Hochschule Kempten
University of Applied Sciences

Master Thesis

in Computer Science

Privacy Enhancing Audit Trail in Hyperledger Blockchain

submitted by

Hofmann Benedikt

Examiner:	Prof. Nikolaus Steger
Supervisors:	Dr. Prabhakaran Kasinathan
	Dr. Martin Wimmer
Date:	10. June. 2021
Faculty:	Informatik

Abstract

The emergence of blockchain and smart contracts facilitate the cooperation between multiple organizations without the need for a trusted third party. Importantly, the participating businesses are not at the mercy of crashes, bugs, or malicious behavior of a trusted third party if they use blockchain. In fact, the blockchain's ledger acts as *single source of truth* which organizations maintain collaboratively and use to orchestrate distributed multilateral workflows. Importantly, it involves the sharing of confidential data between a subset of the blockchain's participants and the execution of confidential workflows where individual tasks and the produced transactions are subjected to access control restrictions. However, the distributed shared ledger and the collaborative execution of smart contracts conflict with the access control restrictions of confidential data assets and confidential smart contracts. Consequently, different technologies and blockchain variants that promise to solve this conflict are surveyed. The focus is the blockchain Hyperledger Fabric (HLF). Then, a solution for a distributed multilateral workflow is designed and implemented using this blockchain. Finally, the proposed design is evaluated based on defined attacker models. The results show that HLF can be used to orchestrate distributed multilateral workflows with confidential data assets, also pitfalls and best practices were identified while developing the solution for the workflow.

Kurzzusammenfassung

Das Aufkommen von Blockchains und Smart Contracts erleichtert die Zusammenarbeit zwischen mehreren Organisationen, ohne dass eine vertrauenswürdige dritte Partei erforderlich ist. Wichtig ist, dass die teilnehmenden Unternehmen bei der Verwendung von einer Blockchain nicht den Abstürzen, Fehlern oder dem bösartigem Verhalten einer vertrauenswürdigen dritten Partei ausgeliefert sind. Vielmehr fungieren die Einträger in der Blockchain als "Single Source of Truth", die von den Unternehmen gemeinsam gepflegt und zur Steuerung verteilter multilateraler Arbeitsabläufe genutzt werden. Wichtig dabei ist, dass vertrauliche Daten zwischen einer Teilmenge der Blockchain-Teilnehmer geteilt werden und, dass vertrauliche Arbeitsprozesse ausgeführt werden, bei denen einzelne Aufgaben und die erzeugten Transaktionen Zugriffskontrollbeschränkungen unterliegen. Der verteilten gemeinsamen Einträge in der Blockchain und die kollaborative Ausführung von Smart Contracts stehen jedoch im Konflikt mit den Zugriffskontrollbeschränkungen von vertraulichen Datenbeständen und vertraulichen Smart Contracts. Daher werden verschiedene Technologien und Blockchain-Varianten untersucht, die versprechen, diesen Konflikt zu lösen. Der Fokus liegt dabei auf der Blockchain Hyperledger Fabric (HLF). Anschließend wird eine Lösung für einen verteilten multilateralen Arbeitsabläuf entworfen und unter Verwendung dieser Blockchain implementiert. Auch wird das vorgeschlagene Design anhand von definierten Angreifermodellen evaluiert. Die Ergebnisse zeigen, dass HLF verwendet werden kann, um verteilte multilaterale Arbeitsabläufe mit vertraulichen Datenbeständen zu steuern. Außerdem wurden Fallstricke und Best Practices bei der Entwicklung der Lösung für den Arbeitsprozess identifiziert.

Contents

List of Figures

List of Tables

1. Introduction

Complex and highly specialized products sometimes lead to multilateral, distributed business processes. *Multilateral*: not one but multiple organizations are involved in producing goods or services. *Distributed*: The workflow is not orchestrated by a central *trusted authority*. All organizations govern their own business processes. An example of a multilateral and distributed workflow is the "construction and certification of a power plant" use case. At least four organizations take part in the workflow. The Owner orders the power plant, the EPC is the contractor that is responsible for managing the power plant construction, NOBO is a certification body that certifies the correct execution of various partial goals and documents, and a Supplier builds parts for the power plant. Importantly, the amount of participants is small in comparison to a retail use case where one seller and potentially millions of buyers exist. Moreover, the participating organizations might not trust everyone that participates in the workflow equally. For example, the Owner does not necessarily know the Supplier which results in no trust relationship between these two organizations. Specifically, organizations have varying trust relationships with other organizations that participate. The result is that the orchestration of such processes is complex. In addition, each organization may have preferred communication mechanisms such as e-mail, fax, or mail. Also, there might be no established mechanism which allows all participants to verify the integrity and actuality of documents. Thus, each organization might have a different view of the workflow and its documents. For instance, an offer for the power plant construction got sent per mail by the EPC but it has not arrived because of errors in the distribution process. Hence, the Owner waits for the offer and the EPC waits for an answer from the Owner. Thus, the process gets delayed or halts. Moreover, paper documents can easily be manipulated. For instance, the Owner ordered a power plant which can produce up to 10MW. However, the delivered power plant is only capable of producing 9MW. Thus, the Owner starts a dispute that the delivered power plant is insufficient. Then, the EPC changes his version of the offer to list the power plant 9MW. The result is that the Owner and EPC have different views of the offer and there is no fast and accessible solution to verify the integrity of the offer. For instance, a notary might solve this for paper contracts. However, such entities are trusted third parties which may be corrupt or incorrect which is a reason why a digital solution without a trusted third party is desired which solves the issues of a centralized audit trail which can

be manipulated by one participant without any other participants noticing it. Therefore, this thesis takes a look at the blockchain technology which enables trusted transactions among untrusted participants in a network [34]. Since Satoshi Nakamoto's publication [56] in 2008, blockchain has been a frequently discussed topic in securing data storage and data transfer. The main component is the cryptographic-based distributed ledger that enforces *immutability* and therefore allows for protections against *tampering*. The ledger is made up of blocks which are linked to their predecessors with their hash. These blocks are distributed to every participating node in the blockchain network which enables *transparency*. The value of a trustless, decentralized, immutable ledger has been recognized by many industries that are looking to apply these concepts to existing business processes [77]. Each node can verify the validity of the chain with the references to the predecessor and subsequently to the genesis block - the first block [76]. These properties make blockchains an interesting choice for audit trails. The second breakthrough development were smart contracts which were introduced by Nick Szabo [78]. Smart contracts enforce *computational integrity* without a central authority. A consensus protocol ensures that only valid updates are made to the ledger. These smart contracts can be used to orchestrate distributed multilateral workflows. For instance, the workflow can be implemented as a smart contract which changes the state of the blockchain according to the workflow's requirements. Thus, the state of the workflow and the necessary documents can be shared and validated with smart contracts and the blockchain.

However, the workflow deals with various *confidential data assets*, for instance, the offer for a specific part. These are documents which must only be accessible to a subset of the participating organizations. Also, certain parts of the workflow are confidential as well. For example, only a subset of the participants can know of the Suppliers and the workflows which concern them. The result is that certain workflows must only be visibly to a subset of the organizations. In short, multiple organizations want to collaborate and need to share confidential data assets without using a trusted authority. In addition, these organizations want to keep certain parts of their workflows confidential and produce an audit trail which can be used to verify the correct workflow execution and which can verify the integrity of exchanged data assets. However, smart contracts are often naively *re-executed* on all nodes of the blockchain network to verify their correct execution [18]. Moreover, Re-execution is only possible if transaction inputs are available to validating nodes. Therefore, this accessibility result in a lack of confidentiality and privacy thus restricting the scope of applications that use smart contracts and blockchains [83].

This conflict between an immutable shared ledger that is visible to all participants and the required confidentiality of data assets and workflows of the aforementioned use case

is the central focus of this thesis. Further, a solution for the use case is to be designed using the HLF blockchain.

1.1. Problem Statement

The multilateral distributed workflow of the "construction and certification of a power plant" is to be implemented with the HLF blockchain. Importantly, the workflow includes *confidential* data assets and workflows. Moreover, if data assets of the workflow is distributed through a blockchain which is shared among all participants and the workflow is implemented with smart contracts that are collaboratively executed then it is apparent that the *confidentiality* of data assets and workflows conflicts with the shared nature of blockchain. Thus, this conflict is the main challenge of this thesis. Further, the execution of a multilateral distributed workflow must produce an audit trail which can be used to verify the correct execution of the workflow and which can verify the integrity of data assets. However, this privacy enhancing audit trail must not conflict with the confidentiality requirements of the data assets and workflows. Thus, the goal of the thesis it to design, implement and evaluate a solution for this use case. Moreover, technologies to implement a multilateral workflow are identified.

1.1.1. Research Questions

The thesis attempts to answer the following research questions:

Research Questions (RQ)	
RQ1:	What technologies can be used to implement multilateral distributed workflows that work with confidential data assets and workflows?
RQ2:	What are important design choices when designing a solution for multilateral distributed workflows when using HLF?

Table 1.1.: Table of all research questions.

1.2. Scope

The scope of the thesis is based on the requirements analysis of Chap. 2 and the research questions that are defined in Sec. 1.1. In brief, a solution for the distributed multilateral workflow "construction and certification of a power plant" using the Hyperledger Fabric blockchain will be created. The following points are considered withing the scope of the thesis:

- *Formulating a succinct representative use case*: Create a use case based on the works of Stahnke [76], Kasinathan, and Wimmer [70, 71] that focuses on identifying confidential data assets and confidential workflows via a commonly agreed distributed multilateral workflow. In particular, the use case is representative for the "construction and certification of a power plant" use case. Moreover, the use case is condensed to its core features and omits things like price negotiation or supplier selection. Furthermore, a requirements analysis is executed which identifies the key features of the use case.
- *Identify state-of-the-art technologies*: Survey different blockchains which might satisfy the requirements of the defined use case. Further, identify technologies which may aid blockchains to deal with confidential data assets and workflows.
- *Design a solution for the use case*: Design and implement a prototype that represents a solution for the use case. In particular, it shows which features of HLF solves which requirement. Moreover, it precisely identifies suitable technologies, designs and concepts for solving the use case. Further, only the parts of the design that run within the blockchain network are implemented.
- *Identification of important design choices*: The design of the use case shows what implications certain design choices may have. The implication and significance are discussed and pitfalls are identified. The creation and application of attacker models are withing the scope.
- *Requirements mapping*: Each requirement that is identified will be mapped to features withing HLF or parts of the created concept that has been created.

Further, the use case is not a classical retail use case where millions of buyers buy one standard product for one seller. Thus, transaction throughput and transaction latency are not focused on. Also, only technologies that were available as open source were considered for the thesis.

1.3. Structure of the Thesis

The thesis has the following structure. Chapter 1 presents the goal of the thesis. Next, Chapter 2 introduces the use case and its requirements. Further, Chapter 3 presents the background of this work which includes DLT's, HLF, and other innovative technologies. Then, Chapter 4 presents the concept for the use case. Further, Chapter 5 details the implementation of the prototype for the concept. Also, Chapter 6 evaluates the solution based on proposed attacker models, shows pitfalls, and provides lessons learned. Lastly,

the conclusion is drawn in Chapter 7, which also points out possible future research directions.

2. Use Case

The use case that is used for this thesis is discussed in this section. Importantly, the use case involves a distributed multilateral workflow. Importantly, the use case involves a small number of stakeholders that work together to produce a highly complex product for one buyer. Further, the consequence of errors is significant, the costs are considerable, and at the moment these processes are mostly paper based. The challenge is to digitize this process while securing the confidentiality of data assets and workflows and simultaneously creating a privacy enhancing audit trail which can attest the correct execution of the workflow. First, Sec. 2.1 will outline the context of the "construction and certification of a power plant" use case. Further, Sec. 2.2 will describe a reduced use case that will be used throughout this thesis. Lastly, Sec. 2.3 conducts the requirements analysis for the reduced use case.

2.1. Overview

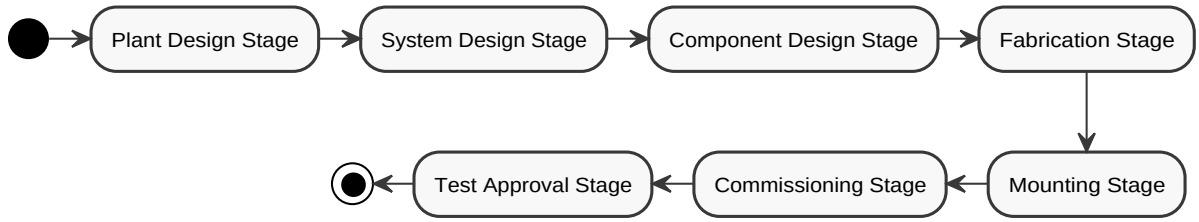

Figure 2.1.: State diagram that illustrates the different stages of the construction and certification of a power plant use case according to [76].

Generally, the use case describes the process of constructing and certifying a power plant is composed of various stages (see Fig. 2.1). First, the plant is designed during the *plant design stage*. Next, multiple system designs that make up the plant are created in the *system design stage*. Further, the *component stage* sees the planning of various components that make up each system. These components are fabricated during the *fabrication stage*. Next, these components are mounted together to form the systems during the *mounting stage*. The *commissioning stage* sees the first establishment of normal operations for each system. Lastly, the plant is assembled and put into operation during the *test approval stage* [76]. During every stage of the process, the produced documentation is certified and

approved by different stakeholders. The documents may contain sensitive information. For instance, the design of various systems or components is valuable and may have to be purchased before the usage and viewing is permitted. In addition, details like building plans or the device operation manuals are most important to the safe operation of power plants. For instance, a faulty operation manual may lead to an incorrect operation of the power plant which may lead to a calamity. Thus, the workflow and its data asset's integrity and confidentiality must be assured in each stage. For instance, an offer for a certain part must only be visible to the sender and recipient. In addition, each participant needs to be able to verify the integrity of the offer to check that it has not been tampered with. The focus of this thesis is the workflow and flow of data assets **between** organizations, the **audit trail** for the distributed workflow, and the **confidentiality** of data assets and workflows.

2.1.1. Stakeholders

Generally, a great variety of stakeholders may be involved when planning and certifying a power plant. To keep the scope focused only the *active* stakeholders are considered. Hence, the participants of the distributed multilateral workflow are the stakeholders. Each stage sees the participation of a distinct subset of the stakeholders. Figure 2.2 illustrates which stakeholder is involved in which states. In brief, the following active stakeholders were identified.

