# **Supporting private data in Hyperledger Fabric**

**Authors**: Fabrice Benhamouda, Shai Halevi, Tzipora Halevi

<https://jira.hyperledger.org/browse/FAB-5131>

[**Supporting private data in Hyperledger**](#_vblyhi80y8d2)[**F**](#_vblyhi80y8d2)**a**[**b**](#_vblyhi80y8d2)**r**[**i**](#_vblyhi80y8d2)**c 1**

[The proposed changes](#_7l7f3jic2r9) 2

[Local configuration](#_mif2rvuzx1qn) 2

[Proposed interface](#_gvggunms0rjy) 3

[Inter-peer communication](#_g7fj8vw1pad9) 3

[Proposed interface](#_tnsbd1xunqrx) 3

[Remarks](#_c96nhk4kjlfo) 4

[Comparison with other solutions](#_ubh5604nixcx) 4

[Blockstream Confidential Assets](#_i0tovnr1oxjr) 4

[Side DB - Private Channel Data (FAB-1151)](#_tc0abrjb50kf) 5

[Appendix](#_8u4cjydzfvle) 5

[Background - cryptographic secure computation](#_7ks1es7t6gu3) 5

[Background - zero-knowledge proofs and verifiable computation](#_r87p8hqyb47b) 6

[Our demo](#_hyrwm7r0pczp) 6

[Scenario](#_letw6mmzhbyn) 6

[Current implementation of the demo](#_nlh2q7sirezg) 7

The object of this proposal is to support in Hyperledger Fabric private data that is visible to (ie can be decrypted in) some peers but not others, and moreover support transactions that are based on such private data. We propose below a **very modest change** to the current architecture, that opens the door to a wide range of advanced applications.

In the current Hyperledger Fabric architecture, all the endorsers of a transaction must have an identical view, hence each variable is either visible to all of them or to none. Supporting transactions based on private data would open a whole new level of applications, some examples include:

* *Marketplace*: Transaction should only go through if buyer has enough money, but buyer balance is confidential and not shared with other peers. This would allow online sell of items, where the seller would not need to verify the buyer's credentials but the system would do that for him.
* *Medical*: Drug should be dispensed only if the patient's condition warrants it, but only patient's agent has access to confidential medical data.
* *IoT*: Aggregate statistics should be publicly recorded on the chain, but detailed data is private to the organization that supplies it. Examples of this may include vehicular data and electricity usage data.
* *Audit*: When departments align their books, checking that their data is consistent without them having to share confidential data (e.g., if there is a "Chinese wall" between them, i.e., an information barrier that was created to avoid conflict of interests between the different departments).
* *Silent auctions*: Holding an auction while keeping secret the bid prices, except for the winning bid. The seller could also have a secret reserve price.

The current Hyperledger Fabric architecture has some limited support for private data, but it is not nearly enough for applications such as above. For example, private channels can only be used for a complete partition of the data, where access to the data on a given channel is all-or-nothing. Moreover, hiding data in a private channel means that it cannot be a part of any transaction on any other channel.

Beyond the strict partition of channels, private data in the current architecture can only be supported by relying on client-side encryption: A client can include an encrypted field as an opaque blob in its transaction, but such fields will not be visible to the peers and so cannot be used for endorsing transactions. Also, we stress that shifting application logic to the client means that these client-side parts cannot enjoy the same immutability and consistency guarantees that are provided by on-chain processing.

In contrast, the solution that we propose here allows private data to be used in arbitrary ways as part of the endorsing transaction logic, while enjoying all the security guarantees that the blockchain provides.

## **The proposed changes**

Our proposal maintains the fundamental property of a globally consistent ledger, where all peers see (a prefix of) the same ledger. To support private data, some of the information on the ledger is kept in an encrypted form, under keys that are only available to those peers that are allowed to see it. To support transactions that depend on such private data, we require two changes in the architecture:

### **Local configuration**

The chaincode implementing the endorsement logic at the different peers should have access to local parameters that are only available to that peer and not others. This can be used to give the chaincode at peer X the decryption keys owned by X, while the same chaincode at peer Y will have no access to those keys. Decryption keys given to peers are independent of any cryptographic material used by Hyperledger Fabric. Since the decryption keys will be used to protect each peer secret and will only be given to its copy of the chaincode, the overall security guarantees of the Fabric will not be impacted negatively by it. However, the privacy options of the peer will be extended, as it allows this peer to now keep private data.

Access to local configuration has many uses, beyond just our proposal of supporting private data. For example it can be used to store tokens for access to an external resource or API by that peer, or to specify proxy configuration.

