the user-interface aspects.

3) Acknowledgments: We thank Angelo De Caro and Yacov Manevich for all their help with integrating our solution into Fabric.

II. On-Chain Secure-MPC in Hyperledger Fabric

A. 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 from among the five organizations," etc. Roughly speaking, the ledger has only a single endorsement policy that applies to all the transactions in it.

B. 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. For example, the peers often need access to the secret key of their organization.

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. Since the chaincode in Fabric does not even know the peer ID on which it is running, it just sends to the helper server its Docker container ID (from `/proc/1/cpuset`), and the helper server uses the Docker executable to convert it into a container name and extracts the peer name from it.

We note that the helper server is an insecure hack for implementing the above two components, we implemented it as a trusted party with access to all the secrets. This quick-and-dirty hack lets us study the feasibility of our approach without having to change the Fabric architecture itself, and it demonstrates that secure-MPC protocols can be used on-chain in Fabric with just the above two simple new components. Building on our demo experience, we currently are working on integrating these two components into Fabric in a secure way. It seems that we can utilize new features of Fabric 1.1 to ease this integration.

C. 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. If the endorsement policy requires peers from different organizations (cf. Section II-D), 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.

In our demo we used per-organization "privileged clients" that have 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. Another option 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. 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, the library we are using) are written in C++. To call EMP-toolkit from the Go chaincode, we use SWIG, which allows calling C++ code from other languages.

To add support for SWIG and EMP-toolkit, we patched the Fabric SDK for Node.js and added SWIG files (`*.cpp`, `*.hpp`, `*.swigcxx`) to 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.

EMP-toolkit normally uses its own communication channels using UNIX sockets, but to use it within Fabric we implemented new 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 can collectively 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 can run a secure-MPC protocol with them.

D. 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 example 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.)

For another example, we may want to set the endorsement policy to ensure that the secret values of an organization cannot be modified without endorsement of that organization. However, the policy language used in (the CLI interface of) Fabric cannot specify such a constraint. Within the supported policy language, it seems that the only "safe setting" is to require that every transaction be endorsed by all organizations, which may make the endorsement process very slow. Perhaps a good compromise is to require endorsement from (say) at least three organizations. A similar issue is that we (roughly) have to use a single endorsement policy for all the transactions, whereas in many cases we may want to impose different constraints on different actions. For example, in our demo scenario it seems natural to let organizations endorse their new-item transactions on their own, but require that auctions are endorsed by all the participants. We speculate that such non-standard endorsement policies can be implemented in Fabric using a custom “system chaincode”, but did not investigate this option.

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 rouge 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 include using a threshold endorsement policy (so no single peer can block the transaction). We can also implement a commit transaction in which the result is kept secret, followed by a reveal transaction where it is revealed.

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 non-interactive zero-knowledge proofs of proper endorsement (together with the transaction itself). A cheaper alternative is to allow verification only by privileged auditors, by 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.

III. DEMO

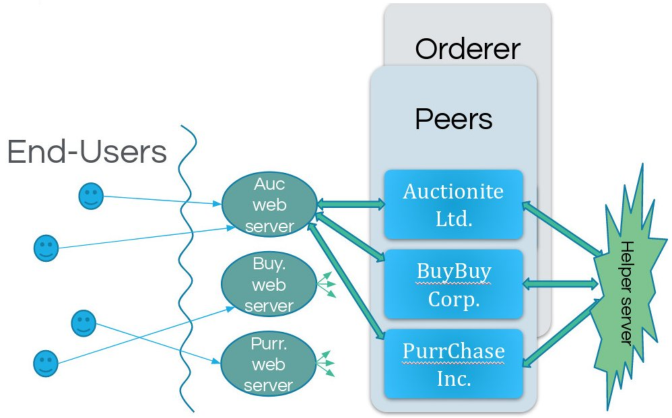

Fig. 1: High-level demo architecture. End-users access the web servers of the different entities. These web-servers play the role of the Fabric clients, talking to Fabric back-end, which is assisted by our helper server.

Our demo implements a simple 1st-price auction scenario with secret reserve prices and bids. It includes three organizations, called AUCIONITE 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 for the listing, a description, an (optional) picture, a category, a cleartext minimum bid amount, an encrypted reserve price (under the key of the listing organization), and the time/date for the auction. Similarly each bid record includes the ID of the item, the identity of the bidder, and an encrypted bid amount.

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. (Namely, for the seller, the reserve price; and for each potential buyer, the amount of the bid.) 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.)

