On Private Data Collection of Hyperledger Fabric

Shan Wang*‡, Ming Yang*, Yue Zhang†, Yan Luo‡, Tingjian Ge‡, Xinwen Fu‡, Wei Zhao§
* Southeast University. Email: {wangshan, yangming2002}@seu.edu.cn
†Jinan University. Email: zyueinfosec@gmail.com
‡University of Massachusetts Lowell. Email:{Yan_Luo, Xinwen_Fu}@uml.edu, ge@cs.uml.edu
§Shenzhen Institute of Advanced Technology. Email: weizhao8686weizhao@163.com

***Abstract*—Hyperledger Fabric is a popular permissioned Blockchain framework for a consortium of organizations to develop Blockchain based applications and transact within the consortium. Hyperledger Fabric introduces a fine-grained access control mechanism called the private data collection (PDC), which allows private data to be shared by only a subset of participants. In this paper, we analyze PDC and show three classes of use cases in which misuse of Hyperledger Fabric features may endanger implemented Hyperledger Fabric systems. We present two groups of potential attacks including fake PDC results injection and PDC leakage against the misuse of the policy based consensus protocol. We use prototype systems to validate the discovered attacks. We also collected 6392 Hyprledger Fabric projects on GitHub and built a tool to statically analyse them. We find that 86.51% of the PDC related projects are potentially vulnerable to the fake PDC results injection attacks, and 91.67% have PDC leakage issues. We design new features for the Hyperledger Fabric framework to mitigate the attacks and show that the new features have minor impact on the system performance.**

***Index Terms*—Blockchain, Hyperledger Fabric, Private Data Collection**

I. Introduction

Permissionless Blockchain systems such as Bitcoin [1] and Ethereum [2] are open to the public. A party does not need a permission to join the system and can see all the data recorded on a blockchain. This prevents the Blockchain technology from being applied in scenarios where particular organizations come together as a consortium to transact with each other and do not desire to expose their data to the public. The emerging *permissioned Blockchain systems* [3] often involve multiple access control mechanisms to protect data privacy and confidentiality, and allow only authorized participants to join the permissioned system and transact with each other privately.

Hyperledger Fabric is a very popular permissioned Blockchain system, designed for enterprise applications where a consortium of organizations develop Blockchain based applications and transact privately within the consortium. For simplicity, we will denote Hyperledger Fabric as Fabric in the rest of this paper. According to Forbes' *Blockchain 50* [4], half of the top 50 enterprises that embrace the Blockchain technology choose to use the Fabric system. Many representative cloud platforms offer Blockchain as a Service (BaaS) with Fabric, including IBM, Oracle, Microsoft Azure, Amazon, Google Cloud, Tencent and Alibaba Cloud.

In order to achieve high scalability, performance and modularity, the Fabric decouples the tasks of a node in a traditional Blockchain system such as Ethereum into three types of nodes, i.e. peers, orderers and clients. A three-phase "execute-order-validate" transaction workflow is adopted. A transaction is proposed by a client, then endorsed by peers, packaged into a block by orderers, and finally verified and stored into the ledger by peers. A valid transaction needs to collect enough endorsements from peers specified by the endorsement policy in order to pass the validation at all peers. For access control and data isolation, the Fabric adopts permission related mechanisms including identity, policy, channel and private data collection (PDC). Organizations with the same business goals can be grouped into one channel. The organizations in one channel maintain the same ledger while different channels maintain different ledgers.

The private data collection (PDC) is proposed for those organizations that need to keep private data from others in one channel. The PDC hash is stored at all peers in a channel. The original PDC is only maintained by a subset of peers of authorized organizations in a channel. For brevity and clarity, we denote the organizations/peers that own the PDC as "member organizations/peers", and others as "non-member organizations/peers", and call non-private data as public data in the rest of this paper.

We perform a comprehensive analysis of the PDC of the Fabric. Our major contributions are summarized as follows: We present three classes of use cases in which misuse of the Fabric's policy based consensus protocol endangers the PDC of a Fabric system: (i) PDC Non-member peers endorse PDC transactions; (ii) PDC transactions are validated through the same endorsement policy as public data transactions; and (iii) The "Payload" field is used to return information in transaction proposal response. In the Fabric, a valid transaction shall pass two checks: endorsement policy and version conflict. We denote such a consensus protocol as proof-of-policy (PoP).

We are the first to discover the fake PDC results injection attack and PDC leakage issue exploiting the misuse of PoP of the PDC. Under the fake PDC results injection attack, malicious peers or clients may disrupt the integrity of the ledger. They may inject a valid transaction with a fake value into the blockchain or write fake values into the PDC in the ledger's world state. With the PDC leakage issues, the PDC can be revealed to PDC non-member peers.

We use prototype systems to validate the discovered attacks. We also build a tool to scan all the 6392 Hyperledger Fabric projects available on Github, and find that 86.51% of the PDC related projects are potentially vulnerable to the fake PDC

results injection attacks, and 91.67% have PDC leakage issues.

We design new features for the Fabric to mitigate the discovered attacks, and implement these new features by modifying the source code of the Fabric. The implemented new features have minor or negligible impact on the system performance.

Responsible Disclosure: The authors have communicated and reported the findings of this paper to the Hyperledger Fabric team since August 2020.

II. Background

In this section, we introduce the core components of Hyperledger Fabric and how they work together to process transactions.

A. Hyperledger Fabric Overview

Hyperledger Fabric [3] is a permissioned blockchain system composed of three types of nodes, namely peers, orderers and clients. Hyperledger Fabric adopts multiple mechanisms to implement access control.

*1) **Peers**:* Peers maintain ledgers and smart contracts, and endorse and validate transactions. When peers endorse transactions, they are also called endorsers. When peers update the ledger, they are called committers.

Ledger. The ledger hosted at peers has two components, i.e. the world state and Blockchain. The world state is a database and stores current data. The data is stored in the form of $\langle$key, value, version$\rangle$. *version* is used for the transaction concurrency control, and monotonically increases every time the corresponding key is updated. The blockchain stores all the transactions that produce the current world state.

Chaincode. Smart contract is also called *chaincode* in Hyperledger Fabric. Chaincode is a program that defines the business logic operating on the world state. A transaction is created by a client invoking a function in the chaincode and contains the results of the function, which may perform `read`, `write` or `delete` operations on the world state. The chaincode should be deployed on all involved peers sharing the same business goals. These peers agree on the configuration of the chaincode including the endorsement policy. A valid transaction should be signed by the endorsers specified in the endorsement policy.

*2) **Orderers**:* Orderers are a group of special nodes responsible for block generation using the consensus algorithm Raft [5] [6]. They blindly bundle transactions into new blocks without validating the content of the transactions.

*3) **Clients**:* A client runs a Hyperledger Fabric application to interact with peers and invoke chaincode functions so as to access and operate on the ledger. The operations take effect only after the transaction is validated as valid by all peers.

*4) **Access Control Mechanisms**:* Hyperledger Fabric assigns each node an identity, and uses a set of policies to control who can access specific resources or make changes to the system. Hyperledger Fabric also introduces *channel* and *Private Data Collection (PDC)* mechanisms for communication and data confidentiality.

Identity. Each participant needs an identity to prove who it is so as to transact in the permissioned blockchain system. Particularly, Hyperledger Fabric issues each participant a certificate to indicate its identity, and each identity is bonded with one organization within the consortium of organizations participating in the Hyperledger Fabric system.