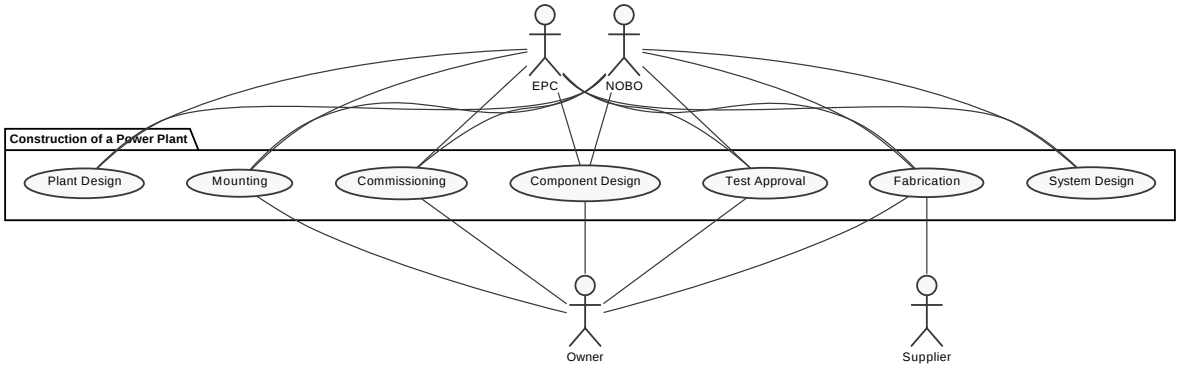

Figure 2.2.: Use case diagram which depicts which stakeholders are participating in which use case according to [76].

- Engineering, Procurement, and Construction contractor **EPC**: This is the entity that is responsible for all the activities from design, procurement, construction, commissioning and handover of the project to the Owner.
- **NOBO** is usually a governmental entity that is notified on each major step. It certifies the correct execution of certain steps and documents. In addition, it may

perform audits on site and creates audit reports.

- The **Owner** of the final product usually approves and accepts steps and will trigger rework steps if it is not satisfied with the results. The Owner will own the plant in the end and most likely operate the power plant themselves.
- The **Supplier** is an organization that produces one or multiple components for the EPC and they get their assignments directly from the EPC. Multiple suppliers may exist for a single component which leads to a possible competition for assignments among the suppliers

2.2. Fabrication Stage Use Case

Modeling the use case in its entirety with all workflows, processes, and critical assets is a lengthy and complex task. The scope of the complete use case (see Sec. 2) is deemed too big for this thesis. Hence, the need arises to construct a smaller use case. The work of Stahnke [76], Wimmer and Kasinathan [70, 71] and multiple discussions with supervisors shaped the construction though multiple iterations. The goal is to encompass most of the actors of the original use case, to include a distributed workflow where multiple actors participate in the execution, and to include confidential data assets and workflows.

2.2.1. Setting

The use case is named “fabrication stage” because it might happen during the fabrication stage during a power plant construction. The component that is produced is a special cabinet. The cabinet contains various electrical components that control an arbitrary system inside the power plant. Importantly, the cabinet is to be placed inside an area that may contain explosive reagents. For instance, the storage for liquid natural gas might be close by. Cabinets which are placed in explosion-risk areas are subject to high safety standards because any explosion could cause a calamity. Therefore, the cabinet is constantly flushed with an inert gas, e.g. nitrogen or argon. The inert gas and the elevated pressure inside the cabinet prevent the explosive gases from igniting. In addition, a special *pressure sensor* inside the cabinet constantly monitors the pressure difference between the inside of the cabinet and the outside. Whenever the pressure difference drops too low, explosive gas might enter the cabinet and ignite on the electrical components. Thus, special requirements apply to the pressure sensor and the cabinet. In conclusion, it is vital for the Owner that the quality, compliance and provenance with regulation and reliability of the cabinet and its components is assured. This assurance must be held up in manufacturing, installation, deployment and during regular maintenance checks.

However, this use case is not derived from an actual power plant construction and only serves as an example for this thesis.

2.2.2. Actors

The stakeholders **NOBO**, **EPC** and the **Owner** that have been described in Sec. 2.1.1 are involved in the fabrication stage use case as well. Further, there are multiple suppliers available. S1 and S2 can produce components for the EPC. **S1** is the *Supplier 1* which can produce the pressure sensor for the EPC. Also, **S2** is the *Supplier 2* which is not involved in the cabinet production but may be needed for other systems. S2 is important because its omission simplifies the design in Chapter 4 too much. For example, if only one supplier exists there is no need to protect the C_{PSO} (pressure sensor offer see Sec. 2.2.3) from unauthorized access.

2.2.3. Data Assets

The following data assets are involved in the fabrication stage use case. Further, the prefix C (**C**onfidential) states that this document is confidential and should only be shared between a subset of the actors and the prefix D (**D**ocument) states that this is a document, and no special precautions need to be taken. The data assets are listed in the order of their appearance:

1. D_{CDS} *Cabinet Design Specification*: It describes the requirements that apply to the cabinet. Importantly, the cabinet is to be placed in an explosive area. Hence, it is equipped with a pressure sensor that monitors the pressure difference between the inside and outside of the cabinet to assure excess pressure inside the cabinet.
2. C_{CO} *Cabinet Offer*: This is the offer that the EPC sends to the Owner whenever the Owner requests a particular cabinet. It may include confidential information such as price and delivery date which must only be shared with the EPC and the Owner.
3. D_{PSS}: *Pressure Sensor Specification*: This document describes the requirements for the pressure sensor. The specification depends on the design specification of the cabinet.
4. C_{PSO}: *Pressure Sensor Offer*: This document is sent by the *Supplier 1* to the EPC and it includes information regarding the price and delivery date of the pressure sensor. This information needs to stay confidential and should only be shared with the EPC and the Supplier 1.

5. D_{PSFS}: *Pressure Sensor Fact Sheet*: It describes the technical features of the final pressure sensor. For example, sensitivity, reaction time and precision.
6. D_{EAS}: *EPC Acceptance Sheet*: The sheet states that the EPC approves and accepts the pressure sensor. It might contain a test report and approval of the pressure sensor. It is handed to the Supplier 1 when the EPC accepts the pressure sensor delivery.
7. D_{CFS}: *Cabinet Fact Sheet*: It describes the technical features of the cabinet. For example, excess pressure inside, compliance with regulations or the list of functions.
8. D_{AR} *Audit Report*: The audit report is issued by NOBO and it states that the D_{CFS} satisfies the requirements of the D_{CDS}. Importantly, the Owner bases their decision whether it should accept the cabinet on this report. The audit report assures the Owner of the correct production of the cabinet.
9. D_{OAS}: *Owner Acceptance Sheet*: Represents the final approval by the Owner when being handed over the cabinet.

2.2.4. Confidentiality

Table 2.1 shows the access control list that applies to the data assets that are used in the use case. The following enumeration provides explanations for each data asset regarding the access control list. In addition, Figure 2.3 is a use case diagram for the fabrication stage use case. It shows that the overall workflow can be separated into three different sub use cases. First, the *Cabinet Order* happens between the EPC and the Owner. Second, the *Pressure Sensor Order* takes place between the EPC and S1. Finally, *Cabinet Evaluation* incorporates the interaction of NOBO and the EPC. In addition, the required data assets are also visible in the use case diagram.

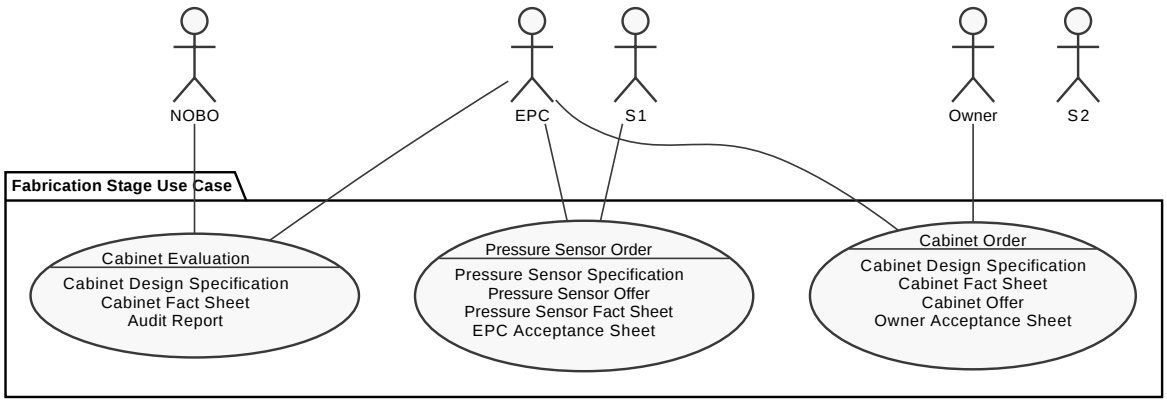

Figure 2.3.: Use case diagram for the fabrication stage use case which also shows which data assets are associated with which sub use case.

Objects \ Subjects		Owner		EPC		NOBO		S1		S2	
		R	W	R	W	R	W	R	W	R	W
D_{CDS}	Cabinet Design Specification	✓	✓	✓		✓					
C_{CO}	Cabinet Offer	✓		✓	✓						
D_{PSS}	Pressure Sensor Specification			✓	✓	✓		✓		✓	
C_{PSO}	Pressure Sensor Offer			✓				✓	✓		
D_{PSFS}	Pressure Sensor Fact Sheet			✓		✓		✓		✓	
D_{EAS}	EPC Acceptance Sheet	✓		✓	✓	✓					
D_{CFS}	Cabinet Fact Sheet	✓		✓	✓	✓					
D_{AR}	Audit Report	✓		✓		✓	✓				
D_{OAS}	Owner Acceptance Sheet	✓	✓	✓		✓					

Table 2.1.: Access control list for the data assets of the fabrication stage use case. A check mark in the R(W) column indicates read(write) access for the given stakeholder.

1. D_{CDS}: The cabinet design specification is created by the Owner. It needs to be read by the EPC, because it needs the information to decide what kind of conditions it offers for the production. In addition, NOBO needs to read it to learn about the cabinet's requirements to produce the audit report. However, the suppliers must not know about the cabinet design specification.

2. C_{CO}: The cabinet offer is created by the EPC. The Owner needs read access to decide whether to accept the cabinet offer or not. Importantly, NOBO, S1 and, S2 must not know about these conditions.

3. D_{PSS}: The pressure sensor specification is created by the EPC. S1 needs to read it to decide what kind of conditions it offers for the production of the sensor. Furthermore, S2 does not produce pressure sensors yet. However, the read access to the specification might enable them to make more informed decisions regarding new investments. Yet, the Owner does not need to know about the Suppliers of the EPC. Subsequently, the Owner does not need to know about the pressure sensor specification.

4. C_{PSO}: The pressure sensor offer is created by S1. The EPC needs to read it in order to decide whether to accept or decline the offer. However, NOBO, S2 and, the Owner must not know about this offer.

5. D_{PSFS}: The pressure sensor fact sheet is created by S1. The EPC needs access the fact sheet to create the cabinet fact sheet, because the pressure sensor is a part of it. However, S2 may also read the fact sheet. Perhaps such information leads to better informed business decisions.

6. D_{EAS}: The EPC acceptance sheet is created by the EPC. Only the Owner and NOBO may read the acceptance sheet. Importantly, S1 and S2 must not learn anything about the interaction between the EPC and the Owner.
7. D_{CFS}: The cabinet fact sheet is created by the EPC. NOBO needs to read it to produce the audit report. The Owner buys the cabinet and therefore should have access to it as well. However, the suppliers must not be involved.
8. D_{AR}: The audit report is created by NOBO. The EPC needs to read it in order to know whether the produced cabinet satisfies the requirements, and the Owner needs to read it in order to determine whether they accept the cabinet delivery. The supplier should not learn anything about this data asset.
9. D_{OAS}: The owner acceptance sheet is created by the Owner. Access to it need not be controlled because it only specifies that the owner accepted the delivery of a cabinet that was produced by the EPC. However, the suppliers are not involved in the cabinet production and must not know about it.

2.2.5. Workflow

Next, the data assets, actors and confidentiality are used to explain the entirety of the multilateral distributed business process. Figure 2.4 illustrates this business process. Importantly, each step is associated with an actor. In summary, the business process is distributively executed by the actors EPC, NOBO, S1, and the Owner. The following steps make up the workflow:

1. The **Owner** wants to order a cabinet that satisfies the specification D_{CDS}. Thus, the Owner request an offer for a cabinet from the EPC. They provide the cabinet design specification D_{CDS}.
2. The **EPC** sends the cabinet offer C_{CO} that is based on the cabinet design specification D_{CDS} that the Owner sent in the previous step.
3. The **Owner** receives the cabinet offer C_{CO} and accepts it. The EPC and the Owner are now committed to the specification and the offer. Importantly, this is a simplification of the negotiation and the case where the offer is not accepted is omitted. The rejection of an offer would either end the workflow prematurely or introduce more complexity with another EPC which is out of scope.
4. The **EPC** can produce most of the cabinet. However, the EPC needs an external supplier to produce a pressure sensor for the cabinet. Hence, the EPC produces the pressure sensor specification D_{PSS}. Employees of the EPC may extract the

specification through some unknown process from the cabinet design specification D_{CDS}. This process is out of scope.