A. Demo Implementation

As illustrated in Fig. 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 II, for the demo we used three organizations with one peer each (and IDs org$\langle \mathrm{n} \rangle$.example.com and peer$\langle \mathrm{n} \rangle$.org$\langle \mathrm{n} \rangle$.example.com, respectively, $\langle \mathrm{n} \rangle \in \{0, 1, 2\}$). We used a single *orderer*, and the helper server that we used to implement local state and communication channels (cf. Section II-B).

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 III-B 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).

B. 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 screen shots are illustrated in the long version of this note [1]. The normal flow of an auction is as follows:

1) 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.
2) 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.
3) The 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.

C. The secure-MPC Protocol

Our demo only handles upto three parties, namely a seller and upto two buyers. In the description below we refer to these parties as Sally the seller, and the bidders Boyd and Debra. We designed a simple three-party protocol for these three parties, building on the implementation of two-party semi-honest protocols in the EMP-toolkit library. Our protocol is secure in the semi-honest model, assuming honest majority (i.e., at most one adversarial party).

a) Input & output.: Sally's input is the reserve price s for the item, and the inputs of the two bidders are b (Boyd) and d (Debra). These numbers are all 32-bit integers.

At the end of the protocol, all parties should receive the output 'trit' whose value is either 0 if the reserve price was not met (or the computation aborted), 1 if Boyd won the auction, or 2 if Debra won the auction. (If Boyd and Debra submit the same bid, we arbitrarily let Boyd win the auction.) Namely, the function that they compute is:

$$f(s,b,d) = \begin{cases} (0,0) & s > \max(b,d) \quad \text{// Reserve not met} \\ (1,b) & b \geq \max(s,d) \quad \text{// Boyd won} \\ (2,d) & d \geq s, d > b \quad \text{// Debra won} \end{cases}$$

The protocol consists of three main steps:

1) First the two bidders compare their bids using Yao's protocol for the Millionaires problem, where the output is secret-shared among them. Namely at the conclusion of this protocol they get two output bits x_b (Boyd) and x_d (Debra) that are individually uniform and satisfy $x_b \oplus x_d = \{0 \text{ if } b < d, \text{ or } 1 \text{ if } b \geq d\}$.
2) Next the bidders runs two instances of 1-out-of-2 string Oblivious Transfer (OT), to get an XOR-sharing of the value $\max(b,d)$:

   In the first instance Debra plays the OT-receiver, using x_d as her choice bit. Boyd chooses a random 32-bit string r_b, then he plays the OT-sender, using r and $r \oplus b$ as his two strings, ordered according to x_b. Namely if $x_b = 0$ then Boyd uses the pair $(r_b, r_b \oplus b)$, and otherwise he uses $(r_b \oplus b, r_b)$. Boyd's output share is r_b, and Debra's share is the received string (which we denote r_d). It is easy to check that $r_b \oplus r_d = \{0 \text{ if } x_b \oplus x_d = 0, \text{ or } b \text{ if } x_b \oplus x_d = 1\}$.

   The second instance is symmetric, resulting in the two bidders having output strings r'_b, r'_d satisfying the condition $r_b \oplus r_d = \{d \text{ if } x_b \oplus x_d = 0, \text{ or } 0 \text{ if } x_b \oplus x_d = 1\}$.

   The two bidders XOR their shares from the two instances, thus obtaining $y_b = r_b \oplus r'_b$ and $y_d = r_d \oplus r'_d$, and indeed $y_b \oplus y_d = \max(b,d)$.
3) Next, Boyd sends its shares x_b and y_b to Sally over a private channel. Then Sally and Debra engage in another Yao protocol, computing whether the reserve price was met, i.e., the indicator bit for $(y_b \oplus y_d) \geq s$. If the reserve was met then Debra sends x_d, y_d to Sally and Boyd, who can recover the winning bid $y_b \oplus y_d$ and the winner $x_b \oplus x_d$ (and then they send them back to Debra).

PERFORMANCE. We ran our demo on a Lenovo Carbon X1 machine (4th generation), with Intel Core i5-6300U CPU and 8GB or RAM, running Ubuntu 16.04, where the peers and servers were all running on separate docker container on the same machine. The time of the execution (and endorsement) of a transaction proposal involving the secure-MPC protocol was about 0.3s, which is faster than what it took to commit a single block to the ledger. We speculate than 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 even much faster, in particular by improving the communication channels.

IV. CONCLUSIONS

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, and are currently working on integrating these components into Fabric.

REFERENCES

[1] F. Benhamouda, S. Halevi, and T. Halevi. Supporting private data on Hyperledger Fabric with secure multiparty computation. https://shaih.github.io/pubs/bhh18.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 B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 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. *Journal of the ACM*, 38(3):691–729, 1991.

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

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

[6] 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.

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

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

[9] 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-ceo-ginni-rometty-blockchain-transactions-internet-communications-2017-6, 2017.

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

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

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

[13] 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.

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