Policy. A policy is a logical expression of identities evaluating if a specific task is performed by required identities. Almost everything is controlled by policies including not only resource access but also how members reach an agreement. For example, the endorsement policy stipulates whose endorsements a transaction must collect so that it can be validated as valid. Policies can be categorized into signature policies and implicitMeta policies. A signature policy is a logical expression using logical operators including *AND*, *OR* and *NOutOf*. For example, the signature policy "*AND(Org1.peer, Org2.peer)*" means a peer from org1 and a peer from org2 must sign the data of interest so that the corresponding transaction is to be considered valid when this signature policy is used as the endorsement policy. An implicitMeta policy is an expression using logical operators *ALL*, *MAJORITY* or *ANY* over participating organizations' signature policies. For example, the implicitMeta policy "*MAJORITY Endorsement*" means that most of the signature policies called "Endorsement" defined by participating organizations shall return true.

Channel. Channel is a private communication mechanism which groups organizations into multiple groups for common goals. Only the clients, peers and orderers of the organizations in the same group can join in the same channel and communicate with each other. Each channel maintains a separate ledger. Peers in the same channel share the same ledger. Any outsider can not participate in activities or access the ledger in the channel. This provides data and communication isolation for privacy and confidentiality.

Private Data Collection (PDC). The PDC is introduced when a subset of organizations needs to keep sensitive data private from other organizations in the same channel.

We show an example in Fig. 1 illustrating the architecture of a Hyperledger Fabric system. It has four organizations org1, org2, org3 and org4. Client $A1$ and Peer $P1$ are contributed by org1. $A2$ and $P2$ are contributed by org2. $A3$ and $P3$ are contributed by org3. $O4$ and $P4$ are contributed by org4. Peers $P1$, $P2$ and $P4$ join the same channel $C1$ and host the same chaincode $S1$ and ledger $L1$. Only clients $A1$ and $A2$ can access $L1$ via $S1$. $P2$ also joins the channel $C2$, and hosts $S2$ and $L2$, which $A2$ can access too. $P1$ and $P4$ maintain a PDC. Only $P1$ and $P4$ have the original private data, and other peers in the same channel like $P2$ only have the hash of private data.

B. Three-phase Transaction Workflow

The lifecycle of a transaction in Hyperledger Fabric has three phases, i.e. execution, ordering and validation. Fig. 2 describes the whole process. Note that we introduce the public data transaction workflow in this section while the PDC transaction workflow is introduced in Section III-A2.

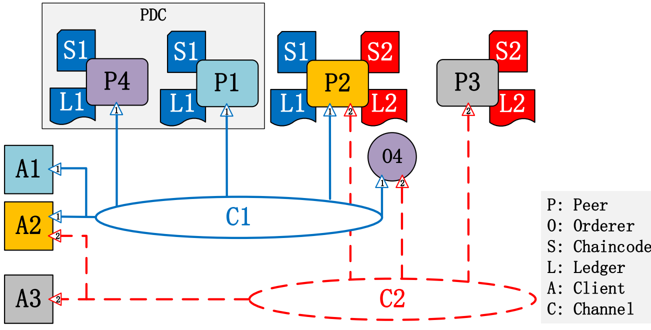

Fig. 1. Example of Hyperledger Fabric system

*1) **Execution Phase**:* A client sends a transaction proposal to the peers which act as endorsers required by the endorsement policy, such as peer1 and peer2 in Fig. 2. The transaction proposal includes the client identity, target chaincode ID, target function name and its parameters. These endorsers simulate the target function execution. Each endorser generates the *proposal response* and *endorsement* with a signature of the proposal response and return the two parts to the client. The chaincode execution result is stored in the two fields of "read/write set" and "response" of *proposal response*. In the execution phase, the ledger is not updated with the execution result. After the client collects all proposal responses with endorsements returned from required endorsers, it checks if all the returned results from different endorsers are the same. If yes, the client will assemble a transaction and submit it to the orderers.

*2) **Ordering Phase**:* The orderers may receive transactions from multiple clients. When orderers collect a pre-defined number of transactions or after a pre-defined time, they group these transactions into a new block. Then orderers distribute the new block to all peers including endorsers and other peers.

*3) **Validation Phase**:* After peers receive a new block, each peer independently validates transactions in the block through two checks of the proof-of-policy (PoP) consensus protocol: (i) *Endorsement policy check*. The peers verify if the transaction gets enough endorsements according to the endorsement policy. (ii) *Version conflict check*. The peers check if the current world state is still consistent with the state when the transaction was generated at the execution phase through a version mechanism. Specifically, there is a field called *version* in a transaction's read/write set, which is generated in the execution phase. Peers check if the version in the read/write set is consistent with the version in the current world state. Only the transaction that passes both checks above will be marked as valid. Only results in a valid transaction then are committed to update the world state. After peers validate all the transactions, the block is appended to the Blockchain in the ledger. Finally, the client gets a notification about the status of the transaction initiated by the client.

Fig. 3 shows an example block that contains one transaction in Hyperledger Fabric. A block is composed of a header, a list of transactions and the metadata. The metadata includes the flag vector indicating the validity of each transaction. A transaction has 4 parts: transaction header, transaction proposal, proposal response and endorsements.

III. PDC Use Cases

The private data collection (PDC) is the sensitive data shared by a subset of organizations in a channel. In this section, we first present its storage and workflow design and then the discovered three classes of use cases where misused Fabric features will cause security issues.

A. Private Data Collection Design

*1) **Public and Private Data Storage Comparison**:* Private data is stored in a different way from public data in the world state. Public data is stored in the form of ⟨key, value, version⟩ at all peers in the channel. Private data has two storage formats: the original ⟨key, value, version⟩ is stored at a subset of peers that are PDC members while the hashed ⟨hash(key), hash(value), version⟩ is stored at all peers. For example in Fig.1, $P1$ and $P4$ store both ⟨key, value, version⟩ and ⟨hash(key), hash(value), version⟩. $P1$ only stores ⟨hash(key), hash(value), version⟩.

*2) **Public and Private Data Transaction Workflow Comparison**:* The PDC transaction workflow differs in the execution phase from the public data transaction workflow as shown in Fig. 2. The PDC read/write set in the proposal response is hashed to avoid exposing the original private data. The endorsers such as peer2 keep and send the original read/write set via the peer-to-peer gossip protocol to PDC member peers who are not endorsers such as peer3 and need the original read/write set in the validation phase. The transaction with the read/write set hash is packaged into the block and distributed to all peers by orderers and all peers will validate the transaction. Before updating the ledger, the PDC member peers verify if the original read/write set matches the hash in the transaction.

B. Use Case 1: PDC Non-member Peers Endorse PDC Transactions

*1) **Semantics of Read/Write Set**:* In the transaction workflow in Fig. 2, peers don't update the ledger in the first execution phase but generate a read/write set. The read set stores a list of ⟨key, version⟩ of the read operation results and is used for version conflict check in the validation phase. The version is obtained by querying the world state. The write set stores the results of write operations in a list of ⟨key, value, is_delete⟩ derived from the chaincode without interacting with the world state. The is_delete flag is set to "false" for the write operation. The delete operation uses the same semantics as the write operation. If the operation is to delete a key (an entry in the world state), is_delete is set to "true" and the corresponding value is set to "null". We group transactions into three types: read-only, write-only and read-write. Table I shows the read/write sets in these three types of transactions. It can be observed that the read set in a write/delete-only transaction is null, which is different from the read-only and read-write transactions.