Adding a local configuration has been previously requested in [FAB-2585](https://jira.hyperledger.org/browse/FAB-2585), for use in multi-channel Fabric setup where members have sub-ledgers. In this case, the sub-ledgers' names will be stored in the local configuration.

#### **Proposed interface**

In interface **ChaincodeStubInterface** from [github.com/hyperledger/fabric/core/chaincode/shim/interfaces.go](https://github.com/hyperledger/fabric/blob/master/core/chaincode/shim/interfaces.go), add the following function:

// GetLocalConf returns the byte array value specified by  
 // the `key` from the peer local configuration for the chaincode  
 GetLocalConf(key string) ([]byte, error)  
  
 // GetPeerID returns the peer ID executing the chaincode  
 GetPeerID() string

The second function GetPeerID allows a chaincode to have a different behavior depending on the peer on which it is executed. We can use the same ID string as in the gossip protocol, i.e., a hash of <Peer MSPID || peer certificates bytes>. We remark that we could use the local configuration to do that too, but it seems cleaner to also have GetPeerID.

Usage (assuming stub shim.ChaincodeStubInterface is correctly defined):

theSecretKey, err := stub.GetLocalConf("myKeyNumber7")

#### **Implementation details**

The **ChaincodeStubInterface** is implemented by the **ChaincodeStub** struct defined in [github.com/hyperledger/fabric/core/chaincode/shim/chaincode.go](https://github.com/hyperledger/fabric/blob/master/core/chaincode/shim/chaincode.go). When the peer needs to communicate with the chaincode, it instantiates a **ChaincodeStub** and its init function is invoked. The **ChaincodeStub** needs to be enhanced with the following field:

* localConf map[string][]byte

Then, *GetLocalConf* will return the entry in the localConf field corresponding to the passed *key*. *GetPeerID* just looks up in the localConf for the special key *peerID* and cast the result to a string.

To complete the picture, we need to have the localConf loaded from somewhere. Part of the configuration is chaincode-dependant, some other part is peer-dependant (i.e. peerID).

For the chaincode-dependant local configuration, the chaincode installation process needs to be enhanced to allow the peer’s administrator to pass (or include in the chaincode package) the chaincode’s local configuration. This a more involved change, it requires the modification of the lifecycle system chaincode (lscc.go). The entry point is the function *executeInstall*. Modification to the **CCPackage** struct are probably needed as well.

For the peer-dependant local configuration, this can be set in the core.yaml configuration file, loaded at peer bootstrap.

When a chaincode is invoked, *localConf* needs to be passed. One way to do intervene and modify the *endorser.go#callChaincode* function. One could add a new field to the CCContext to carry the local configuration and load it at *callChaincode* invocation.

### **Inter-peer communication**

To be able to handle data that is only visible to some peers but not others, the endorsement process must allow communication between peers. 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. (We stress that in our solution the endorsers never get to see others' secrets, instead we use cryptographic techniques for secure computation to "compute on hidden data." Some background information about these techniques is included in an [appendix](http://localhost:6419/#background---cryptographic-secure-computation).)

The run-time for the communication between the peers is estimated to be very short, and is not expected to impact significantly the run-time of the overall endorsement.

Actually, in our demonstration running with 3 peers on the same machine (and a single orderer), the endorsement with multiparty computation using inter-peer communication is faster than the ordering of the resulting endorsed transaction (i.e., the time to get confirmation of the commitment of the transaction - about 2s).

#### **Proposed interface**

In interface **ChaincodeStubInterface** from [github.com/hyperledger/fabric/core/chaincode/shim/interfaces.go](https://github.com/hyperledger/fabric/blob/master/core/chaincode/shim/interfaces.go), add the following functions:

// MakeLink returns a new link to `peerID`   
 // Blocks until the link is ready  
 MakeLink(peerID string) (Conn, error)

where Conn is the connection interface, that needs to support read and write capabilities. For example, Conn may be the connection interface from the package net, or any other similar interface.

Usage (a simple example assuming only two peers peer1 and peer2, and assuming that stub shim.ChaincodeStubInterface is defined):

if stub.GetPeerID() == "peer1" {  
 // ....  
 conn, err := stub.MakeLink("peer2", "my-sid")  
 n, err := conn.Write(message)  
 // ....  
 } else {  
 // ....  
 conn, err := stub.MakeLink("peer1", "my-sid")  
 n, err := conn.Read(message)  
 // ....  
 }

Note that the values “peer1” and “peer2” (in the example above, line 1, line 3, and line 8) do not need to be hardcoded in the chaincode. They can also be arguments of the invoke function or can be deduced from them. More complicated scenarios can be handled, for example, using roles (e.g., a seller is communicating with one potential buyer, etc.). The above example is just meant a toy example to show the use of MakeLink and GetPeerID, and should not be seen as a typical use case.

