# Design for Fabric SGX Chaincode Execution

*Marcus Brandenburger, Christian Cachin, Alessandro Sorniotti*

This design assumes familiarity with Fabric and SGX.

https://arxiv.org/abs/1801.10228

https://hyperledger-fabric.readthedocs.io/

## Problem Description

This section describes the problem of secure smart-contract execution using trusted hardware. We explore intricacies caused by roll-back attacks, illustrate a strawman approach that is infeasible, and introduce our approach to support secure chaincode execution using Intel SGX. This executes each chaincode in its own enclave during endorsement at a peer and thereby protects the confidentiality and integrity of the blockchain application.

### System model

We consider a Hyperledger Fabric blockchain network with clients, an ordering service, and a set of peers, which collaboratively execute transactions and maintain a distributed ledger. Although Fabric supports multiple channels, which correspond to multiple blockchains within the same network, we focus on only one channel here.

A client invokes transactions by sending a chaincode operation to some peer, which then executes (simulates) it and produces an endorsement containing the resulting state change on the ledger. The operation, the response, as well as the ledger may contain sensitive information that should stay secret.

To prevent such information leakage, every peer is equipped with an SGX-enabled CPU and executes transactions inside an enclave. The chaincode is stateless, and a transaction only takes the operation and the blockchain state in the KVS as inputs, accessed with \*getState\*. The chaincode must perform updates to the ledger only through \*putState\* operations. The execution of a chaincode operation returns a response that may include a computation result, the state update, and the read-write dependencies.

A Fabric blockchain is initialized through a so-called genesis block, which is created collaboratively by all network participants. The ordering service is responsible for collecting the transactions into blocks, establishing a total order among transactions, and delivering the blocks to all peers. Each block is digitally signed by the ordering service. All clients and peers can validate this signature and recognize the block's validity. We assume the ordering service as a whole is correct, and all clients and peers trust it.

### Threats

Although most peers are usually correct, a peer may become \*malicious\* and behave incorrectly, for instance, when it tries to maximize its own profit or becomes corrupted by an attacker. A peer has full control over the operating system, applications, and the data residing in memory and persistent storage (i.e., the blockchain state). A malicious peer, however, cannot access or tamper with the code and data residing in an enclave. A malicious peer may neither break cryptographic primitives nor extract any secret information that from an enclave. Consequently, a chaincode running in an enclave always produces the correct results, that is, the chaincode does not deviate from its specification, the enclave-internal state is only known to the enclave itself, and nothing is revealed apart from the resulting state change.

However, a malicious peer can invoke the chaincode enclave with any input and in arbitrary order. The peer may intercept, modify, reorder, discard, or replay chaincode operations, and when the chaincode enclave accesses the KVS, the peer may feed any blockchain state to it.

Furthermore, the peer might even drop messages or completely halt an enclave, but we do not consider such denial-of-service attacks in this work. We also ignore potential information leakage from SGX on side channels because this appears orthogonal.

As is well-known from the research papers on secure computation with cryptographic protocols, integrity and confidentiality cannot be considered separately. Likewise, for a secure application running in an enclave, a malicious host may break confidentiality by triggering the enclave to execute on ``incorrect'' inputs. In the blockchain context, this means that the chaincode execution deviates from the agreed-on order on the transactions established through consensus.

Such an attack could reveal secret information as follows. Recall the blockchain auction mentioned earlier, where the auction evaluation executes inside an enclave that also protects the private decryption key for the bids. During the auction evaluation, the enclave reads the submitted bids from the blockchain, decrypts them, determines the winner, and outputs it in the clear. Even if the malicious peer cannot decrypt bids, it might let the enclave execute the evaluation transaction multiple times and reset it again afterwards, for instance, every time when a new bid has been stored on the blockchain. Thereby the peer could learn information about other bids. This illustrates the complex interplay between integrity and confidentiality.

### Strawman approach

The roll-back problem just described can be avoided by letting the enclave only execute operations that have been ``ordered'' by the network. This amounts to running the entire blockchain peer inside an enclave, as also suggested in earlier work (e.g., Microsoft COCO white paper). We call this the \*strawman approach\* that might work for an order-execute architecture, but argue later why better designs exist.

For Fabric, the strawman design would mean to encapsulate the chaincode execution, endorser, committer, ledger-state access, and all other parts of a peer inside an enclave. This obviously protects the integrity of the input sequence for the chaincode, since the entire Fabric peer runs within SGX. A similar approach may allow to deploy the Fabric peer as a \*secure service container\* on an IBM Z system, which includes the whole operating system, middleware stack, and blockchain platform.