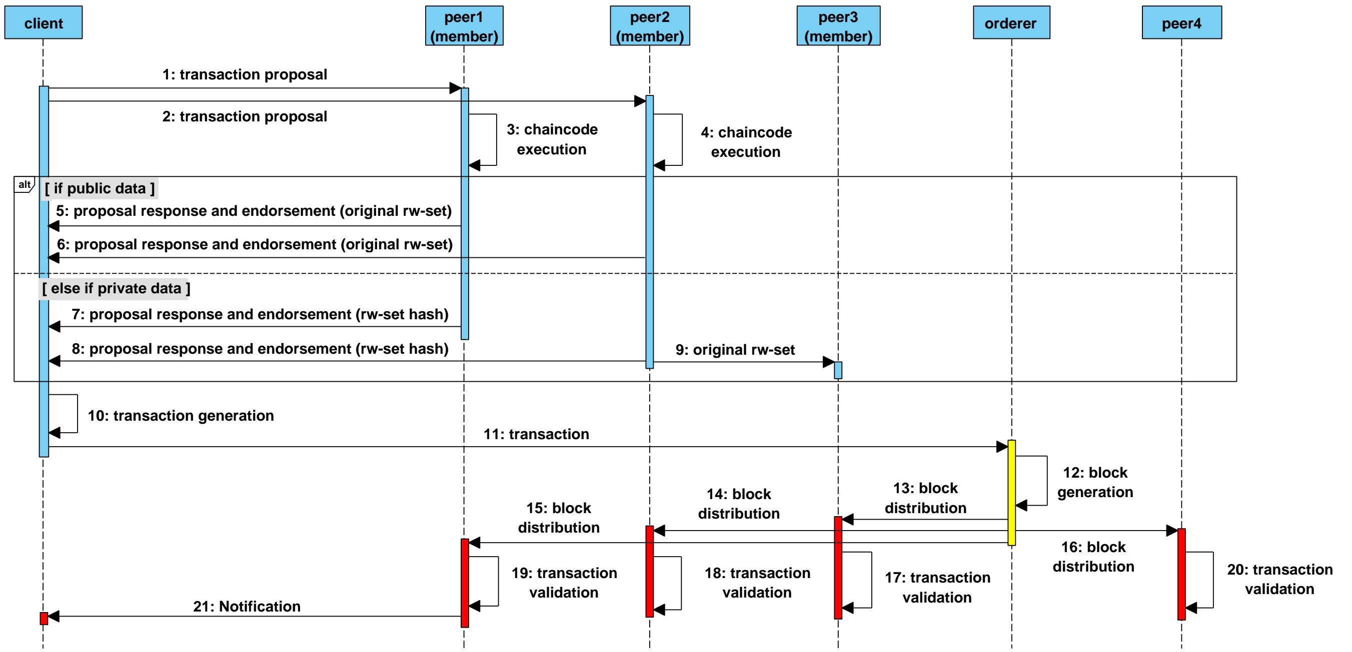

Fig. 2. Sequence diagram of three-phase transactions workflow in Unified Modeling Language (UML). The *alt* frame is the alternative combined fragment, modeling the if-then-else logic. (I) Public data transaction workflow (Steps 1-6 and 10-21); (II) Private data transaction workflow (Steps 1–4 and 7-21), where peer1, peer2 and peer3 are PDC members. The rw-set denotes the read/write set.

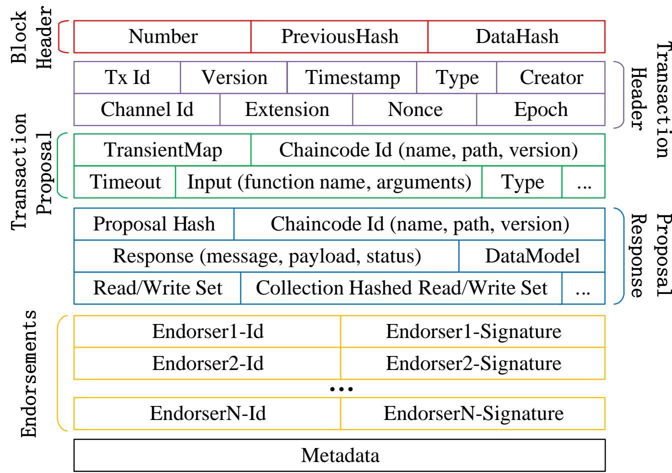

Fig. 3. A block with one transaction. Only related fields are shown and the shown fields are not necessarily in the actual order of these fields in the block.

TABLE I
READ/WRITE SET IN DIFFERENT TYPES OF TRANSACTIONS OPERATING ON ⟨K1, VAL1⟩. ASSUME THE VERSION OF K1 IS 1.

Transaction Type	Read Set		Write Set		
	key	version	key	value	is_delete
Read-only	k1	1	NULL		
Write-only	NULL		k1	val1	false
Read-Write	k1	1	k1	val1	false
Delete-only	NULL		k1	null	true

*2) **PDC Non-member Peers Endorsing Transactions on Private Data:*** In the Fabric, PDC non-member peers may endorse transactions on private data. Recall that PDC non-member peers do not hold the original private data in the world state. We now show how a PDC non-member endorser deals with read and write transaction proposals and the difference shall be carefully considered in the design of an actual Fabric system.

A PDC non-member endorser returns errors when asked to process a read-only or read-write transaction proposal. To endorse a read-only or read-write transaction, the endorser shall query the world state and return ⟨key, version⟩ in the read set. A peer from PDC non-member organizations cannot complete such endorsement this way since they have only the private data hash in its world state. If a non-member peer tries to execute private data read operations, the Fabric reports an error since the corresponding key cannot be found in the world state.

A PDC non-member endorser does not return errors when asked to process a write-only transaction. Recall that a write-only transaction contains no read operation and its read set is Null. It means that results of the write-only transaction do not rely on any data in the world state. All endorsers including PDC non-member endorsers in the channel can successfully return endorsements for private data write-only transactions to clients with no errors.

It can be observed that PDC non-members can endorse PDC write-only transactions, but not read related (read only or read-write) transactions given that write operations write data into the world state and read related operations read the data from the world state.

C. Use Case 2: PDC Transactions Validated through the Same Endorsement Policy as Public Data Transactions

*1) **Endorsement Policy**:* Each chaincode has its specific endorsement policy that specifies which endorsers shall sign a transaction. A policy is a logical expression, which accepts n Boolean inputs and returns one Boolean output. For example the endorsement policy "*MAJORITY Endorsement*" can be written as follows,

$$Majority(e_1, e_2, ..., e_n) = \lfloor \frac{1}{2} + \frac{(\sum_{i=1}^{n} e_i) - 1/2}{n} \rfloor, \quad (1)$$

where n is the number of organizations, and e_i is the Boolean output of the signature policy named "Endorsement" of the ith organization org_i. For example, org_i can define the "Endorsement" policy as "*OR(org_i.peer)*" so that an endorsement from any peer of org_i produces 1 for e_i. If peers from majority of organizations sign the transaction, $Majority(e_1, e_2, ..., e_n)$ returns 1 and the transaction passes the endorsement policy check.

*2) **PDC Transactions Validated Using the Chaincode-level Policy**:* Each chaincode has a chaincode-level endorsement policy. By default, the chaincode-level endorsement policy applies to both public and PDC transactions. Hyperledger Fabric allows to customize a collection-level endorsement policy for a PDC. Based on source code [7] of the Fabric, the read-only transactions are always validated using the chaincode-level policy, and the read-write and write-only transactions are validated using the chaincode-level policy if no collection-level endorsement policy is defined.