5. The **EPC** requests an offer for a pressure sensor with the specification D_{PSS}. Thus, the EPC makes the specification available to the supplier S1.
6. **S1** sends the pressure sensor offer C_{PSO} to the **EPC**. It is based on the pressure sensor specification D_{PSS}.
7. The **EPC** receives the pressure sensor offer C_{PSO} and decides to accept the offer. S1 and the EPC are now committed to D_{PSS} and C_{PSO}. Importantly, the negotiation is a simplification. The reasons are the same as in step 3.
8. **S1** uses their internal processes to produce the pressure sensor. This process is out of scope. The pressure sensor fact sheet D_{PSFS} is a result of this process.
9. **S1** delivers the D_{PSFS} and the pressure sensor to the EPC.
10. The **EPC** tests the pressure sensor with their internal processes. These tests are executed to ensure that the pressure sensor satisfies its own fact sheet D_{PSFS}. These tests are not considered, and it is assumed that the *EPC Acceptance Sheet* D_{EAS} is issued upon successful execution these tests.
11. The **EPC** integrates the pressure sensor into the cabinet.
12. The **EPC** creates the cabinet fact sheet D_{CFS}. Importantly, the pressure sensor fact sheet D_{PSFS} is used when the D_{CFS} is created. This process is out of scope and may be by some of EPC's employees.
13. The **EPC** requests an evaluation of the D_{CFS} and D_{CDS}. The result states whether the fact sheet fulfills the requirements of the design specification. The request is directed to NOBO.
14. **NOBO** accepts the evaluation request. The case where NOBO declines is not considered for the reasons mentioned in step 3.
15. **NOBO** finishes the evaluation of D_{CFS} and D_{CDS} and subsequently produces the audit report D_{AR} which is sent to the EPC.
16. The **EPC** hands the cabinet and the audit report D_{AR} over to the **Owner**.
17. The **Owner** approves the cabinet with the issuance of the owner acceptance sheet D_{OAS}. The case where the Owner does not approve the cabinet is not considered. Such an extension would add multiple steps to the already lengthy use case. It is assumed that such additions only add complexity.

Figure 2.4.: Sequence diagram of the fabrication stage's workflow.

Lastly, Figure 2.3 showed that the overall workflow can be separated into 3 sub workflows. All of these bilateral workflows happen between the EPC and on other actor. Importantly, the EPC does not want to disclose the *Pressure Sensor Order* workflow to the Owner. Consequently, it can be said that this workflow is a confidential workflow.

2.3. Requirement Analysis

This section describes the methodology that was used to gather and structure the requirements of the fabrication stage use case in Sec. 2.3.1. Next, Sec. 2.3.2 lists the collected requirements.

2.3.1. Methodology

The requirements that are listed in Sec. 2.3.2 were gathered through an iterative process. The "construction and certification of a power plant" that Stahnke [76] worked on in her thesis was used as a reference. Next, the supervisors of the thesis take part in a project which is called Cyber Security for Europe. Some publicly published results of the project [71, 70] were also used as a reference. Then discussions and reviews by the supervisors resulted in the refinement of the gathered requirements.

2.3.2. Fabrication Stage Requirements

ID	Requirement	Description	Priority	Mandatory
FA01	Asset Creation	Every entity can create an asset or multiple assets.	High	Yes
FA02	Access Control on Assets	Entities can be given read and or write access to an asset.	High	Yes
FA03	Revoking Access	Entities can revoke access rights from assets.	High	Yes
FA04	Transferring Assets	The control over an asset can be transferred from one entity to another	Medium	No
FA05	Access Traceability	Read and write access must leave tamper proof evidence.	High	Yes
FA06	AC auditability	Changes to access control must leave tamper proof evidence.	High	Yes

Table 2.2.: Functional requirements for data assets of the fabrication stage use case.

Table. 2.3 lists and prioritizes the identified nonfunctional privacy and security requirements for the fabrication stage use case. Table 2.2 shows the functional requirements for data assets.

ID	Requirement	Description	Priority	Mandatory
SP01	Actor Authentication	Actors must be authenticated and have a verifiable identity.	High	Yes
SP02	Transaction Authentication	Actors must sign all their transactions with their digital signature.	High	Yes
SP03	Identity Management	Organizations can use their existing PKI to provide their employees with keys.	Medium	No
SP04	Non-Repudiation	Interactions with the system must leave tamper proof evidence.	High	Yes
SP05	Accountability	Interactions with the system can be traced to verify the compliance with established regulations, contracts or similar.	High	Yes
SP06	Data in Transit Confidentiality	Messages that contains sensitive data must run over secure channels.	High	Yes
SP07	Data at Rest Confidentiality	Sensitive data must be protected against unauthorized access.	High	Yes
SP08	Access Control	Access can be customized individually for each data asset.	High	Yes
SP09	Integrity	The integrity of data assets can be verified by entities with read access rights.	High	Yes
SP10	Organizations	Only the stakeholders mentioned in Fig. 2.1.1 can participate and be a part of the system.	High	Yes

Table 2.3.: Security and privacy requirements for the fabrication stage use case.

3. Background and Related Work

Following chapter sets up the foundation for this thesis. Section 3.1 will introduce the distributed ledger technology (DLT) field and clarify some important terminology. Next, Sec. 3.2 will introduce Hyperledger Fabric (HLF). It will introduce the most important aspects of it that are necessary for this thesis. Lastly, Sec. 3.3 briefly introduce zero knowledge proofs ZKP, Sec. 3.4 surveys the possibilities of homomorphic encryption (HE), Sec. 3.5 looks into secure multiparty computation (SMPC), and Sec. 3.6 will briefly look at the TEE.

3.1. Distributed Ledger Technology

According to Liu et al. "distributed ledger technology (DLT) is a general term that is used to describe technologies for the storage, distribution, and exchange of data between users over private or public distributed computer networks." Many different DLT concepts exist. Blockchain is one of them and will be the focus of this thesis. "Blockchain is a specific type of distributed ledger technology. It takes several records and puts them into blocks. Each block is chained to the next block, using a cryptographic signature" [79]. Figure 3.1 illustrates three blocks which are chained together. Figure 3.2 further details the contents of the block header.

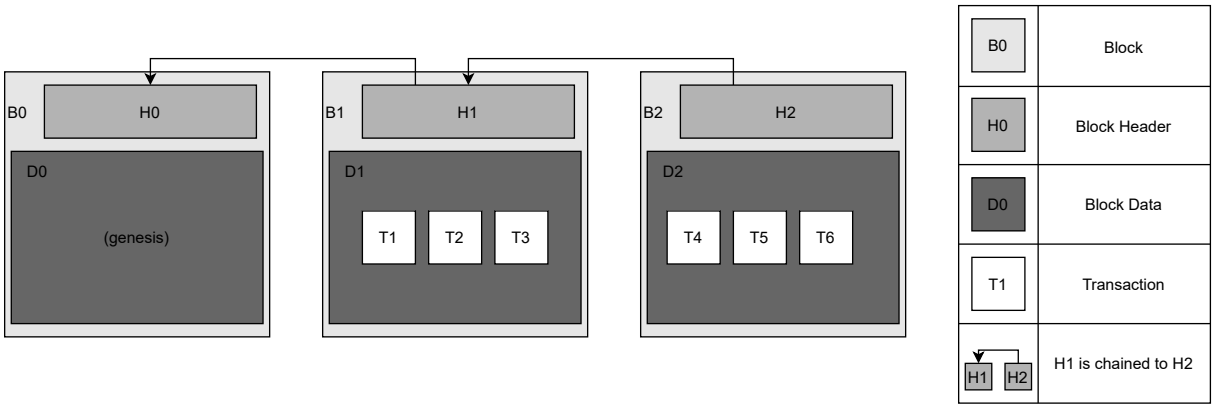

Figure 3.1.: Generic chain of blocks according to [3, 80].

Other concepts such as *blockDAG* and transaction directed acyclic graph (TDAG) exist [44] but they are not further considered. Bitcoin is the first blockchain and it was introduced in 2008 by Nakamoto.

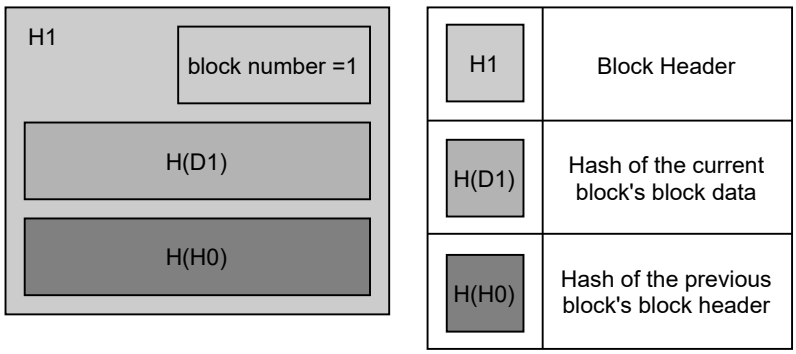

Figure 3.2.: Generic block header according to [3, 80]

Bitcoin introduced a new kind of money which is called decentralized money which is not dependent on institutions or governments [14]. Then Ethereum brought the decentralization of markets to the table in 2014. It generalized the concepts of Bitcoin and enabled the development of smart contracts. Smart contracts are decentralized applications which modify the state of the blockchain according to their own set of rules [8]. In short, the characteristics of blockchain such as decentralized consensus, data immutability, decentralization, transparency and auditability make it appealing for different industries. For example, Agbo and Mahmoud explored its usability for healthcare applications where protecting patients' health data is an important consideration [9]. In addition, blockchain is applied to public key infrastructure (PKI), traffic load balancing and power systems [34]. This section introduces general concepts of blockchain and surveys multiple blockchains that can execute smart contracts.

3.1.1. Permission model

The way in which nodes are added to a blockchain network varies. Some blockchain networks allow anyone to participate. Others are tailored towards classical business models and restrict access. This is known as the permission model. Two distinct models are distinguished. There is the permissionless and the permissioned blockchain.

- "**Permissionless** blockchain networks are decentralized ledger platforms open to anyone publishing blocks, without needing permission from any authority" [80]. The result is that every user of such platforms can publish or read blocks. There may exist malicious users that try to publish blocks in a way to subvert the system. For example, users might try to produce an alternative chain where a transaction has been erased. This could be an attempt to refund a payment. Section 3.1.2 will introduce consensus models which try to mitigate these kinds of threats.
- "A User must be authorized to act in a **permissioned** blockchain network. This includes read and write access control of the ledger and the issuance of transactions. Similarly, only authorized nodes can join such a network" [80]. The authorization for the users may be done by a centralized or decentralized authority.

These blockchains also employ a *consensus model*. Further, consensus models in permissioned blockchain networks can rely on the identity of nodes in the proof of identity (POI) model or on hardware requirements in proof of elapsed time (PoET) (see Sec. 3.1.2). Yaga et al. argue that consensus models in permissioned blockchain networks can be less computationally expensive than the proof of work (POW) consensus model which is used by the permissionless blockchain Bitcoin.

3.1.2. Consensus

An important part of each blockchain system is the *consensus* model. It is responsible for deciding which block gets added next to the blockchain [80]. There exist a multitude of approaches. Thus, only the most important algorithm POW is briefly mentioned. This *consensus* model is used by Bitcoin and it is a randomized protocol which implicitly selects a node based on a probabilistic scheme that is difficult to bias [25, 56]. The first participating node that solves the puzzle: $Hash(Data + X) = HashValue$ gets to publish the next block. The value of **X** must be guessed such that the **HashValue** is smaller than the *difficulty target* **D**. Further, the verification of a correct solution is easy, but the discovery of a solution is hard and can be adjusted depending on the size of **D**. If the *difficulty target's* value increases, then it get easier to find possible solutions because the amount of possible solutions increases. Likewise, a decrease of the *difficulty target* makes finding solutions harder because the probability of a hash being smaller than **D** decreases. Bitcoin calculates the next difficulty target D_{next} every 2016 blocks with the following formula: $D_{\text{next}} = (D_{\text{prev}} * 2016 * 10min)/(\text{time to mine last 2016 blocks})$. Thus, the difficulty is scaled such that a block is published on average once every ten minutes [57]. The solution is the "proof" that the node has done the "work". In brief, other nodes receiving the block can verify the solution and accept or reject it based on the result of the verification. More consensus models like proof of stake, proof of authority or round robin consensus [80, 66] exist. In permissioned blockchains where the participation to the blockchain network is limited other consensus models may be applicable which require a certain level of trust. For instance, the blockchains Quorum and Hyperledger Fabric (HLF) implement a consensus mechanism called Raft [60]. “Raft is a consensus algorithm for managing a replicated log” [60]. The consensus protocol can withstand a crash of individual nodes as longs as a majority remains active. Hence, it can be called crash fault tolerant (CFT) [3]. Importantly, CFT has the following property: “In response to a failure, the component changes to a state that permits other components to detect that a failure has occurred and the stops” according to Schneider [67]. Hence, it is suitable only for permissioned blockchains where it can be assumed that nodes do

Blockchain	Smart Contract Language
Hyperledger Fabric	Go, Java,Node.js,*
Ethereum	Solidity
R3 Corda	Kotlin,Java

Table 3.1.: Programming languages that are used to create smart contracts [9, 78].

not act maliciously. Furthermore, it cannot withstand so-called Byzantine faults. "In Byzantine model, a component can exhibit arbitrary and malicious behavior, perhaps involving collusion with other faulty components [47, 68]". However, Quorum implements a protocol called Istanbul Byzantine fault tolerant (IBFT) [6, 3] which is inspired by Castro and Liskov's work [30] on practical Byzantine fault tolerance (PBFT). Such systems can tolerate at most **F faulty** nodes when the number of nodes is **N**: $N = 3F + 1$ [30, 6].

3.1.3. Smart Contracts

"A **smart contract** is a collection of code and data [...] that is deployed using cryptographically signed transactions on the blockchain network [...]. The smart contract is executed by nodes within the blockchain network; all nodes that execute the smart contract must derive the same results from the execution, and the results of execution are recorded on the blockchain" [80]. The definition of smart contracts from Yaga et al. speaks of executing smart contracts on multiple nodes. Two different approaches for executing smart contracts are known in the context of blockchain. This section will introduce the widespread *order execute* architecture (see Sec. 3.1.3.1). Later in Sec. 3.2 the novel *execute order validate* architecture is introduced. Further, potential benefits of smart contracts include low contracting, enforcement, and compliance costs [79]. Their code can represent a multilateral transaction, typically in the context of business processes [80]. Hence, securing smart contracts that work with confidential data assets is one of the focuses of this thesis. Consequently, only blockchains that allow for custom smart contract execution will be considered and discussed. Two important properties of smart contracts are:

1. *Determinism*: Smart contracts may need to be deterministic if multiple nodes must produce the same results. Their results might have to be reproduced on multiple nodes to validate the transaction. However, there is the concept of an **oracle** [80]. It enables values that are only available outside the blockchain network to be accessible to nodes inside the network. In brief, every re-executing node can access the same value. Oracles are not needed for the fabrication stage use case (see Sec. 2.2). Hence, oracles are not considered further.