Although no operating system is running within SGX, recent research has demonstrated how legacy applications can run in SGX through a library OS that executes unmodified applications in an enclave. Note that the library OS adds tens of thousands of lines of code that also run along the application in the enclave.

This approach introduces multiple problems, however. First, it stands in contrast to the important computer-security principle of minimizing the size of the trusted computing base (TCB). Specifically, also the SGX developer guidelines recommend to partition an application into a trusted and an untrusted component; only a small portion of the application code should execute inside the enclave. A smaller TCB has fewer errors, reduces the attack surface, and is more amenable to detailed security analysis than the entire application.

A second problem stems from the limited memory available to enclaves. An enclave's memory resides in the enclave page cache (EPC) isolated from the rest of the system. The EPC is currently limited to 128 MB. Once an enclave reaches that limit or a context switch occurs, pages are moved to DRAM. This results in a dramatic loss of performance, as reported in several research papers. In particular, since the ledger grows with every block, holding the whole blockchain state in the enclave quickly reaches the memory limitation.

## Solution overview

To avoid the drawbacks of the strawman approach, we propose a modular architecture that separates the chaincode execution conceptually from the peer and only execute the chaincode inside an enclave. The protocol-specific aspects of the peer are encapsulated in an abstract ordering service, of which one process might run on the same peer.

The ordering service produces a signed sequence of transactions for execution within the enclave. The enclave can verify that transactions originate from the ordering service, are in the proper order, and have not been tampered with. The enclave also keeps information about the transaction history, which allows to detect transaction-ordering violations or replayed transactions. The malicious host might still reset the enclave to an earlier point in the execution sequence, but this would not harm the application since the transactions are deterministic and execution would simply produce the same outputs again.

As described so far, this approach works fine with an order-execute architecture for state-machine replication. Fabric, however, uses the execute-order-validate paradigm, where a peer executes a transaction before consensus on the order is reached. Consequently the execution is speculative and can be repeated without affecting the blockchain state, as transactions are simulated during endorsement and only take effect after the ordering. This means a malicious host could infer information about the secret application data from the speculative execution. Not even a trusted ordering service can prevent this type of leakage.

To resolve this issue, we will adapt the applications to respect the speculative nature of execution in Fabric. For the auction example, in particular, a \*barrier\* will be stored on the blockchain such that the chaincode enclave only evaluates the auction if the barrier is present. The barrier is set by invoking the chaincode with an operation to ``close'' the auction but not yet evaluate it. If the barrier is present on the ledger, a malicious peer may no longer submit new bids to the auction. On the other hand, the auction evaluation will only consider bids added to the ledger before the barrier. Note that this barrier plays a role similar to a memory barrier in a multi-core computer system with concurrent threads.

Following the execute-order-validate architecture, the chaincode enclave must execute transactions only on the committed blockchain state, that is, with ledger entries that result from ordered transactions and that have been committed by all peers. Otherwise a malicious peer may produce the barrier itself and feed the resulting state into the enclave when evaluating the auction. The system described next ensures this.

To formally model the information leakage permitted in the execute-order-validate architecture of Fabric, we model a blockchain as stateful functionality F: C x T -> S. At any time the state of the chaincode is an element of C. The clients invoke transactions in T, which may contain operations with arguments according to F, but these are subsumed into the different t \in T. Given s \in S, applying a transaction t \in T of F means to compute s' := F(s,t), resulting in a subsequent state s' \in S. Using a trusted ordering service as introduced earlier, the blockchain's state evolution is defined through the sequence of transactions signed by ordering.

With the chaincode functionality F running in an SGX enclave, even a malicious peer may only learn the subsequent state resulting from a transaction, but nothing about the computation itself. Since cryptographic keys could reside in the enclave, the ledger state doesn't necessarily reveal all relevant information. Due to the rollback attacks introduced earlier, however, such a peer can execute any transaction \*on any input state\* that is in the history of transactions issued by the ordering service.

The desired security is therefore summarized as follows. Consider a blockchain system with an execute-order-validate architecture and suppose the correct ordering service produces a sequence of states [s\_0, s\_1, ... ,s\_m], where s\_j = F(s\_{j-1}, t\_j) for t\_j \in T and j \in [1,m]. We say that the chaincode is \*secure up to resets\* if any malicious peer, through interacting with the chaincode running inside the enclave, may obtain states s^\*\_{k+1} = F(s\_{k},t^\*), for any k \in {0, 1, ..., m} and an arbitrary transaction t^\* \in T, but no further information.