The validation phase checks if the list of endorsements in a transaction satisfies the endorsement policy and does not differentiate between endorsements from PDC members and from non-members. That is, the validation phase does not check if a PDC transaction is endorsed by a PDC non-member peer.

D. Use Case 3: "Payload" Field Used to Return Information in Transaction Proposal Response

As shown in Fig. 3, a transaction has an important field called proposal-response, which contains two crucial parts: (i) `Response` returned to the client has its own three fields: payload, status and message. The "payload" carries the information returned by the corresponding chaincode function. The "status" indicates if the chaincode function successfully executes. If an error occurs, the "message" will describe the specific error. (ii) `Read/Write set` is used for validating transactions and updating the world state in the validation phase. Recall that the read set contains ⟨key, version⟩ that the chaincode function reads from world state, but not the corresponding "value" stored in the world state as introduced in Section III-B1. If the client requires the value, the chaincode function can return this value through the "payload" field. For private data, the read/write sets are hashed to avoid revealing the sensitive private data collections. **It shall be noted that the "payload" in `Response` is in plaintext for PDC transactions.**

IV. Attacks and Defense

In this section, we show that malicious peer/client nodes can conduct the fake PDC (read/write) results injection attack and PDC leakage attack against the misused PDC features presented in the three use cases in Section III. We will also introduce our new Fabric features to mitigate these attacks.

A. Fake PDC Results Injection Attack

For a Fabric system, the chaincode manages both public data and PDC data. There exist use cases in which the implicitMeta chaincode-level policy, such as MAJORITY or NOutOf, is needed for public data. In those cases, misuse may occur and PDC non-member peers may manipulate the PDC read/write results by colluding with each other or colluding with malicious PDC member peers to disrupt the integrity of the ledger, blockchain or world state. (i) The chaincode-level policy uses implicitMeta policies and there is no collection-level policy. Fake PDC (read/write) results injection attacks can work. (ii) The chaincode-level policy uses implicitMeta policies and collection-level policy is used to explicitly specify endorsers to avoid undesired endorsers. Attacks on write-related transactions cannot work, but the attacks on read-only transactions can work since read-only transactions are validated only through the chaincode-level policy.

*1) **Fake Read Result Injection**:* The fake read result injection can be deployed when a PDC system allows endorsements from PDC non-member endorsers (Use Case 1), and read-only transactions are validated only using the chaincode-level policy (Use Case 2). When a client sends a PDC read-only transaction proposal to a PDC member endorser, the endorser generates a read set of ⟨key, version⟩ and a null write set. The value of the key is returned in the transaction's "payload" field of "proposal-response". When the client sends the PDC read-only transaction proposal to a PDC non-member endorser, the endorser cannot find the key or value in its world state. It returns an error to the client and the endorsement fails.

We now present an attack that allows PDC non-member peers to endorse PDC read-only transactions without errors. This shows that endorsers shall be carefully chosen for an endorsement policy and implicitMeta policies can be dangerous.

Endorsement Forgery. A PDC non-member endorser can collude with other endorsers, which can be either PDC member or non-member endorsers, and they collaboratively forge the same ⟨key, version⟩ and $value$ in the "payload" to endorse read-only transactions as follows.

- *Obtaining* $version$ *through GetPrivateDataHash(.).* When a PDC *member* endorser receives a PDC read-only transaction proposal from a client, it shall use the API `GetPrivateData(collection, key)`, and query its world state to get the genuine ⟨key, version⟩ and the $value$. If a malicious PDC *non-member* endorser invokes `GetPrivateData(collection, key)`, an error occurs since the malicious PDC non-member does not have the private data. However, the malicious PDC non-member peers can invoke another API `GetPrivateDataHash(collection,`

`key)` which generates the same *version* as `GetPrivateData(collection, key)`. Note *collection* and *key* are sent from the client and *version* is taken from the world state. It will not report an error since any peer can invoke this API to get the private data hash stored in every peer in the channel.

- *Obtaining value in the "payload" through customizable chaincode.* In the Fabric, chaincode does not need to be identical at all peers in one channel as long as the execution results are the same across the endorsers. This feature of customizable chaincode is designed for organizations to extend their specific business logic, such as additional validation before endorsing a transaction or integration of data from their existing system into the chaincode. However this feature allows malicious peers to put malicious logic into the chaincode. A group of malicious peers can extend the chaincode to collude with each other. For example, malicious endorsers can collaboratively customize the chaincode function to return the same fake *value* in the "payload". With ⟨key, version⟩ obtained with `GetPrivateDataHash(collection, key)` and the fake "payload" from colluding with other endorsers, the malicious PDC non-member endorser can create a correct proposal-response and the corresponding endorsement.

In the Fabric, the version conflict check in the validation phase only checks if the version in the read set ⟨key, version⟩ is consistent with the current world state, and does not re-execute the chaincode to check the correctness of execution results. The fabricated proposal-response can pass the version conflict check. Therefore, when the chaincode-level policy such as the MAJORITY policy accepts endorsements from PDC non-member endorsers, enough endorsements from PDC non-member endorsers can pass the endorsement policy check in the validation phase. The fabricated read-only transactions endorsed by PDC non-member peers can be appended into the immutable blockchain as valid while the *value* in the "payload" is fabricated. The integrity of the blockchain is breached.

*2) **Fake Write Result Injection**:* When the chaincode deployed in PDC non-member peers does not provide appropriate checks of write value, the fake write result injection can occur. A malicious client may fabricate write results in a transaction to change the private data in the world state and thus disrupt the integrity of the world state.

Considering Use Cases 1&2, all peers including PDC non-member peers can generate an endorsement for PDC write-only transactions. Therefore, a malicious client can get endorsements from any endorsers. By default, PDC write-only transactions are validated using the chaincode-level policy, where endorsements from PDC non-member endorsers can satisfy the policy.

In the write-only scenario, different private data owners may define different constraints for write operations since the customizable chaincode allows endorsers to extend additional code to describe their own specific logic. Intuitively, the PDC non-member peers with no interest in such private data will add no constraints. Therefore, a malicious client can exploit this practice and send transaction proposals to PDC non-member peers to get the required number of endorsement to pass the endorsement policy validation. In this way, the malicious client may change the PDC in the world state at will, and violate the business logic of some PDC member peers. The integrity of the world state is broken too.

*3) **Fake Read-Write Result Injection**:* The read-write transactions are often used to update data stored in the world state. For example, an *add* function first reads the data in the world state, next adds a value to it, and then uses the sum to update the data in the world state. The attacks introduced in Section IV-A1 and Section IV-A2 can be combined to conduct more powerful attacks.

In a read-write transaction, the value obtained from the read operation may be used in the write operation (such as the add operation) or used in control statements such as "if-else". Through the fake read result injection in Section IV-A1, the malicious endorsers can fabricate a fake read value, so as to change the value in the write set or change the control flow of a chaincode function. Further through the fake write result injection in Section IV-A2, the attackers can update the world state with the fake value in the write set and this disrupt the integrity of the world state.

*4) **PDC Delete Attack**:* As introduced in Section III-B1, a delete operation is the special case of a write operation. The system won't report an error when PDC non-member peers endorse the delete-only transactions. So any non-member peers also can endorse delete-only transactions on private data. The attacks on write-only transactions are also valid in delete-only transactions, and can be combined with the other operations to conduct more sophisticated attacks.