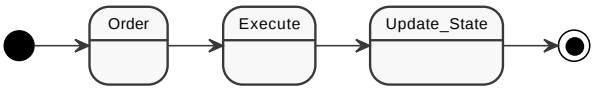

Figure 3.3.: Order execute architecture according to [11].

2. *Programming language*: Another important aspect of a smart contract is the programming language that it is written in. Table 3.1 illustrates the great variety when it comes to languages that can be used to develop smart contracts. While HLF takes an open approach, Ethereum uses domain specific languages, e.g. Solidity.

Next, Sec. 3.1.3.1 will describe the order execute architecture. It determines how the results of smart contract invocations are disseminated and agreed upon in a blockchain network.

3.1.3.1. Order Execution

It is used by all blockchains that are surveyed in Sec. 3.1.5 except HLF. As illustrated in Fig. 3.3 there exist 3 main steps.

1. *Order:* The blockchain network establishes an order using some consensus model.
2. *Execute:* Every node executes all transactions in the same order sequentially.
3. *Update:* The state is updated according to the results of step 2.

For example, Ethereum combines the proof of work (POW) consensus (see Sec. 3.1.2) and the *order execute* architecture as follows. First every node that participates in consensus gathers transactions that it deems valid. The transactions are usually executed first to decide if they are valid or not. Secondly every node tries to solve the POW puzzle. Thirdly the lucky node that solved the puzzle disseminates the solution and the block to other nodes. Lastly, all nodes check the validity of the puzzle and all of the transactions [11]. Androulaki et al. mentions the following limitations of this system:

1. Sequential execution: This limits the effective throughput and introduces vulnerabilities to malicious smart contracts that intentionally take a long time. This problem may be solved with cryptocurrencies and the demand of a fee for the execution of a smart contract function. In Ethereum this fee depends on the complexity of the function and is billed to the submitter of the transaction.
2. Non-deterministic code: The smart contracts must be deterministic. Hence, all nodes that execute the same smart contract function with the same inputs derive the same results. In conclusion, the smart contract's code has to be written in a

way that make it impossible to introduce nondeterminism. A solution to enforce determinism is the usage of a domain specific language (DSL) (see Sec. 3.1.5.1). However, this forces developers to learn new languages and therefore may slow the rate of adoption.

3. Confidentiality of execution: Which is known in this thesis as *evaluation privacy*. The requirement for re-execution to validate transaction requires the code to be public. Hence, the business logic in the smart contracts are shared with all participants. Consequently, no evaluation privacy is possible. Sections 3.3, 3.4, 3.5, and 3.6 investigate upcoming technologies which could be used to achieve evaluation privacy despite using the *execute order validate* architecture. But according to Androulaki et al. this is not yet viable in practice because of the considerable overhead.

3.1.4. Privacy and Confidentiality

Privacy and confidentiality are important properties for the use case (see Sec. 2.3.2). For example, the cabinet offer must stay confidential. It must only be accessible by the EPC and the Owner. Equally important the privacy of patients and their health records must be considered and the strict rules of the general data protection regulation (GDPR) must be applied [9]. However, the use case does not handle privacy related data. Further, smart contracts inherit undesirable properties from blockchains. As can be seen in Sec. 3.1.3.1 the *order execution* architecture requires that all nodes in the network execute all transaction. Therefore, all nodes need the smart contract code and the respective inputs. Hence, existing smart contract systems may lack confidentiality or privacy [83]. A survey of Khan and Nassar [45] assessed the ongoing development in this direction. They concluded that many solutions require trust in a third party [45]. HLF provides multiple ways to mitigate this undesirable property. Features such as channels, private data collections and endorsement policies are investigated in more detail in Sec. 3.2. Next, a brief overview of some notable blockchains follows in Sec. 3.1.5.

3.1.5. Overview

This section gives a brief summary of selected blockchain networks. The large number of distributed ledger platforms that have been developed recently [11] makes it impossible to compare each of them in this thesis. Hence, following sections will highlight the key distinguishing factors of each blockchain. In addition, a conclusion for each candidate attempts to give reasons as to why HLF is chosen over the reviewed technologies. Furthermore, HLF is described in detail in its own Sec. 3.2.

3.1.5.1. Ethereum

Ethereum attempts to be the platform on which all transaction-based concepts can be built upon. Its key feature is the ability for untrusted individuals to do transactions with each other [66]. Ethereum uses the *order execution* architecture for the smart contract execution. Further, Ethereum uses a domain specific language (DSL) named *Solidity* for smart contract development. This DSL may only allow the implementation of smart contracts that are deterministic which is a necessity for the *order execution* architecture (see Sec. 3.1.3.1). The downside is that developers have to learn a new programming language to implement smart contracts for Ethereum. In addition, Ethereum provides a built-in currency which is named *Ether* [78, 11]. Furthermore, this cryptocurrency must be used to pay for the transaction fee. Moreover, the fee depends on the amount of *gas* that is consumed by the transaction. To elaborate, the complexity of the smart contract determines the amount of gas. Specifically, an amount of gas is assigned to each low-level computation step [11]. This is converted to a price with the *gas price* which must be paid for by the transaction's submitter. Therefore, malicious denial of service (DOS) contracts (see Sec. 3.1.3.1) can be prevented with this cost. However, Grech et al. [41] identify vulnerabilities of Ethereum smart contracts which they call Out-of-Gas attacks. These attacks exploit the existence of the gas limit for the smart contract execution in Ethereum. Whenever the gas limit is reached an out-of-gas exception aborts the transaction. Consequently, attackers can force key functionality of smart contracts to run out of gas which is a permanent DOS for the contract. Also Grech et al. show how such vulnerabilities can be avoided [41]. Further, Ethereum uses a POW consensus algorithm (see Sec. 3.1.2). Also, Ethereum is a permissionless blockchain. However, permissioned versions of Ethereum exist. Quorum is the permissioned implementation of Ethereum (see Sec. 3.1.5.2) [4]. Also, Hyperledger Burrow is a permissioned blockchain which is derived from Ethereum [66]. Burrow does not implement any relevant features regarding confidentiality therefore it is not considered further [2].

Conclusion Ethereum is not suitable for the use case described in Sec. 2.3. The permissionless characteristic enables public access to the data and all smart contracts. This is in contradiction to the requirements set up in Tab. 2.3.

3.1.5.2. Quorum

ConsenSys Quorum is an open-source protocol layer that enables enterprises to leverage Ethereum for their private or public blockchain applications. On top of ConsenSys Quorum, you can integrate product modules from ConsenSys to build high-performance,

customizable applications [1]. Quorum is a fork of the Ethereum project that is developed by JP Morgan. However, it is a permissioned blockchain. Its main focus is the financial sector but has been developed for any type of industry [62]. Besides, Quorum facilitates the *order execute* architecture [11]. To illustrate, a few important changes to Ethereum are now highlighted:

- *Consensus*: The resource expensive POW consensus algorithm that Ethereum uses see Sec. 3.1.5.1 is now replaceable. Quorum provides a Raft and IBFT consensus implementation [16] (see Sec. 3.1.2).
- *Privacy*: Quorum allows transactions between a subset of the overall participants. These *private transactions* are ordered like the public transactions. Therefore, their existence is known by every participant. The transaction payload is replaced by a hash of the actual payload which is distributed off chain. Hence, only authorized participants may retrieve the payload and execute the transaction. The state of these *private transactions* is stored off chain. The hash of the *private transaction payload* is still written to the public ledger [6]. Likewise, Quorum can deploy private smart contracts that are only visible to authorized participants [16].

Conclusion The privacy features and the permissioned nature of Quorum are applicable to the requirements mentioned in Sec. 2.3. The brief overview of Quorum shows similarities with HLF. Polge et al. [62] reviewed different permissioned blockchains and summarized that HLF outperforms Quorum in the analyzed metrics *adoption* and *privacy*. The former indicates that HLF is adopted by more industrial use cases and the latter measured how restrictive a framework could be regarding the granularity that data and transactions can be shared [62]. Specifically, the granularity with which data and transactions can be controlled is important for the use case (see Sec. 2.3). In brief, Quorum has interesting features which can be facilitated to implement the fabrication stage use case. However, HLF is chosen because it is the focus of the thesis.

3.1.5.3. Corda

Corda is a blockchain for recording and processing financial agreements [66, 28]. It is open source and is a permissioned blockchain [62]. It records financial agreements and other arbitrary shared data between two or more identifiable parties. Furthermore, it takes the highly regulated environment into account by augmenting smart contracts with legal prose. It is a human-language document that is linked to the smart contract. The result is that the smart contract is legally enforceable [28, 42]. This is where it sees its main field of application [78]. Corda does not try to keep a global ledger. In cases where

transactions only involve a small subgroup the data is kept purely between this group [28]. This concept is similar to channels in HLF (see Sec. 3.2.1.7). According to Brown et al. the core concepts of Corda are [28]:

- *State object*: represents an agreement between multiple parties. The agreement consists of two parts. Human readable *Legal Prose* and machine-readable *Contract Code*. Only privileged parties can access the state object of specific agreements between parties [66].
- *Transactions*: The actions that transitions a state object though its lifecycle.
- *Business Flow*: Enabling parties to coordinate actions without a central controller.

Conclusion Corda was built for the explicit purpose of recording and enforcing business agreements between financial institutions. Especially the addition of human readable *Legal Prose* is to be noted. However, this addition is no requirement for the use case discussed in Sec. 2.3. Also, Polge et al. compared HLF with Corda [62]. They analyzed multiple metrics and among those were privacy and adoption both of which are important for the use case. Moreover, the authors concluded that HLF outperforms Corda in these metrics. The ranking for privacy depended on the granularity by which the visibility of data and transactions could be controlled [62]. In brief, Corda has interesting features which could be used to implement the fabrication stage use case. However, Corda's focus on financial institutions and this thesis's focus on HLF lead to Corda not being investigated further.

3.1.5.4. Ekiden

Ekiden is a combination of blockchain and a trusted execution environment (TEE) which is a tamper resistant processing environment that allows for execution of confidential code on an untrusted node while preserving the privacy of inputs [65] (see Sec. 3.6). Ekiden employs the trusted execution environment from Intel named software guard extension (SGX). Ekiden can be combined with any blockchain system according to [83] it can therefore neither be classified as permissioned nor permissionless. Ekiden attempts to address the shortcomings of blockchain regarding confidentiality and performance with a combination of TEE and blockchain. Blockchains like Ethereum verify the correct execution of smart contracts through re-execution on all nodes (see Sec. 3.1.3.1). This results in every node knowing every input of every smart contract invocation. Consequently, the inputs cannot be confidential because every node needs them to replicate the smart contract invocations. Ekiden combines TEE and blockchain to achieve confidential inputs

while still enabling other participants to verify the correct computation. Ekiden shows that a combination of TEE and blockchain can achieve a decentralized system that can compute on sensitive data while guaranteeing computational integrity and confidentiality of inputs [83].

Ekiden separates the *computation* from *consensus*. The processing of private data is delegated to *compute nodes*. The correct execution is attested on the blockchain. This attest is derived from the TEE. So called *consensus nodes* which do not need to run on trusted hardware maintain the blockchain [83]. The outsourcing of computation to TEE allows for minimal performance overhead. Importantly, this removes the need for recomputation to verify the results of smart contract invocations meaning that the network can spend less resources on the replication and verification of smart contract invocations. While this is possible with SGX the reliance on it means that Ekiden is not a fully decentralized solution [29].

Further, the actual smart contract code must be published to the blockchain. This results in the smart contract code being public. Meaning that confidential smart contracts cannot be implemented with Ekiden. More detailed descriptions of the system can be found in [83].

Conclusion Ekiden is an interesting approach that combines TEE and DLT. The reliance on the third party begs the question if Ekiden can still be decentralized. Bünz et al. state that Ekiden is in fact not fully decentralized [29]. Considering the reliance on SGX it can be concluded that Ekiden is not picked over HLF.

3.1.6. Conclusion

This section briefly introduced exiting DLTs. Ethereum was reviewed in Sec. 3.1.5.1, Quorum in Sec. 3.1.5.2, Corda in Sec. 3.1.5.3 and finally Ekiden in Sec. 3.1.5.4. An overview of all existing DLTs to find the best candidate is out of the scope of the thesis. Androulaki et al. argue that the number of different upcoming developments is too large [11]. Regardless, it is important to show the diversity of the different technologies. Also, each approach offered unique characteristics which shows the diversity in the field. In brief, the DLTs that have been reviewed in this section are interesting candidates for the fabrications stage use case. A reason why HLF is chosen over them is to reduce the scope of the thesis. Consequently, Sec. 3.2 will introduce HLF's features in detail.

3.2. Hyperledger Fabric

Hyperledger Fabric is an enterprise-grade *permissioned* blockchain. HLF has been designed for enterprise use form the start. It is established under the Linux Foundation. Hence, it is open source and subject to open governance resulting in a community which includes 35 organizations and 200 developers. Furthermore, HLF is not dependent on a cryptocurrency [11]. Equally important it is the first blockchain that runs distributed applications that are written in standard, general-purpose languages [3]. This means that more developers already have preexisting skills that can be reused for the development of HLF applications. The permissioned characteristic allows HLF to use a different approach to consensus. Fabric has no fixed consensus algorithm and simplifies the replacement of consensus models [11]. The aforementioned smart-contract executing blockchains (see Sec. 3.1.5) all use the *order execute* architecture (see Sec. 3.1.3.1). However, HLF introduces the *execute order validate* architecture which is discussed in Sec. 3.2.3.1. This Sec. will introduce the most important aspects of the HLF blockchain. Section 3.2.1 will introduce the basic components that are present in HLF, Sec. 3.2.2 explains how smart contracts are executed, Sec. 3.2.3 will introduce the stages that each transaction passes through, and Sec. 3.2.4 will introduce the private data collection (PDC) which is used to store confidential data with HLF.

3.2.1. Hyperledger Fabric Basics

Hyperledger Fabric (HLF) is made up of a multitude of different components. Each of these components has different responsibilities. This Sec. introduces the peer in Sec. 3.2.1.1 and the ordering service in Sec. 3.2.1.2. Further, the difference between chaincodes and smart contracts will be explained in Sec. 3.2.1.3, and Sec. 3.2.1.4 shows how chaincodes are deployed in HLF. Then, the structure of the ledger will be shown in Sec. 3.2.1.5 and the membership service provider (MSP) will be explained in Sec. 3.2.1.6. Finally, the channel concept will be introduced in Sec. 3.2.1.7.