#### **Remarks**

* If two invoke queries q1 and q2 are performed in a short amount of time on two peers peer1 and peer2, it is possible that these two peers do not see them in the same order. Some mechanism needs therefore to be implemented to ensure that there is no deadlock: e.g., peer1 executing q2 waits for a link to peer2 for q2, while peer2 executing q1 waits for a link to peer1 for q1.
* Each link must be associated with an ID, uniquely identifying the invoke query that had triggered the establishment of that link. For example, this ID may be a hash of the session ID and of the transaction ID.

## **Comparison with other solutions**

### **Blockstream Confidential Assets**

[Blockstream Confidential Assets](https://blockstream.com/2017/04/03/blockstream-releases-elements-confidential-assets.html) enables exchange of confidential assets between participants. It uses client-side "encryption" (or commitments) as described above to hide the confidential information, and the client also adds a cryptographic zero-knowledge proof to its transaction (cf. [appendix](http://localhost:6419/#background---zero-knowledge-proofs-and-verifiable-computation)), making it possible for everyone to verify that the confidential information was processed correctly without revealing its value. So, for example, a client that wants to subtract 10 from one account and add 10 to another account, can prove that it did so without revealing the balance in these two accounts.

The are two major differences between this solution and what we propose here:

* Blockstream is a client-side solution: It is up to the client who knows the secret information to apply all the relevant transformations and generate the cryptographic proofs, and then the "peers" can check the proofs without knowing any secret information.
* In contrast, our proposal seeks to allow the peers themselves to deal with secret information. This would let us use all the blockchain services (such as membership, authorization, and perhaps even broadcast) while processing the data.
* The client must have access to *all* secrets: To prepare the cryptographic proofs, the client who submits the transaction must have access to all the relevant information, including any secret values. This means, for example, that the transaction cannot compare a secret value that only client A knows to another secret data that only client B knows, since neither of them knows enough to determine the result of that comparison.

### **Side DB - Private Channel Data (**[**FAB-1151**](https://jira.hyperledger.org/browse/FAB-1151)**)**

Side DB is a proposed extension to the basic Hyperledger architecture, where peers can maintain private data "on the side" (i.e., not on the ledger), keeping only some hash of this data on the ledger. This extended architecture requires a new set of tools to maintain the separate corpus of "side data". In contrast, our proposal does not require additional DBs or other tools, storing the private data directly on the chain in encrypted form.

Our multiparty secure computation (MPC) techniques could also work with SideDB when it becomes available. In this case, instead of encrypting the data, we would use SideDB to store it privately. This has the advantage of not keeping the data, even encrypted, on the ledger (in case future computing advances could break the encryption).

Another major difference is that Side DB still assumes that there are some peers that can see *all the private data* for a given transaction, so these peers can locally make the right endorsement decision and compute the correct Write-Set (and communicate it to the other peers via gossip as needed). Our proposal is more flexible, in that it allows a transaction to depend on different pieces of private data that are visible to different peers (so for example we can compare X that only peer#1 can see to Y that only peer#2 can see).

## **Appendix**

### **Background - cryptographic secure computation**

Cryptographic techniques going back to the 80's [5,3] 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.

A simple illustration of such a protocol (though not the most efficient way of doing it) uses homomorphic encryption, which allows computation on encrypted data: The participants can all encrypt their secret data, then they can compute the desired function on the encrypted inputs, and finally jointly decrypt the result, recovering only the outcome and not the individual inputs.

Over the last decade we saw many advances in practical protocols for cryptographic secure computation, and this technology is now efficient enough to handle many real-life problems, such as distributed voting, electronic auctions, and sharing cryptographic signature or decryption functions, and more.

### **Background - zero-knowledge proofs and verifiable computation**

A "Zero-knowledge proof" [4] allows a prover to convince a verifier of the veracity of some statement, without revealing to the verifier anything beyond the fact that the statement being proven is true. In our context, we can use such proofs to verify that processing of a transaction was done correctly, namely that the values encrypted in ciphertexts in the "write-set" were obtained from the values encrypted in the ciphertexts of the "read set" via the correct prescribed procedure. Since we want these proofs to be publicly verifiable, we need to use the non-interactive variants of such proofs [2,1].