*5) **51% Attack Discussion**:* When the chaincode-level endorsement policy is set to MAJORITY, our attacks require that 51% of the organizations in one channel have malicious peers and clients. This is similar to 51% attacks against Blockchain systems such as Bitcoin and Ethereum [8]–[10]. In our attacks, malicious PDC non-member endorsers need to collude together to fabricate required number of endorsements to satisfy the MAJORITY policy. If there is no enough non-member peers, they need to further collude with PDC member peers.

The chaincode-level endorsement policy can also use the *NOutOf* policy. In this case, it is possible that our attacks require peers from far less than 51% malicious PDC non-member organizations. For example, *2OutOf(org1.peer, org2.peer, org3.peer, org4.peer, org5.peer)* means endorsements from any two peers out of the 5 listed organizations can satisfy the policy. Assume org1 and org2 share a PDC. Endorsements from org3.peer and org4.peer also can satisfy the *NOutOf* policy even though both of them are PDC non-member organizations. Org3 and org4 can collude to conduct all the proposed attacks above to manipulate the private data without colluding with any PDC member peer.

B. Private Data Leakage

In Use Case 3, the "payload" filed in the proposal-response may reveal the private data to non-member peers though both PDC-read and PDC-write transactions with no need of peers or clients being malicious.

1) PDC Leakage through PDC Read Transactions: We first check how a PDC read transaction is processed in the Fabric. A Fabric system may want to record who performs the PDC read into the ledger for the purpose of auditing [11]. In such a case, a client sends a transaction proposal to multiple endorsers using the API `submitTransaction(name,[args])`. In this way, the client will receive multiple proposal responses and endorsements from the endorsers. Then the client assembles a transaction, which is then sent to the orderers and put into the block. The block is then distributed to all peers in the channel. All peers can obtain the contents of a transaction by just fetching it from their local blockchain.

In the Fabric, the chaincode function returns the requested PDC value to the client through the "payload" field in `Response` of a transaction's proposal-response. The ⟨key, version⟩ part of the target PDC is stored in the read set. According to Use Case 3 in Section III-D, only the PDC read/write set of a transaction is hashed but the "payload" in `Response` of proposal-response keeps the original value. Therefore, all peers including PDC non-members can search the local blockchain and obtain the private data value while by design PDC is shared by only a subset of peers in one channel for data isolation and confidentiality. Therefore, the PDC leakage due to Use Case 3 violates the design principles of PDC. Therefore, this use case shall be avoided.

2) PDC Leakage through PDC Write Transactions: In a PDC write transaction, a client must use `submitTransaction(name,[args])` to invoke PDC write chaincode functions so that a transaction can be generated and the update to the world state can be fulfilled in the validation phase. A sloppily written chaincode function may return the PDC value or other sensitive information through the "payload" field as response to the client. The "payload" field will reveal the sensitive information to all peers through the block distributed to all peers. Our experiments in Section V-B2 show that the GitHub Fabric projects at GitHub may leak PDC this way. A designer in such a use case shall understand the payload field is in plaintext even for PDC operations.

C. Defense

Sections IV-A and IV-B show that endorsers shall be carefully chosen and implicitMeta policies may be dangerous if there are untrustworthy organizations and peers. We present two new features for a Fabric system to defeat the fake PDC results injection attack as well as the PDC leakage attack.

1) New Feature 1: Collection-level Policy Check for PDC Read Transactions during Validation: This feature can be used to defeat the fake PDC read results injection. With Use Case 2 in Section III-C, PDC read-only transactions are validated

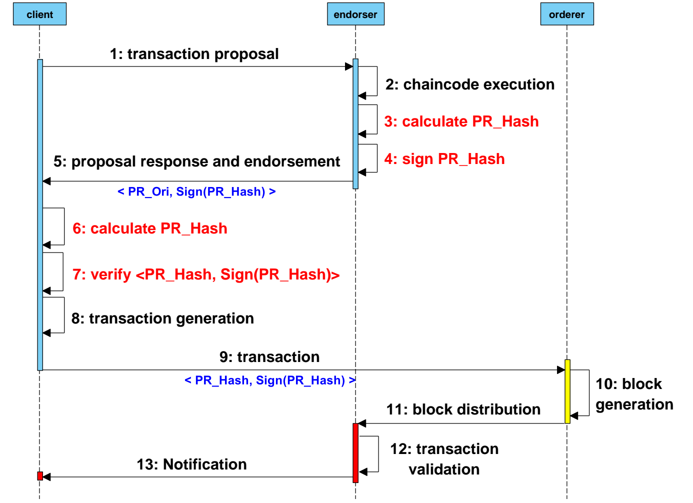

Fig. 4. Cryptographic solution to fix Design Issue 3

solely using the chaincode-level policy. We propose to add the collection-level policy check for PDC-read operations. If a collection-level policy is defined, the validation phase will apply such policy to PDC read-only transactions. Otherwise, the chaincode-level policy is adopted. To defeat the fake PDC write results injection, collection-level policy may be used to explicitly specify endorsers to avoid undesired endorsers when the chaincode-level policy uses implicitMeta policies.

2) New Feature 2: Cryptographic Solution in the Execution Phase: In Use Case 3, the "payload" in the `Response` field of proposal-response can reveal the private data to all peers including PDC non-member peers. An intuitive approach to prevent such private data leakage is to hash the "payload" in proposal-response, which is returned from an endorser to the client. However, the client needs to obtain the "payload" in plaintext since that is what the client asks for.

We propose to modify the work mechanism of both endorsers and clients as shown in Fig. 4 with newly added Steps 3-4 and 6-7. For the endorsers, they still return the proposal-response with the original "payload" (i.e. PR_Ori) to the client to provide the PDC read service. The modification is that the endorser signs the proposal-response with hashed "payload" (i.e. PR_Hash), not the original one. Then the endorser returns ⟨PR_Ori, Sign(PR_Hash)⟩ to the client. The client receives the proposal-response and the signature. It first calculates the hash of "payload" again to get PR_Hash using the same algorithm as endorsers (SHA256 in Hyperledger Fabric), then verifies the signature, and assembles the transaction using ⟨PR_Hash, Sign(PR_Hash)⟩. The remaining ordering and validation phases proceed without any modifications. In this way, the client receives the private data and the transaction does not contain the private data.