3.2.1.1. Hyperledger Peer

The blockchains main participants are the *peer nodes* which are responsible for hosting instances of the ledger (see Sec. 3.2.1.5) and chaincodes (or *smart contracts* see Sec. 3.2.1.3). Hence, peers are responsible for the *execution* and the *validation* of smart contract invocations [11]. Figure 3.4 illustrates a network with 3 peers one smart contract and one ledger. It shows that the smart contract S1 is deployed on all peers. In addition, L1 which is the ledger is hosted on all peers.

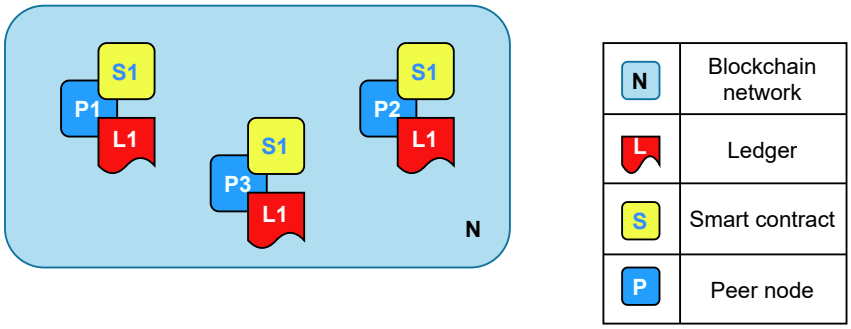

Figure 3.4.: Illustration of a simplified blockchain network according to [3].

3.2.1.2. Ordering Service

The ordering service is responsible for establishing the *total order* of all transactions. It consists of one or more ordering service nodes (OSN). Importantly, the *ordering service* neither participates in the execution nor the validation of transactions [11]. However, it is possible to deploy a *peer* and an OSN on the same physical machine [3].

Next, the ordering service manages channels, which are separate blockchains running within HLF (see Sec. 3.2.1.7). To elaborate, the ordering service manages a group of organizations called the *consortium*. It contains organizations that are eligible to create channels [3]. Androulaki et al. lists the technical details for this process [11].

Further, the ordering service has multiple implementations. As of HLF version 2.2 the Raft implementation is recommended in the documentation [3]. In brief, Raft keeps a *log* consistent across multiple nodes. The *log* is considered consistent if most nodes agree on the entries and their order. The *consenter set* is the set of OSN that actively participate in the consensus algorithm [3]. For more details surrounding the Raft protocol see [60].

In addition, a *Solo* ordering service is available. It consists of a single OSN and is intended only for testing. Also, HLF implements an ordering service that is based on *Kafka* which is an open source, distributed publish-subscribe messaging system [58]. However, it was deprecated in v2.x [3]. In summary, the recommended implementation is CFT. However, Androulaki et al. introduces the possibility for an Byzantine-fault tolerant (BFT) ordering service [11]. Section 3.2.3 will detail the ordering service's role in the transaction flow. Details for BFT and CFT can be found in Sec. 3.1.2.

3.2.1.3. Chaincode and Smart Contracts

When reading about HLF the terms smart contract and chaincode are often used interchangeably [3, 11]. However, they can mean different things depending on the context. Figure 3.5 shows a simple example of a *sensor chaincode* which contains two smart contracts the *temperature sensor contract* and the *pressure sensor contract*. In short, multiple

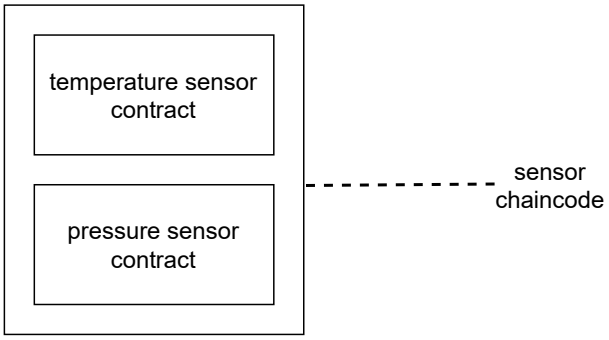

Figure 3.5.: Relation between chaincode and smart contract according to [3].

smart contracts can be a part of the same chaincode. In brief, *chaincode* is used when referring to the package of smart contracts, or when talking about deployment (see Sec. 3.2.1.4). Chaincodes are deployed to channels which will be introduced in Sec. 3.2.1.7. Furthermore, chaincodes contain the programs that capture the actual business logic of the desired application. Chaincodes interact with the ledger to store business objects (see Sec. 3.2.1.5) and they offer functions that can be executed in transactions [46]. The term smart contract is used when talking about the business logic. There are two distinct types of chaincodes:

1. *Application chaincode:* These are the chaincodes that contain the application logic and they may be written by untrusted developers [11].
2. *System chaincode:* These are the special chaincodes that exist to manage the interactions with the blockchain features. For instance, the lifecycle system chaincode (LSCC) manages how chaincodes are deployed to channels, the endorsement system chaincode (ESCC) runs on peers to sign transaction responses and the validation system chaincode (VSCC) validates transactions [3, 46]. The article of Androulaki et al. provides a more detailed explanation of *system chaincodes* [11].

Now HLF supports a multitude of programming languages in which chaincodes can be written in. These include Go, Java and Node [78, 11, 62]. However, other languages can be used because HLF loosely coupled the chaincode execution with the peer. Each *application chaincode* runs in its own Docker container environment. The interaction between chaincode and peer is implemented with the remote procedure call (gRPC) protocol. It is an open source remote procedure call framework that enable communications between client and server applications transparently [6]. Consequently, if a language supports gRPC than it can be used to write chaincode in theory. Next, Sec. 3.2.1.4 will introduce the chaincode lifecycle which outlines how chaincodes are deployed and updated.

3.2.1.4. Chaincode Lifecycle

The chaincode lifecycle allows multiple organizations to agree on how a chaincode can be used. This process is not static. It is governed by the *lifecycle endorsement policy*. Each channel (see Sec. 3.2.1.7) can have a different policy. The policy states who must approve a chaincode before it can be used. This policy works just like an endorsement policy (see Sec. 3.2.2). There are 4 steps that need to be completed if a chaincode is to be installed on a channel:

1. **Package chaincode**: The program code that is used for the chaincode has to be packaged. HLF uses the tar file format to package the chaincode. This step is done by the application's developer.
2. **Install chaincode**: Every organization that wants to install the chaincode needs to have the tar package. Importantly, the package is not saved on the blockchain. It is only shared deliberately by the developer. The install process returns a *hash* which is used to identify the packaged chaincode.
3. **Approve chaincode definition**: Every organization that wants to use the chaincode must approve the chaincode's definition on the channel. The definition contains the name, endorsement policy, and collection configuration among other important things. Importantly, it is not necessary to install the chaincode to approve its definition. Hence, it is possible for organizations to approve chaincodes without them knowing the chaincode's source code. The organization only needs to have the chaincode definition, which can be shared out of band, to approve it.
4. **Commit chaincode definition**: Lastly, once enough approvals are present and the *chaincode lifecycle policy* is satisfied **one** organization can commit the chaincode definition to the channel. Consequently, it is possible to submit transactions for the new chaincode.

The process that applies to the installation of chaincodes applies to the upgrade of chaincodes as well. The same 4 steps must be followed. A special case applies in the case of endorsement policy updates. In this case the actual binary might not change. Then it is sufficient to approve the new chaincode definition. Which means the steps 1 and 2 can be omitted.

Consequently, HLF enables the execution of confidential business logic while still benefiting from the properties of the ledger (see Sec. 3.2.1.5). For instance, Figure 3.6 shows 3 organizations participating in the same channel. The *lifecycle endorsement policy* in this case is **OutOf**(1, EPCMSP.admin). This means that an approval of the EPC's admin is enough to enable the commit of the chaincode definition. The chaincode S1 is deployed

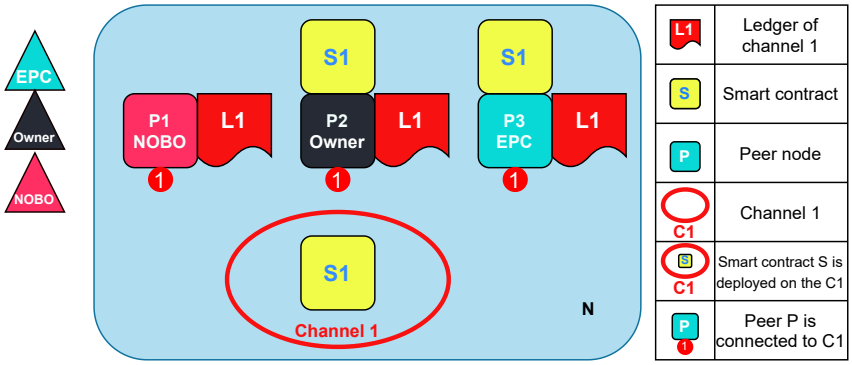

Figure 3.6.: Example of a confidential smart contract.

on the channel 1. In addition, it is only installed on P3 and P2. P1 which belongs to NOBO does not have access to the S1's code. Hence, it can be concluded that S1 can be used on channel 1 even though it is not shared with NOBO. Thus, confidential smart contracts are possible with HLF.

3.2.1.5. Ledger

The ledger is the component that is responsible for storing the information about the complete transaction history and the current values of the business objects that are managed by the smart contracts. It is hosted on peer nodes and consists of two components. The first component is the *transaction block store* or *blockchain*. The second component is the *peer transaction manager* or *world state* [3, 11]. Figure 3.7 illustrates the two components. The *world state* represents the current values of the business object and is implemented as a key-value store. Smart contracts can use functions like *putState* and *getState* to interact with it. The *blockchain* is the transaction log that incorporates all transactions that have been disseminated in blocks [3, 11].

Importantly, the chaincodes that are deployed on individual peers only have access to *their* specific world state. To elaborate, two smart contracts that are deployed with in same chaincode can access the *same* world state. Likewise, two smart contracts that are deployed with two different chaincodes *cannot* access the other's world state directly. This

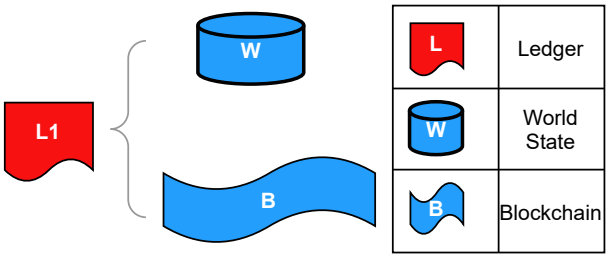

Figure 3.7.: Illustration of the ledger according to [3].

separation is achieved with so-called *chaincode namespaces*. If access across the borders of chaincodes should be required than *cross chaincode access* is needed. HLF provides the *invokeChaincode* API see. [3] which allows the invocation of different chaincodes that are installed on the *same* peer but not necessarily on the same *channel*.

3.2.1.6. Membership Service Provider

The membership service provider (MSP) maintains the identities of all nodes in the system. That means that all ordering service nodes, peers and clients have identities. The permissioned nature of HLF means that all interactions among nodes are authenticated [11]. The MSP is an abstraction meaning there are multiple different implementations possible. There exists a default implementation which handles standard PKI methods based on digital signatures. Further, organizations can use their existing PKI with HLF. In addition, an implementation is available which relies on anonymous credentials for authorizing clients to invoke transactions without revealing their identities [3] (see. Sec. 3.3.3).

3.2.1.7. Channels

It is possible to support multiple blockchains within the same HLF network. Each of these blockchains is called a *channel*. It allows a subset of organizations to exclusively communicate and maintain a ledger together. Every transaction that takes place within this channel is only visible to the channel members and the responsible ordering service. Each channel can be configured independently through its own channel configuration (CC). The configuration system chaincode (CSCC) is responsible for the managing the CC. Importantly, the CC defines policies for the channel which govern what specific organizations are allowed to do. For instance, the *Writers* policy defines who can submit transactions [3]. Figure 3.8 shows a minimal example of a very simple blockchain network with one channel, two organizations R1 and R2, and two applications. Each organization has their own certificate authority which is used to create identities for each of their clients and peers. In addition, they have deployed a smart contract S1 on each of their peers.

3.2.2. Endorsement Policies

Endorsement policies are defined for each chaincode and they specify the set of organizations that must execute a chaincode method and endorse the result from this execution. Approval or endorsement is given by executing the proposed transaction and subsequently signing the inputs and results [3, 11]. For instance, given three organizations EPC, NOBO and Owner. An example endorsement policy can take the following form:

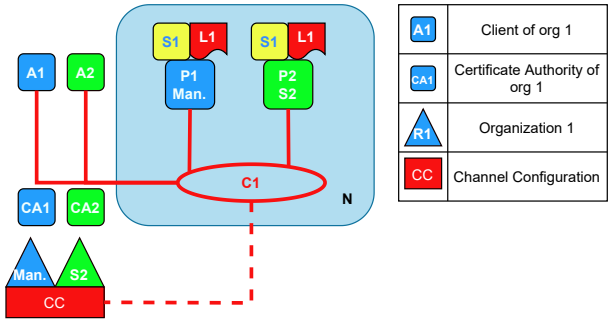

Figure 3.8.: Illustration of a simplified blockchain network with one channel [3].

- **AND**(EPC, NOBO): Which would mean that both the EPC and NOBO have to endorse a transaction for it to be deemed valid.
- **OR**(EPC, NOBO): Means that one endorsement from either the EPC or NOBO would be sufficient.
- **OutOf**(**N**, EPC, NOBO, Owner): States that **N** endorsements are sufficient regardless of which one is missing. For instance, if N is 2 then three possible combinations of endorsements are valid:
  1. EPC and Owner
  2. EPC and NOBO
  3. Owner and NOBO

This is a key distinction that HLF has in comparison to the surveyed DLTs from Sec. 3.1.5. The novel architecture *execute order validate* architecture (see Sec. 3.2.3.1) in combination with endorsement policies allows peers that are not executing chaincode to still validate and apply the results of the transactions to their copy of the ledger. To sum up, non-executing peers *trust* the endorsements of executing peers.