Verifiable computation received a lot of interest over the last few years, and many techniques are known to prove the correctness of complex computations in an efficient manner.

### **Our demo**

#### **Scenario**

As an example for an application that requires dealing with private data, consider holding eBay-like auctions without a trusted auctioneer. In this scenario, we want to hide from the seller everything but the winning bid. There are real-life scenarios where this is needed, e.g., where the parties are making repeated bids, and the bidders with the losing bids do not want to expose their bid to the seller or anyone else. (For example, suppose that the seller is dishonest, and only lists an item to see the bids of all the potential buyers - if the losing bids are not exposed, then the seller does not learn anything about the other bidders, just the winning bidder.)

We created a demo that runs on the alpha version of Hyperledger Fabric 1.0, demonstrating how this example application would work.

In this demo there are sellers and buyers, each with its own peer in the system. The actions that participants can perform are: (1) creating an item as a seller, (2) bidding on an existing item as a potential buyer, and (3) holding an auction for a given item.

Each item offering comes with some public fields such as the item description, future auction date, and also auction starting price. But also for any new item there is a secret reserve price, which is known only to the seller.

Similarly, each new bid has some public fields such as the item in question and the identity of the bidder, but also a secret value which is the bid price, that only this bidder knows.

The secret fields will appear on the ledger, but they will be encrypted and only the relevant peer will have the keys needed to decrypt them. When the seller holds an auction, it has to communicate with all the bidders in order to decide if the auction is successful or not. During this protocol the parties will communicate with each other, and at the end of the protocol they will learn whether any of the bids was higher than the reserve price, and if so what is the winning bid and who made it. Other than that, the peers will learn nothing. For example if none of the bids was high enough, then all they learn is that the bids were too low, but not (for example) what they were.

#### **Current implementation of the demo**

As Hyperledger Fabric 1.0 does not support the two extensions we mentioned above (local configuration and inter-peer communication), we implemented them in a side server in Go (called the *helper server*). The chaincode communicates with this helper server (via gRPC) to access its local configuration and to create communication links with other peers. In our current demo, there is a unique helper server for all peers and it must therefore be trusted. This is just an artifact used to simplify development (and, as there is currently no support for local configuration, this artifact is somehow necessary if we want to run all peers on the same machine, in different docker containers, which is the setting of our a demo). Of course, in any real use case we cannot have such a trusted helper, instead the fabric itself must implement the features that the helper server currently provides (i.e., the two extensions that we describe in this proposal).

The multiparty computation part of the chaincode is developed in C++ using the [emp-toolkit](https://github.com/emp-toolkit) library. It is included in the chaincode and is linked to the Go part of the chaincode via [SWIG](http://www.swig.org). As the emp-toolkit library only does a part of what we needed — it only implements secure multi-party computation between two peers — we extended it to handle the case of three peers. This would need to be further extended to handle a larger number of peers.

We modified fabric-sdk-node to include SWIG and C++ files in the chaincode package or tarball sent for installation and instantiation. See<https://github.com/fabrice102/fabric-sdk-node/commit/c1bca292a26fa03ac6295c8e6c291119ab82fef1>.

We also used a customized fabric-ccenv Docker image to include the emp-toolkit library and SWIG. (Independently of this proposal, we found that adding a simple line to the Dockerfile of the fabric-ccenv Docker image significantly reduce compilation time of the chaincode - see<https://jira.hyperledger.org/browse/FAB-4947>.)

To summarize, the current demo implementation was designed using an additional external helper server to provide the two simple (but critical) extensions of local configuration and communication links. But to go beyond a demo, these features must be integrated into the Hyperledger architecture itself, so that they are available to the designers of smart contracts. The secure computation protocols are implemented in the chaincode, using libraries that will be developed for this purpose.

[1] Manuel Blum, Paul Feldman, and Silvio Micali. Non-Interactive Zero-Knowledge and Its Applications. STOC 1988, pages 103–112.

[2] Amos Fiat and Adi Shamir. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. CRYPTO 1986, pages 186-194.

[3] Oded Goldreich, Silvio Micali, and Avi Wigderson, Proofs That Yield Nothing But Their Validity Or All Languages in {NP} Have Zero-Knowledge Proof Systems, JACM 38(3), 1991, pages 691-729.

[4] Shafi Goldwasser, Silvio Micali, and Charles Rackoff, The Knowledge Complexity of Interactive Proof Systems, SIAM J. Comp. 18(1), 1989, pages 186-208.

[5] Protocols for secure computations, Andrew Chi-Chih Yao, FOCS 1982, pages 160-164.