V. Evaluation

We build prototype systems following the test-network guideline in Fabric official documents [12] [13] to demonstrate the effectiveness of our attacks and defense measures. We also analyze the Fabric projects at GitHub to evaluate the

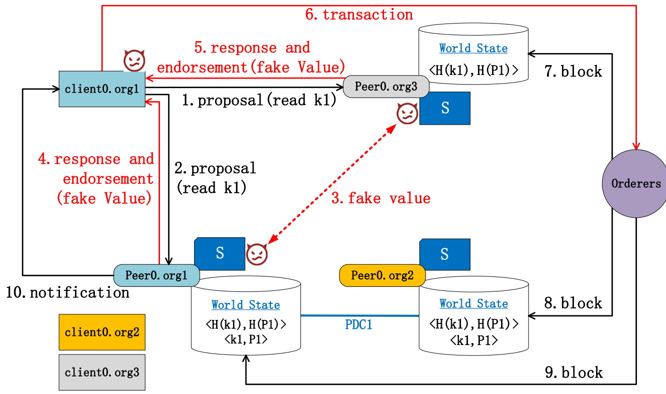

Fig. 5. Injecting fake results into read-only transactions

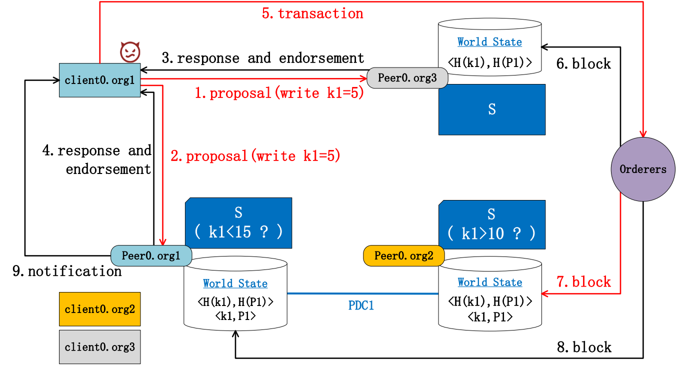

Fig. 6. Injecting fake results into write-only transactions

generality of our attacks. All the experiments were performed on a MacBook Pro. Dockers were used to run as nodes and form a Fabric system.

A. Fake PDC Results Injection Attack

Without loss of generality, we test the attacks against a Fabric system of three organizations joining a channel. Each organization provides one peer node and one client node, denoted as peer0.org1, peer0.org2, peer0.org3, client0.org1, client0.org2 and client0.org3. Org1 and org2 are the member organizations of the PDC1 and maintain private data $\langle$k1, P1$\rangle$. PDC1 uses the default chaincode-level *MAJORITY Endorsement* policy, which is the most popular chaincode-level policy as shown in Section V-C2. Org1 and Org3 are malicious, and Org2 is the victim.

*1) **Fake Read Result Injection Attack**:* Fig. 5 shows the fake read result injection attack. Peer0.org1 and peer0.org3 have malicious chaincode to conduct *Endorsement Forgery* as introduced in Section IV-A1. The malicious client0.org1 sends a read proposal to peer0.org1 and peer0.org3, which return the proposal-response and endorsements with the same fake $value$ in the "payload" and the valid $\langle$key, version$\rangle$ to client0.org1. client0.org1 automatically assembles a transaction and sends it for ordering. This transaction passes the validation at all peers. The malicious transaction is marked as valid and put into the blockchain at all peers. The integrity of the blockchain is broken with the fake value.

*2) **Fake Write Result Injection Attack**:* Fig. 6 shows the fake write result injection attack. A malicious client0.org1 starts a write transaction of setting $k_1.value = 5$ by sending a proposal to peer0.org1 and peer0.org3. Peer0.org1's PDC *write* chaincode function requires $k_1.value < 15$ so as to move forward with its business logic. Since $k_1.value = 5$ is less than 15, peer0.org1 allows the write operation to move forward. Peer0.org3 is a PDC non-member with no constraint on PDC. Therefore, both peer0.org1 and peer0.org3 move forward to generate the proposal-response and endorsements and send them back to client0.org1, which then assembles a valid transaction. This transaction will eventually be used to update the world state at all PDC member peers including peer0.org2 in the validation phase. However, if peer0.org2 requires $k_1.value > 10$ in its business logic, it can be observed that the fake write result injection attack violates peer0.org2's business logic and puts $k_1.value = 5$ into the world state of peer0.org2 which requires $k_1.value > 10$.

*3) **Fake Read-Write Result Injection Attack**:* In this experiment, the read-write chaincode function first reads the value of k_1, then adds a value to k_1 and use the sum to update k_1. Peer0.org1 requires $k_1.value < 15$, peer0.org2 requires $k_1.value > 10$ and peer0.org3 has no constrains on $k_1.value$. Following the fake read result injection attack, peer0.org1 and peer0.org3 collude together to forge a fake read value of k_1 to make the sum less than 10, such as 5. Following the fake write result injection attack, the malicious client0.org1 successfully changes $k_1.value$ to 5 which violates org2 by getting endorsements from peer0.org1 and peer0.org3.

*4) **PDC Delete Attack**:* The delete operation is a special case of the write operation as introduced in Section III-B1. In this experiment, peer0.org1 require $k_1.value < 15$ to delete k_1. Peer0.org2 require $k_1.value > 10$ to delete k_1. Peer0.org3 has no constraint. When $k_1 = 5$, our experiments show the malicious client0.org1 can successfully delete k_1 at both peer0.org1 and victim pee0.org2 with endorsements from peer0.org1 and peer0.org3.

*5) **Attacks under the NOutOf Endorsement Policy**:* In this experiment, we add two more organizations (i.e. org4 and org5) to the prototype system. Peer0.org4 and peer0.org5 are PDC1 non-member peers and the chaincode-level endorsement policy is set to *2OutOf(org1.peer, org2.peer, org3.peer, org4.peer, org5.peer)*. We assume peer0.org3 and peer0.org4 are malicious. PDC member peer0.org1 and peer0.org2 are the victims. Our experiments show that all attacks above can get endorsements from peer0.org3 and peer0.org4 and achieve the desired attack results. Therefore, our attacks work against PDC transactions with no need of malicious peers from over 50% organizations.

*6) **Attacks under the Collection-level Policy**:* In this experiment, we define a collection-level endorsement policy of *AND(org1.peer, org2.peer)* for PDC1 and conduct the above attacks again. Due to Use Case 2, the system still uses the chaincode-level policy to validate the read-only transactions. The fake read-only result injection attack still works and the read-only transaction with endorsements from peer0.org1 and

peer0.org3 is still marked as valid. The fake write-only, read-write and delete result injection attacks fail due to the use of the collection-level policy validating transactions. However, according to generality analysis in Section V-C, most projects do not define the collection-level policy. Our defense measure can defeat all the attacks even if there is no collection-level policy and the chaincode-level policy is used.

B. Private Data Leakage

We use two vulnerable projects [14] [15] at Github to demonstrate the PDC leakage. We build two prototype systems according to the guideline of these two Github projects.

1) ***PDC Leakage through PDC Read Transactions****:* In one GitHub project [14], the system contains pee0.org1 and peer0.org2. Org1 is the PDC member. In its chaincode snippet shown as Listing.1, a function called `readPrivatePerfTest` provides PDC-read service and uses the API `getPrivateData` to get the value of the PDC key `perfTestId`. The required PDC value is stored in a variable `asset` and is returned in Line.10. Its client application code uses `submitTransaction(name,[args])` to invoke the function `readPrivatePerfTest`. Once the client of org1 reads the PDC value, the corresponding transaction is distributed to all peers through blocks. PDC non-member peer0.org2 can then fetch and parse the transaction in the local ledger to obtain the original PDC value.

Listing 1. Chaincode Function in the PDC-Read Case

```
1 // The language is Node.js.
2 async readPrivatePerfTest(ctx,perfTestId)
3 {
4     const exists = await this.privatePerfTestExists(
          ctx, perfTestId);
5     if (!exists) {
6         throw new Error('The perf test ${perfTestId}
              does not exist');
7     }
8     const buffer = await ctx.stub.getPrivateData(
          collection,perfTestId);
9     const asset = JSON.parse(buffer.toString());
10    return asset;
11 }
```

2) ***PDC Leakage through PDC Write Transactions****:* In another GitHub project [15], the system contains three organizations org1, org2 and org3. Org1 and Org2 are the PDC members. The PDC's name is "demo". In the chaincode snippet as shown in Listing 2, a function called `setPrivate` provides PDC-write service. `Args[0]` is the target key and the `args[1]` is the value used to update the key. This function uses the API `PutPrivateData` to update the PDC, and returns `Args[1]` in Line.10, which can leak the PDC value. Each time the clients of org1 or org2 invoke the function `setPrivate` to update the PDC, the peers of non-member org3 receive the corresponding transaction, may parse this transaction in the local ledger, and obtain the original PDC value.