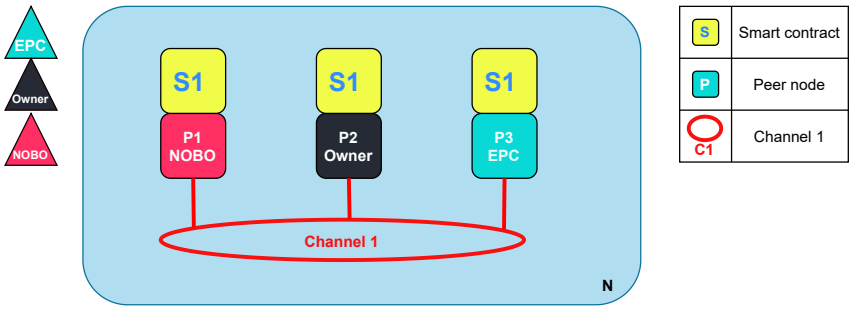

Figure 3.9.: Hyperledger Fabric network for the endorsement policy example.

Figure 3.9 shows a small example network that involves three actors from the fabrication stage use case (see Sec. 2.2). The EPC, NOBO, and the Owner participate in the

network. Only one channel with the name channel 1 exists and one smart contract named S1 is deployed on that channel. Further, this example illustrates three different endorsement policies that can be set up for the smart contract S1. Table 3.2 shows 3 different endorsement policies. Figure 3.10 illustrates different transaction proposals and their endorsements. Importantly, the color of the table cells illustrate if the endorsement policy is satisfied with the existing endorsements.

Policy Nr	Endorsement Policy
P1	**OutOf**(1, NOBOMSP.peer)
P2	**AND**(NOBOMSP.peer, EPCMSP.peer)
P3	**AND**(NOBOMSP.peer, **OR**(EPCMSP.peer, OwnerMSP.peer))

Table 3.2.: Example endorsement policies for the small example.

P1 The endorsement policy states that it requires the endorsement of NOBO's peer. The examples where EPC's peer or the Owner's peer give an endorsement don't satisfy the policy. Therefore, if NOBO's peer is unavailable than no transactions of S1 can be endorsed.

P2 The policy states that NOBO's and the EPC's peer must endorse transactions. Hence, should either peer be unavailable than no transactions of S1 can be endorsed.

P3 This policy is a nested policy. First is always requires an endorsement of NOBO's peer. However, the second endorsement can come from either EPC's or Owner's peer. Hence, should NOBO's peer be unavailable then no transactions can be endorsed. Further, if NOBO's peer is available than either the EPC's or the Owner's peer must be available.

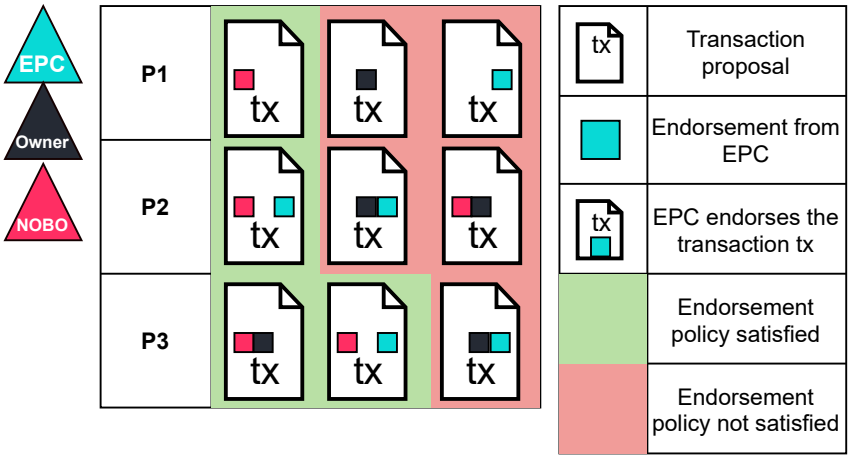

Figure 3.10.: Endorsement examples for Tab. 3.2.

Another essential point is that not every peer that participates in the same channel must have all chaincodes that have been deployed to that channel installed. The outcome is that the endorsement policy dictates where chaincodes must be installed. For instance, policy **P1** only needs endorsements from NOBO's peer. Hence, it would be sufficient to install the chaincode on its peer(s). It is impossible to create valid transactions if NOBO's peer does not have S1 installed.

Furthermore, this results in the ability of HLF to enable fine-grained access control to the smart contracts themselves. For instance, policy **P2** does not need endorsements from the Owner's peer. Hence, it is possible to refuse the Owner access to the smart contract's code.

Also, HLF provides 3 different levels of endorsement policies [3]. The previously mentioned examples only used *chaincode level endorsement policies* which apply to the whole chaincode and all smart contracts inside that chaincode.

1. **Chaincode level endorsement policies**: It is the default policy that applies if no other endorsement policy applies. The chaincode level endorsement policies are agreed to by channel members when they approve a chaincode definition for their organization according to the *LifecycleEndorsement policy* (see Sec. 3.2.1.4). Importantly, an alteration to the chaincode level endorsement policy requires an upgrade to the chaincode. Hence, it may require additional approval from some channel members.
2. **Collection level endorsement policies**: These policies apply to the special private data collection (PDC) which stores data off chain and only leaves a hash of the private data on the actual blockchain. To elaborate, these policies apply whenever a chaincode function uses a key of a PDC. If this is the case it overrides the chaincode level endorsement policy [3]. Importantly, this policy can be *more* or *less* restrictive than the chaincode level endorsement policy. Also, this policy is part of the chaincode definition and requires a chaincode update to be altered just like the chaincode level endorsement policy [46]. That aside, there is a separate policy which states who can store the collection (see. Sec. 3.2.4).
3. **Key level endorsement policies**: They apply to individual entries in the world state and they are also known as *state-based endorsement policies* [12, 46]. For example, the EPC writes a value to a key. But it wants to make sure that it is required to endorse changes to this specific key. Hence, it can add a key level endorsement policy to that specific key. The modification is part of the read-write set of a regular transaction. It can be set up for private data collections and regular world state keys. These restrictions can be less or more restrictive than chaincode or

collection level endorsement policies. If multiple keys are modified in a transaction all applying policies must be satisfied to validate the transaction [3].

The validation system chaincode (VSCC) is responsible for the validation of the endorsement policies. Furthermore, Androulaki et al. provides an overview of the intricacies of endorsements and presents security considerations and security models for endorsement policies [12, 11].

3.2.3. Transactions

Transactions are created when a chaincode function is invoked by clients [3]. This Sec. 3.2.3.1 will introduce the *execute-order-validate* architecture that HLF uses to execute chaincode functions. Next, Sec. 3.2.3.2 will discuss the application of said architecture with HLF components. Finally, Sec. 3.2.3.3 briefly discusses whether HLF transactions fulfill the ACID properties.

3.2.3.1. Execute Order Validate

Hyperledger fabric "follows a novel *execute-order-validate* architecture for distributed code execution of untrusted code in an untrusted environment" [11]. Importantly, all other mentioned blockchains use the *order-execute* architecture (see Sec. 3.1.3.1). The new architecture removes the need for sequential execution, non-deterministic code, and it enables confidential smart contract code. The architecture splits the transaction flow into four phases. Figure 3.11 illustrates these phases with a state diagram. In brief, the execution of transactions happens in the *execution phase*. In addition, the correctness of the transaction is checked. Next, the transactions are ordered in the *ordering phase*. Importantly, the semantics of the transactions are not considered. Then, the application-specific trust assumptions are validated in the *validation phase*. Lastly, the changes are applied to the world state in the *update state phase* [11].

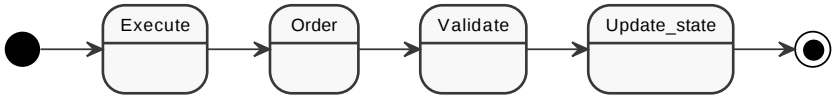

Figure 3.11.: Execute order validate architecture according to [11].

Therefore, each HLF chaincode consists of two distinct parts. Firstly, the smart contracts which represent the business logic. Secondly, the *endorsement policy* which implements the application-specific trust assumptions. It is the central subject in the validation phase. It states which organizations have to endorse which transaction (see. Sec. 3.2.2).

Next, the phases are now explained in more detail. A more detailed description can be found in [11].

- **Execution phase:** The client that intends to execute a transaction forms a *transaction proposal*. This proposal includes any inputs for the chaincode function, the identity of the client, and it specifies which function to call. Next, the client sends their *transaction proposal* to all necessary endorsing peers. This set of endorsing peers may be a subset of all peers on the channel. The *endorsement policy* defines possible subsets. Importantly, endorsing peers must have the chaincode installed to endorse transactions.

  The endorsing peers receive the *transaction proposal* and check if the submitter satisfies the *Writers* policy which authorizes them to submit transactions on the channel. Next, the endorsing peer simulates the transaction proposal by executing the chaincode function with the specified inputs. Importantly, the results of the execution are not are not stored in the world state. On the contrary, the endorsing peers create a signed proposal response that includes a *readset* which lists all read keys of the world state, a *writeset* which lists all written keys of the world state, the response value, and the endorsing peer's signature. Then, the signed proposal response is sent back to the client

  Next, the client collects the necessary amount of *proposal responses*. In addition, the responses are inspected by the client to check if the responses all produced the same results. Also, HLF provides a software development kit (SDK) for Node.js and Java which transparently executes this process for the client [11, 3].

- **Ordering phase:** The client assembles all necessary *proposal responses* into the *transaction message* which it submits to the *ordering service*. However, if the chaincode function did not write the ledger then a submission *can* be omitted, and any further phases are omitted. Transactions that are not submitted may also be called *queries*. Importantly, queries won't be added to the blockchain and are therefore not reliably auditable. Regardless, the *ordering service* establishes a total order of all submitted transactions for each channel. The ordering service does not need to inspect the transactions in detail to establish the order. In addition, it packs multiple transactions into blocks. The result is a sequence of blocks where each block is linked to its predecessor with their hash [11]. The ordering service initiates the dissemination of the blocks to the peers. The detailed dissemination process is described in the documentation [3]. Importantly, there is no time limit when all peers have the same view of the blockchain. In addition, the ordering service has

the **Validity** property which states "If a correct client invokes broadcast(tx), then every correct peer eventually delivers a block B that includes tx, with some sequence number" [11]. This means that every correct peer eventually receives a transaction once it has been sent to the ordering by a correct client.

- **Validation phase:** Each peer that receives a new block executes the *validation phase*. It consists of the following three steps which are executed *sequentially* according to [11]:
  - *Endorsement policy evaluation*: This step evaluates in parallel all transactions. The VSCC checks if enough endorsements are attached to the transactions so that the respective *endorsement policies* are satisfied. In brief, every transaction in the block is marked valid or invalid depending on the outcome of the evaluation.
  - *Read-Write conflict check*: This step is done sequentially for all transactions in order. The read set of each transaction is compared with the current state of the ledger of that peer. To elaborate, each key in the world state has a version number. This number is a part of the read set. For example, a transaction's read-set contains the key *color* and the version *1*. However, the world sate of the peer contains the key *color* and the version *2*. Hence, the transaction is marked invalid and will be ignored in subsequent phases. Also, the transactions are also marked as invalid if the keys in their read-set are a part of the write-set of previous transactions of the *same* block. In brief, "this checks whether a transaction conflicts with any preceding transaction (within the block or earlier)" [11]. This is also known as a multi-version concurrency check (MVCC) [3].
  - *Ledger update phase*: The result of the prior phases and the block are appended to the blockchain (*transaction log*). The results of the prior evaluations are persisted to facilitate the reconstruction of the world state according to [11]. Furthermore, the write-sets of the transactions are applied to the world state of the peer.

In conclusion, the novel *execute order validate* architecture solves the following shortcomings of the *order execute* architecture.

1. Sequential execution: The execution of transactions does not need to happen sequentially. Multiple clients can invoke multiple transactions on multiple peers in parallel. However, the *read-write conflict check* in the *validation phase* has to be se-

quential. This is beneficial to the throughput and latency of transactions according to Androulaki et al. [11].

2. Non-deterministic code: The chaincode functions can be non-deterministic because peers that don't endorse don't execute them. Hence, only the endorsing peers must produce the same results so that the endorsement policy can be satisfied. For instance, Table 3.9 and Figure 3.2 show the policy **1**. The satisfaction of this policy only needs **one** endorsement from NOBO's peer. Hence, the smart contract can be non-deterministic. For instance, a client may receive multiple different proposal responses from NOBO's endorsing peer if the smart contract is non-deterministic. However, one endorsement is enough and the subsequent steps in the transaction flow (see Sec. 3.2.3.2) can successfully be executed. Hence, non deterministic code can be used in HLF.
3. Confidentiality of execution: Only the endorsing peers must execute a specific chaincode. Hence, the chaincode must only be installed on this set of peers. The rest of the peers on the channel need not and cannot access the chaincode code through HLF. Therefore, the chaincodes code can be confidential and must only be shared with the peers that are necessary to satisfy the endorsement policy. For instance, Sec. 3.2.1.4 includes an example for this. Consequently, confidential business logic or business processes can be implemented with *confidential smart contracts* with HLF.

However, the *execute order validate* paradigm also introduces some new challenges. Importantly, endorsement policies are very important. For they dictate which nodes must execute transactions. Hence, weak endorsement policies may lead to incorrectly executed transactions or strong endorsement policies may block the correct validation of new transactions because some nodes may be unavailable. Further, confidentiality of execution also has downsides. For instance, peers that don't execute transactions have to order transactions from smart contracts which these peers don't know anything about. Chapter 6 will further go into detail about the challenges of HLF.

3.2.3.2. Transaction Flow

Figure 3.12 illustrates a simplified example to further expand the description of the *execute order validate* architecture. Four peers are illustrated. Three endorsing peers EP_1, EP_2 and EP_3, and one committing peer CP_1 are shown. CP_1 does not have the chaincode of the respective transaction installed. Furthermore, all peers are a part of the same channel. Also, an arbitrary ordering service is also available. In addition, the transaction needs 3 endorsements from all existing endorsing peers.

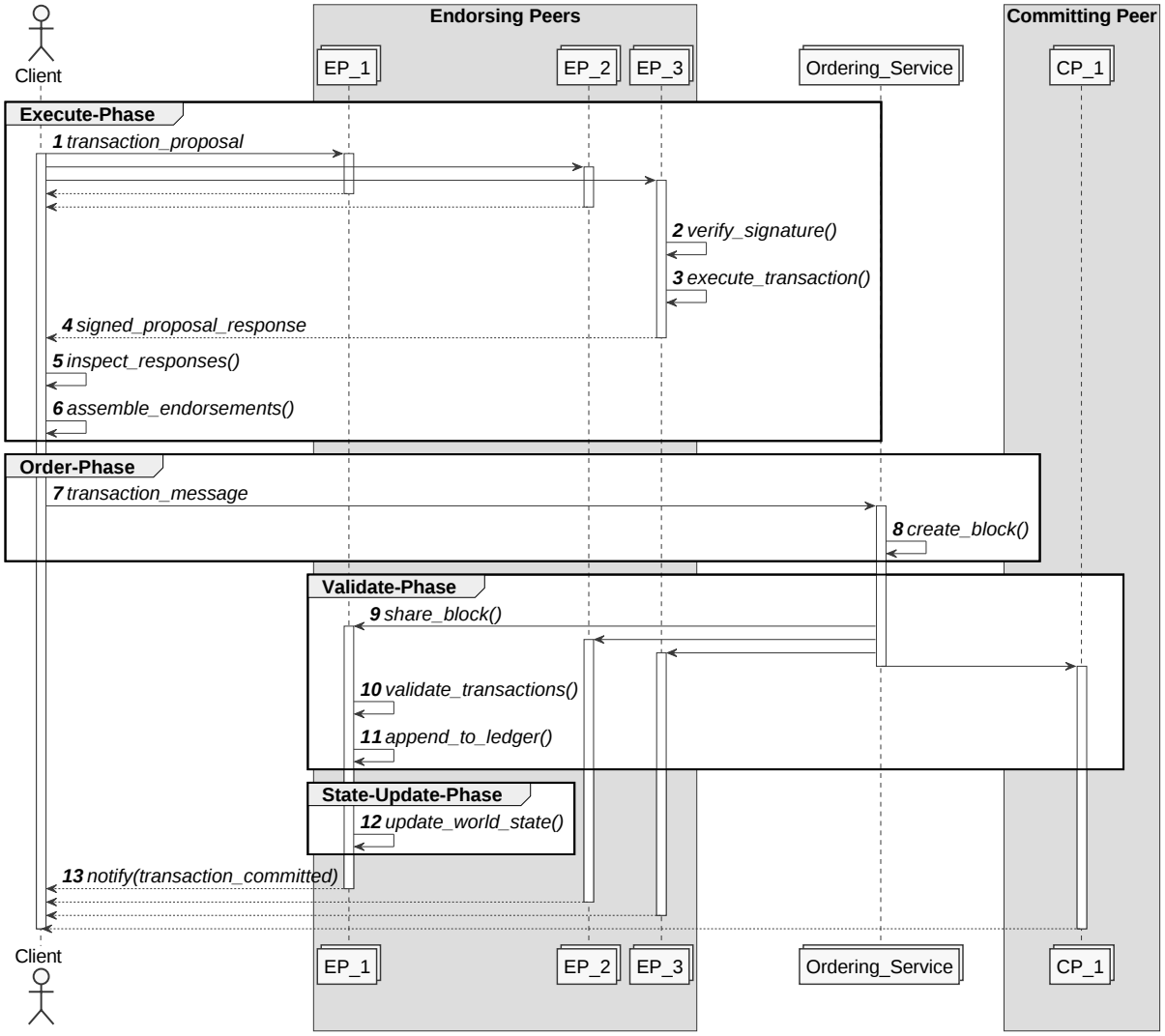

Figure 3.12.: Sequence diagram of the transaction flow according to [3].

3.2.3.3. ACID

Härder and Reuter introduced the atomicity, consistency, isolation, durability (ACID) principle in 1983 to describe the major highlights of the transaction paradigm [43]. It consists of four properties that guarantee the reliability of database transactions [53]. Next, these four properties will be discussed for transactions in HLF.

1. *Atomicity* states that transactions must follow the *all or nothing* rule [53, 43]. This means that either every change made by the transaction occurs or none occurs at all. In HLF a transaction is sent to the ordering service and will eventually be distributed to all necessary peers. Every peer checks if the multi-version concurrency check (MVCC), that checks if the inputs to the transactions exist in the right version in the world state of the peer, holds and they execute the *validation system check*, which checks if enough endorsements are present(see Sec. 3.2.3.1). Based on these previous

checks the transaction is either marked valid or invalid. Nevertheless, it is always present in the block and will always be appended to the blockchain. Regardless, the application of the write-set is only executed if the transaction is marked as valid. Equally important if the transaction is flagged as invalid then the write-set will not be applied [3, 46, 11]. In conclusion, transactions in HLF can be considered *atomic*. Moreover, HLF has its own mechanism that resembles a *commit-protocol*. "Protocols for preserving transaction atomicity are called commit protocols" [75]. The transaction is received whenever the client sends the transaction message to the ordering service. The transaction can be considered received whenever the ordering service added it to a block and distributed it. Finally, the transaction can be considered committed(aborted) whenever it has been validated(invalidated) in the *validation phase* by each peer.

2. *Consistency*: "A transaction reaching its normal end [...], thereby committing its results, preserves the consistency of the database. In other words, each successful transaction by definition commits only legal results" [43]. Moreover, "a database is consistent if and only if it contains the results of successful transactions" [43]. Section 3.2.3.1 described the process of adding successful transactions to the world state. The last phase in the process is the update state phase. All valid transaction's write-sets (legal results) are applied to the world state in this phase. Hence, the world state contains only results of successful (valid) transactions. In conclusion, transactions in HLF are *consistent*.

3. *Isolation:* "Events within a transaction must be hidden from other transactions running concurrently" [43]. Consequently, two different uncommitted transactions that are executed concurrently cannot interact at all. The execution phase (see Sec. 3.2.3.1) describes how transactions are executed in HLF. Importantly, they are executed on the world state of a peer. To elaborate, the world state consists of all previously executed transaction's results. Also, transactions that are executed (simulated) do not alter the world state. Hence, no other transaction that is executed on the same or on other peers can read changes from transactions that are not committed. Therefore, transactions in HLF can be considered *isolated*.

4. *Durability:* "Once a transaction has been completed and has committed its results to the database, the system must guarantee that these results survive any subsequent malfunctions" [43]. According to Medjahed et al. [53] this is usually achieved using database backups and transaction logs. In HLF the ledger consists of 2 parts a *world state* and a *transaction log* (see Sec.3.2.1.5). The transaction log consists of blocks in which each block stores a set of transactions. This transaction log is shared and

stored on all participating peers. Each peer stores this log on its own file system which supports the append only nature of the chain. Consequently, whenever a peer crashes or malfunctions the world state may be lost. Regardless, all transactions are still present in the transaction log which subsequently enables the peer to restore the *lost* world state by applying the results of all transactions. More importantly, when the transaction log of one peer is lost due to hardware failure or malicious access, then the peer can reacquire the missing blocks from other peers through the *gossip protocol* which is used to broadcast ledger and channel data. "The communication layer for gossip is based on gRPC and utilizes TLS with mutual authentication" [11]. It operates by peers receiving messages from other peers and then forwarding this message to several randomly selected peers. In addition, peers may use a pull mechanism instead of waiting for a message. Hence, even if a peer loses its transaction log it can require it which means no transactions that were committed are lost [3, 11, 53]. In conclusions, it can be argued that the transactions of HLF are in fact *durable*.

In conclusion, the previous section shows that HLF transactions do in fact satisfy ACID, The arguments were inspired by Medjahed et al. [53]. However, the argument for all four properties were extended by knowledge from [3] and [11]. Further, transactions in HLF are *executed* concurrently. Importantly, only their validation and the application of their effects happens *sequentially*.

3.2.3.4. Considerations

Lets consider two transactions t1, t2, and a value **X** which is initialized with **1**. The transaction t1 assigns the value $X := X + 1$ and t2 assigns the value $X := X - 1$. In theory if t1 and t2 were executed sequentially in any order, then the value of X would not change. However, if the two transaction t1 and t2 are ordered in the same block in HLF than 2 different outcomes are possible. It is assumed that the transactions were endorsed on the same world state and the same block height.

- t1 comes before t2: t1's readset contains the initial version of X. The read *read-write conflict check* holds and t1 is marked valid. However, when t2 is checked than that same check will fail. This is because t1 already wrote the new value for X. The result is that the version of X is different which leads to a failure of the conflict check. In brief, the value of X is **2** after the block has been ordered. T1 was marked valid and t2 was marked as invalid and is ignored.
- t2 comes before t1: The value of X is **0** after the block has been order. T1 is marked as invalid and t2 is marked as valid.

Further, the example above showed that transactions may be invalid but still be ordered and distributed to all peers. In some cases, this may lead to lots of invalid transactions and thus *overhead* which does not lead to significant changes of the world state. Importantly, the ordering service does not filter out invalid transactions. On the contrary, only peers decide which transactions are valid or not which may lead to a number of transactions which are ordered and part of the blockchain but which are invalid and do not influence the world state.

Further, [43] noted 4 different recovery mechanisms that are applicable for different situations in traditional database systems. Now, let's consider the "Transaction UNDO" or *rollback*. "By definition, UNDO removes all effects of this transaction from the database and does not influence any other transaction." [43]. Let's consider this in the context of a HLF transaction. First, if a transaction is only proposed to a peer it does not change the state of the database. Hence, no changes are made to the blockchain and a *rollback* need not be considered. However, the changes are permanent once the transaction is submitted for ordering and the ordering service published the new block that contained the transaction. If the transaction is considered valid in the *validation phase*, then its changes are applied to each peer's world state. These are the effects of this transaction. However, if the *validation phase* marks the transaction as invalid then its changes are not applied. Hence, only the case were the changes are applied is considered. Khinchi [46] argues that only an additional transaction could rollback the changes made by a previous transaction, because the blockchain is append only. However, this is not trivial. For instance, the effects of rolled back transactions can be read by another transaction which would expand the set of values to be rolled back. Also, even if the values in the world state are restored the *version numbers* of the keys would still change. Consequently, rollbacks are not supported with HLF. Finally, clients, peers and the ordering service can be configured to use transport layer security (TLS) with mutual authentication whenever they communicate with each other. This is possible because each network component has a certificate which it can use for mutual authentication in TLS [11, 3].

3.2.4. Private Data Collections

The private data collection (PDC) is used when one or more organizations that are a part of the same channel (see Sec. 3.2.1.7) need to use confidential data assets that cannot be shared with all organizations on the channel. The PDC is a key value store which only allows the authorized subset of organizations to access the data [73]. Illustration 3.13 shows that the PDC can be seen as being part of the ledger. It consists of two parts:

- *Private data*: The actual private data is shared peer to peer between the authorized

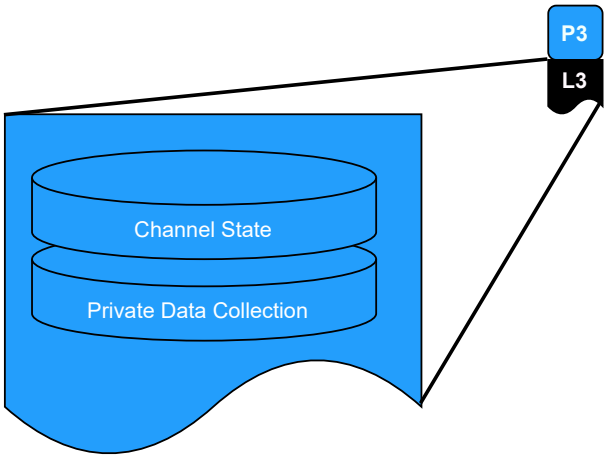

Figure 3.13.: Illustration of the ledger when a private data collection is used [3].

peers through the gossip protocol. The private data is then stored off the blockchain on the *authorized* peers in a so-called *private state database*. Chaincode that is executed on these peers can access the private data. Therefore, only organizations with access to the PDC can endorse transactions that use data from it [73, 3]. Importantly, the ordering service does not participate in the sharing of the private data. Hence, the ordering service cannot access the private data. However, the private data hash will be stored on the blockchain. Thus, the existence of private data is known to the ordering service and all of the channel's participants.

- *Private data hash*: A hash of the private data is stored on the channel's ledger whenever data is written to the PDC [52]. A hash of the key and a hash of the value is stored on the ledger. The result is that every member of the channel notices the existence of transactions that involve a PDC. However, they cannot access the private data if they are not authorized to do so. The hash can be used for auditing and state validation [3]. For instance, the EPC stores the cabinet offer in a private data collection. Next, the EPC sends the cabinet offer to Owner. Then, the Owner verifies that the hash of the cabinet offer that has been sent by the EPC matches the hash on the ledger.

Importantly, PDCs must be defined in the chaincode definition. Hence, every change to the PDC is governed by the *chaincode lifecycle* (see Sec. 3.2.1.4). The following fields are a part of the *private data collection definition*.

- name: The name of the private data collection.
- policy: This policy lists the organizations that can access the PDC. These organizations are called *authorized organizations* and their peers are called *authorized peers*.

- endorsement policy: The collection level endorsement policy is defined here (see Sec. 3.2.2). It applies whenever a transaction uses keys from the PDC.

It can be argued that simply creating a whole separate channel can have a similar effect in some cases. However, the documentation [3] suggests that there is a considerable administrative overhead to creating separate channels. For instance, maintaining MSP's, configuring policies, and deploying chaincode are time consuming activities. Also, the ordering service may have access to the ledger data of multiple channels. However, the ordering service cannot access the data in PDCs. On the other hand, every organization in the channel notices when transactions involving PDCs occur. Complementary to this, channels hide the transactions from organizations that are not a part of the channel [3]. According to Ma et al. [52] channels are best suited when the whole ledger and all transactions between a subset of organizations must be kept confidential from other organizations. Then, PDCs should be used when the occurrence of transactions needs to be shared among all organizations of the channel, but only a subset needs to have access to the transaction's data.

3.2.4.1. Private Data Transaction Flow

The standard transaction flow that is described in the Sec. 3.2.3 differs when a private data collection (PDC) is involved. Figure 3.14 illustrates a simplified sequence diagram of this slightly altered transaction flow with PDCs.