Listing 2. Chaincode Function in the PDC-Write Case

```
1 // The language is Golang.
2 func setPrivate(stub shim.ChaincodeStubInterface,
      args []string) (string, error)
3 {
4     if len(args) != 2 {
5         return '', fmt.Errorf('Incorrect arguments.
              Expecting a key and a value')
6     }
7     err := stub.PutPrivateData('demo',args[0],[]byte
          (args[1]))
8     if err != nil {
9         return '',fmt.Errorf('Failed to set asset :
              %s',args[0])
10    }
11    return args[1], nil
12 }
```

C. Generality of Our Attacks

To evaluate the generality of our attacks, we analyze Fabric projects available on GitHub ranging from January 1, 2016 to December 31, 2020 while the Fabric was first released in 2016.

1) ***Static Analyse Tool****:* We build a tool with Python to conduct a large-scale analysis of Fabric projects on GitHub. Our tool statically scans the use of PDC according to its features. There are two options to define a PDC. One is explicit definition which fits the scenarios of our attacks. The other is implicit definition which is out of scope of this paper since it is used for special cases where only one organization owns the private data. The projects involving the explicit PDC definition must define a ".json" configuration file to describe the PDC properties. The ".json" file has some fixed keywords including "Name", "Policy", "RequiredPeerCount", "MaxPeerCount", "BlockToLive", "MemberOnlyRead" and so forth. If these keywords are detected in a ".json" file of one project, the tool will classify it as an explicit PDC project. The implicit definition includes a field "_implicit_org_" that must appear in the parameters list of PDC operations APIs. If our tool finds "_implicit_org_" in the chaincode of a project, the tool will classify it as an implicit PDC project. Note that a Fabric project can use explicit and implicit PDC simultaneously.

For the explicit PDC projects, our tool detects how many of them use the chaincode-level endorsement policy by default, since the default policy may put the projects under our attacks. There is one optional property called "EndorsementPolicy" in the explicit definition configuration file. It is used to customize the collection-level policy for PDC. If "EndorsementPolicy" is not set, the corresponding project uses the default chaincode-level policy. Our tool looks for this feature and derives the number of explicit PDC projects that use the chaincode-level endorsement policy. Furthermore, our tool also finds what the default chaincode-level policy is. It can be found in a fixed field of a configuration file called "configtx.yaml".

2) ***Results****:* We find 6392 projects in total. There are 252 explicit PDC projects that our paper focus on and 35 implicit PDC projects. Among these PDC projects, 31 projects involve both explicit PDC definition and implicit PDC definition.

Growth trend across years. It can be observed from Fig. 7, since Fabric was introduced in 2016, the projects on Github have increased sharply, particularly in the year 2019 and 2020. It can also be observed that the use of PDC has been growing

TABLE II
ATTACK & DEFENSE EVALUATION SUMMARY. √ MEANS THE ATTACK WORKS. × MEANS THE ATTACK FAILS.

Use Cases	Attack		Default Policy: MAJORITY	Default Policy: 2OutOf5	Define Collection-level Policy : AND(org1, org2)	New Feature 1: Collection-level Policy Check for PDC Read Transactions	Original Fabic Framework	New Feature 2: Cryptographic Solution
1: Non-member Peers Endorse PDC Transactions	1: Fake PDC Results Injection	Read-Only	√	√	√	×	N/A	N/A
		Write-Only	√	√	×	×	N/A	N/A
2: PDC Transactions Validated through the Same Endorsement Policy as Public Data Transactions		Read-Write	√	√	×	×	N/A	N/A
		Delete-Related	√	√	×	×	N/A	N/A
3: Exposed "Payload" Field Used to Return Information in Transaction	2: PDC Leakage	PDC-Read	N/A	N/A	N/A	N/A	√	×
		PDC-Write	N/A	N/A	N/A	N/A	√	×

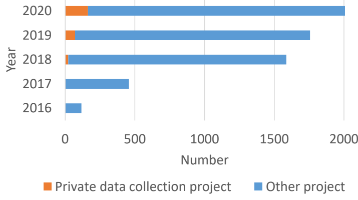

Fig. 7. Projects across years

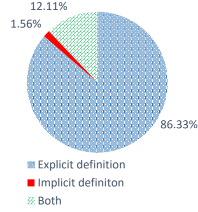

Fig. 8. PDC definition

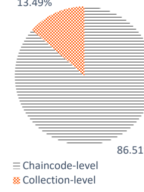

Fig. 9. Endorsement policy of explicit PDC projects

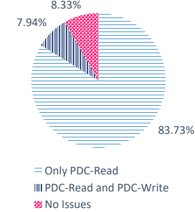

Fig. 10. PDC leakage issues

gradually. The private data mechanism was proposed in 2018, so there was no PDC projects in 2016 and 2017.

PDC type distribution. As shown in Fig. 8, the type of explicit PDC, which this paper focus on, is the most popular one and 98.44% of all PDC projects are of this type. Among them, 12.11% has the type of implicit. Only 1.56% of the PDC is implicit only.

Generality of fake PDC results injection attacks. In the 252 explicit PDC projects, 218 projects use the chaincode-level policy by default and only 34 projects customize the collection-level endorsement policy for PDC. Fig. 9 shows the distribution of endorsement policy types among explicit PDC projects. We can see that 86.51% of them use the chaincode-level endorsement policy, which is vulnerable as discussed in Section III-C.

Popularity of MAJORITY Endorsement policy. Our tool finds 120 configtx.yaml files in the 218 explicit PDC projects that use the chaincode-level policy. 116 out of these 120 configuration files use the *MAJORITY Endorsement* policy as the chaincode-level endorsement policy. This demonstrates that *MAJORITY Endorsement* is very popular. Recall in our experiments, we assume that the chaincode applies the *MAJORITY Endorsement* policy for all transactions and all these prototype projects that adopt the MAJORITY Endorsement policy are subject to our attacks.

Generality of PDC leakage issues. We manually analyse all 252 explicit PDC projects and find that 91.67% projects have such issues as shown in Fig. 10. 231 of these vulnerable projects involve functions that provide the PDC read service which return PDC and will leak the PDC. 20 of these 231 vulnerable projects also involve sloppily written PDC write functions that return PDC and can leak the PDC.

D. Defense Effectiveness and Efficiency

We have implemented the new features proposed in Section IV-C by modifying the source code of the Fabric framework. We also add a supplemental feature filtering PDC non-member endorsers during validation. This can be useful for preventing sloppy developers from crafting dangerous PDC related policies when PDC non-members are not wanted. For Use Cases 1 and 2, we modify the source code for endorsement policy check during validation. For Use Case 3, we modify the source code for endorsing the proposal-response and assembling transactions in the execution phase. With the new features in the modified Fabric framework, we perform the same attacks experiments in Section V-A and Section V-B and have verified all those attacks cannot succeed any more.

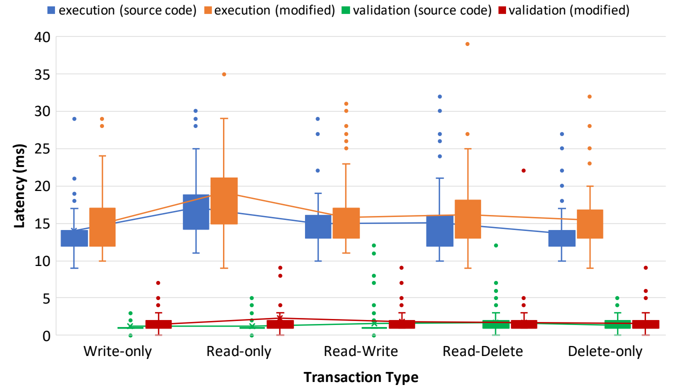

Fig. 11. Impact of defense measures on system performance

To evaluate the efficiency of our defense measures, we measure the execution latency and validation latency of processing one transaction respectively under the original and modified Fabric framework. Each measurement runs 100 times. Fig. 11 shows our new features have minor impact on the system running read/write/delete transactions.

VI. Related Work

In this section, we briefly review the most related work on security of Hyperledger Fabric and some attacks on vulnerable

smart contracts. It can be observed we are the first to identify the security problems caused by misuse of PDC of Fabric.

Yamashita et al. [16] summarize the potential risks in chaincode from multiple aspects. First, chaincode is developed using general-purpose programming languages. Some non-deterministic features of general-purpose programming languages may cause inconsistent endorsing results across different peers. Second, they point out that the concurrency of transactions is not properly designed, and phantom read may occur. There exist attacks against smart contracts in Ethereum [17]–[19], exploiting smart contract implementation issues.

Andola et al. [20] discuss the DoS attack and Wormhole attack against Hyperledger Fabric. In the DoS attack, a malicious insider may intentionally create errors in responses from endorsers so as to hinder transaction processing. In the wormhole attack, a compromised peer may leakage information within a channel to an outer adversary, which violates the confidentiality of the data.

Davenport et al. [21] review the attack surface in Hyperledger Fabric such as private key leakage and insecure CA. Hasanova et al. [22] believe that the sandboxing Docker [23] used in Hyperledger Fabric is not secure enough and may be subject to attacks such as Docker escape or resource abuse. Hyperledger Fabric is also susceptible to security flaws existing in general-purpose programming languages.

VII. Conclusion

In this paper, we explore the security problems caused by misuse of the private data collection (PDC) of the Hyperledger Fabric framework. We present three classes of use cases with respect to endorsing, endorsement policy, and transaction semantics where misused features cause security issues. We propose two group of novel attacks against such misuse: fake PDC results injection and PDC leakage. To demonstrate the severity and generality of these attacks, we perform a large-scale analysis of 6392 Hyperledger Fabric projects on Github, and find that 86.51% of the PDC related projects are potentially vulnerable to the fake PDC results injection attacks, and 91.67% have PDC leakage issues. We present new features for the Fabric system to defeat these attacks. The new features have minor or negligible impacts on the system performance.

Acknowledgements

This research was supported in part by National Key R&D Program of China 2018YFB0803400, National Natural Science Foundation of China under grants 62072103, US National Science Foundation (NSF) Awards 1931871 and 1915780, US Department of Energy (DOE) Award DE-EE0009152, and Jiangsu Provincial Key Laboratory of Network and Information Security under grants BM2003201. Any opinions, findings, conclusions, and recommendations in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," Manubot, Tech. Rep., 2019.

[2] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform," *white paper*, vol. 3, no. 37, 2014.

[3] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich *et al.*, "Hyperledger fabric: a distributed operating system for permissioned blockchains," in *Proceedings of the thirteenth EuroSys conference*, 2018, pp. 1–15.

[4] Forbes, "Forbes blockchain 50," 2020, [Online]. (Accessed 30 July 2020). [Online]. Available: https://www.forbes.com/sites/michaeldelcastillo/2020/02/19/blockchain-50/#1798b8417553

[5] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, 2014, pp. 305–319.

[6] IBM, "The ordering service," 2020, [Online]. (Accessed 30 July 2020). [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.0/orderer/ordering_service.html?highlight=raft

[7] ——, "Hyperledger fabric source code about policy check in the validation phase," 2020, [Online]. (Accessed 28 December 2020). [Online]. Available: https://github.com/hyperledger/fabric/blob/master/core/common/validation/statebased/validator_keylevel.go

[8] I. Eyal and E. G. Sirer, "Majority is not enough: Bitcoin mining is vulnerable," in *International conference on financial cryptography and data security*. Springer, 2014, pp. 436–454.

[9] A. M. Antonopoulos, *Mastering Bitcoin: unlocking digital cryptocurrencies*. " O'Reilly Media, Inc.", 2014.

[10] C. Feng and J. Niu, "Selfish mining in ethereum," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 1306–1316.

[11] IBM, "Glossary," 2020, [Online]. (Accessed 27 October 2020). [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/latest/glossary.html

[12] ——, "Using the fabric test network," 2020, [Online]. (Accessed 31 July 2020). [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.0/test_network.html

[13] ——, "Private data," 2020, [Online]. (Accessed 31 July 2020). [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.0/private-data-arch.html

[14] R. Acosta, "Vulnerable github project," 2020, [Online]. (Accessed 3 November 2020). [Online]. Available: https://github.com/acostarodrigo/fabricPerfTest/blob/master/functionalTests/PerfTestContract-perfTest%400.0.1.test.js

[15] K. Tam, "Vulnerable project in medium," 2020, [Online]. (Accessed 3 November 2020). [Online]. Available: https://github.com/kctam/privatedatadeepdive/blob/master/chaincode/sacc_private3org/sacc.go

[16] K. Yamashita, Y. Nomura, E. Zhou, B. Pi, and S. Jun, "Potential risks of hyperledger fabric smart contracts," in *2019 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, 2019, pp. 1–10.

[17] S. Falkon, "The story of the dao-its history and consequences," 2017, [Online]. (Accessed Apirl 2021). [Online]. Available: https://hackernoon.com/what-caused-the-latest-100-million-ethereum-bug-and-a-detection-tool-for-similar-bugs-7b80f8ab7279

[18] HACKERNOON, "What caused the accidental killing of the parity multisig wallet & how to detect similar bugs," 2017, [Online]. (Accessed Apirl 2021). [Online]. Available: https://hackernoon.com/what-caused-the-latest-100-million-ethereum-bug-and-a-detection-tool-for-similar-bugs-7b80f8ab7279

[19] M. Zhang, X. Zhang, Y. Zhang, and Z. Lin, "{TXSPECTOR}: Uncovering attacks in ethereum from transactions," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 2775–2792.

[20] N. Andola, M. Gogoi, S. Venkatesan, S. Verma *et al.*, "Vulnerabilities on hyperledger fabric," *Pervasive and Mobile Computing*, vol. 59, p. 101050, 2019.

[21] A. Davenport, S. Shetty, and X. Liang, "Attack surface analysis of permissioned blockchain platforms for smart cities," in *2018 IEEE International Smart Cities Conference (ISC2)*. IEEE, 2018, pp. 1–6.

[22] H. Hasanova, U.-j. Baek, M.-g. Shin, K. Cho, and M.-S. Kim, "A survey on blockchain cybersecurity vulnerabilities and possible countermeasures," *International Journal of Network Management*, vol. 29, no. 2, p. e2060, 2019.

[23] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.