- **Execute phase:** The client submits the transaction proposal. But any private data should be sent with the *transient field* which is a special field in the *transaction proposal*. It is used to send private data to the peer. Importantly, this special field is not included in the final on-chain transaction [3]. Hence, the private data will not be a part of the blockchain (transaction log). A peer that can endorse this transaction is called an authorized endorsing peer (AEP). They are *authorized* to access the PDC and they have the chaincode installed. Hence, these peers can access the PDC and subsequently *endorse* transactions. Figure 3.14 illustrates 3 of these AEPs. In this example only the endorsement of the peer AEP_1 is needed. Further, the client sends the peer the *transaction proposal*. Next, the peer simulates the transaction and produces a read-set, write-set and a return value. In addition, the changes to the PDC are recorded in the *transient data store*. It temporarily stores the changes made to the private data collection by uncommitted transactions. It is purged once the transaction is validated. Also, the entries of the *transient data store* are disseminated to other authorized peers via gossip by AEP_1. Lastly, the

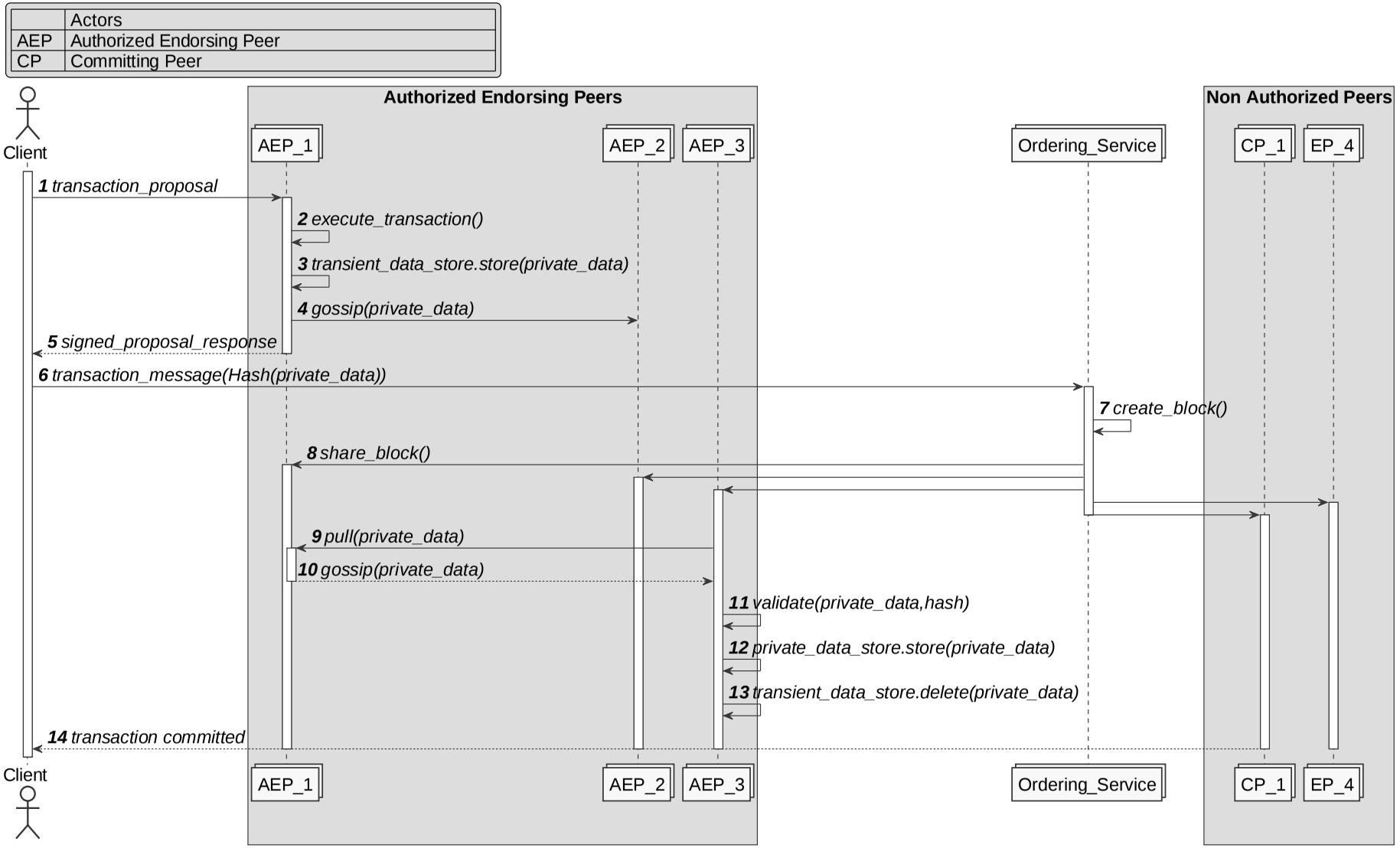

Figure 3.14.: Sequence diagram of the private data transaction flow according to [3].

peer creates the *proposal response* and sends it back to the client. Also, the *proposal response* includes the hashes of the private data.

- **Ordering phase:** The client gathers enough endorsements. In this example the endorsement from AEP_1 satisfies the endorsement policy. Next, the client submits the transaction to the ordering service. This part is identical to the phase that is described in Sec. 3.2.3.1.
- **Validation phase:** This phase works analog to the description in Sec. 3.2.3.1. However, there is one addition. Peers check the collection policy whether they are authorized to have access to the PDC when a transaction includes private data. For example, AEP_2 has already received the private data from AEP_1 in the execution phase. However, AEP_3 notices that it does not have the private data in its transient data store. Thus, it requests the data from another authorized peer. In this case AEP_1. Next, authorized peers update their *private state database* and then they delete the data from the *transient data store*. Peers that do not have access to the private data like the committing peer CP_1 and the endorsing peer EP_4 only validate and then commit the write sets of to their world state.

3.2.5. Policies

The governance of a HLF network is configured through policies. Figure 3.8 shows an example of a channel. The policies are stored in the channel configuration (CC). The following section briefly defines how policies are structured, used and changed. For instance, a policy dictates who is a part of the channel, who are the OSNs of the ordering service, and how chaincodes are deployed on a channel. In brief, each value in the CC is governed by so called groups. Figure 3.15 shows a simplified class diagram of groups. Three important groups are detailed below [3]:

1. Application group: This groups governs the participants of the channel. For instance, the addition, removal, or modification of channel members is governed by this group. Also, the following important policies are defined within this group:
   a) Endorsement: This important policy represents the *default* endorsement policy that is used when no specific endorsement policy is set up for a chaincode in its definition (see Sec. 3.2.1.4). In brief, it specifies which signatures are needed for a default chaincode transaction.
   b) Writers: This policy defines what kind of signatures are needed for a valid transaction proposal. The peer that receives a transaction proposal (see Sec.

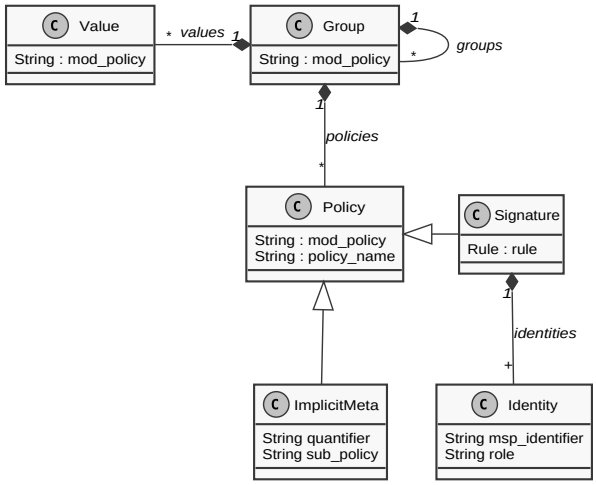

Figure 3.15.: Simplified class diagram of structure of configurations [3].

3.2.3) checks if the signatures that are provided with the proposal satisfy this policy.

c) Admins: This policy is used to govern the changes to the CC itself. For instance, it governs how many signatures are needed when the CC is altered. The result is that new channel members can be added with this policy. Importantly, this is only the default configuration in the test network and in the tutorial [3]. Further, all changes can be governed by other policies which means that this can be changed. The result is that the governance of the CC is highly customization and distributed.

d) Lifecycle Endorsement (see Sec. 3.2.1.4) is defined in this group. This policy is used to check that enough channel members have approved a chaincode's definition before it can be deployed. The default is that most channel member's administrators need to approve a chaincode definition before it can be deployed.

2. Orderer group: This group governs the configuration regarding the ordering service of the channel. It specifies the OSNs, the consensus algorithm that is used, and the details regarding the consensus algorithm. For instance, values like the maximum transaction count per block and the decision to use Raft or Solo can be defined in this groups.

3. Channel group: It is the top hierarchy, and it contains the Orderer and Application group. It specifies the *orderer addresses* which clients use to send their transaction messages and the hashing algorithm which is used to chain blocks to their predecessors. Importantly policy groups are set up hierarchical see Fig. 3.16.

Figure 3.15 illustrates the composition of these configurations. The central component is

the group. A group may contain multiple other groups. It is not possible to form a circle. The result is a directed acyclic graph (DAG) of groups which may be called a *group tree* or *configuration tree*. The groups that have no subgroups are called the leaves of the tree. Each group has a list of values which may or may not be empty. These values are used to store specific configuration values. For instance, the *Orderer* group contains the value *Consensus Type* which is used to store data regarding the consensus model. In the Raft implementation it stores the consent set. This is the list the ordering service nodes which are actively participating in the Raft protocol (see Sec. 3.1.2). Further, a group may contain *policies*. Policies are used to specify a set of signatures. Each policy has a name and a type. Either the policy is a *Signature* policy or an *implicit-meta* policy.

- The *signature* policy specifies a list of concrete identities and a rule. The identities contain a msp_identifier and a role. The msp_identifier specifies the organization and the role the organizational unit within that organization. *Admin*, *Client*, *Orderer*, and *Peer* are default organizational units. *Member* refers to every unit. This allows more granular access control for organizations. Section 3.2.2 shows examples for *Signature* policies in the form of endorsement policies.
- The *implicit-meta* policies are evaluated based on the current *configuration tree*. Importantly, these implicit policies may change whenever new members are added to channels or more ordering organizations are added. The result is that smartly crafted implicit-meta policies can reduce the needed effort whenever new members are added. In brief, the evaluation of *implicit-meta* policies follows the *configuration tree* and results in concrete *signature* policies that are required.

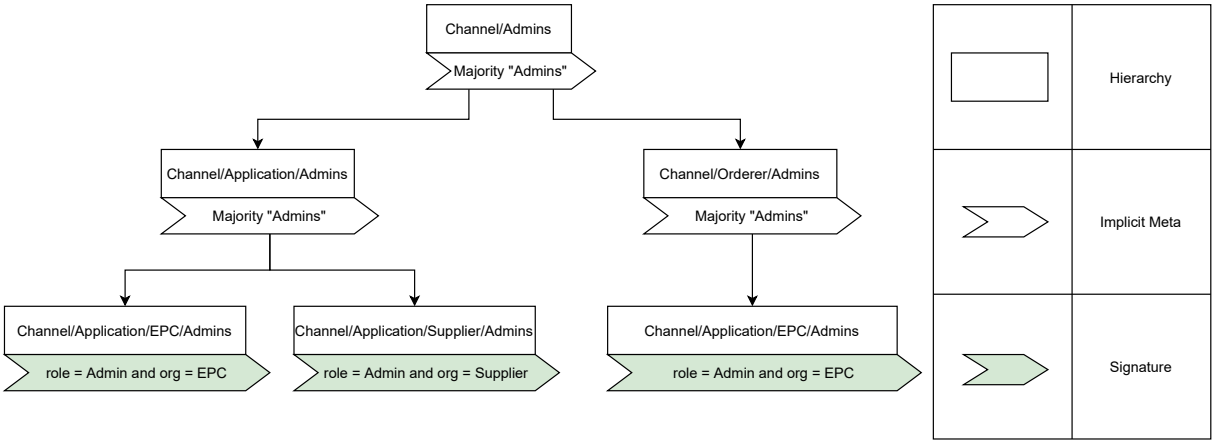

Figure 3.16.: Illustration of an example for implicit meta and signature policies according to [3].

Figure 3.16 illustrates a simplified tree. The implicit-meta policy with the name *Admins* in the group *Channel* should be evaluated. The policy states **"Majority Admins"**. The

quantifier is "Majority". For the sake of completeness, the "All" and "Any" quantifiers also exist. However, the policy states that a majority of the sub policy must be satisfied. The sub_policy is Admins. The result is that the policies called Admins in the groups that are a part of the channel group must be evaluated. The *Application* and the *Orderer* group are subgroups of the *Channel* group. The *Application* group also defined a policy named Admins. This policy also states **"Majority Admins"**. The EPC and Supplier groups are subgroups of the *Application*. They both have a policy named Admins defined. However, this time the policies are not are not implicit-meta policies. They both have signature policies. The policy in the EPC's Admins group requires a signature of an admin of the EPC. The policy in the Supplier's Admins groups requires a signature of an admin of the Supplier. The result is that to satisfy the *Channel/Admins* policy 2 signatures are needed. An Administrator of the EPC and the Supplier is needed.

Further, the configuration tree is not static, and it can be modified. The *mod_policy* or *modification policy* is a part of the Policy, Value and, Group. This policy governs which entities need to sign a proposed configuration update. For a more detailed description of the process that updates the channel configuration please refer to [3, 5]

3.3. Zero Knowledge Proof

A zero-knowledge proof (ZKP) is a cryptographic protocol. It usually involves two participants the *prover* and the *verifier*. The *prover* wants to prove to the *verifier* that a given statement is true. Importantly, no information except that the proof is correct can be extracted from such proofs [24, 40]. An in-depth description of the mathematics behind ZKP's can be found in [39, 17, 20]. Consequently, a ZKP can be useful when handling confidential data that are used in smart contracts. For instance, when an input for a smart contract must remain confidential than it is not possible to distribute this input to other unauthorized nodes of the blockchain network for recomputation. However, a peer that is authorized to view the private input could compute the smart contract function and then provide a proof of the correct execution to the other nodes with the use of a ZKP. In HLF endorsement policies already limit the number of peers that need to compute a smart contract function. However, the committing peers must trust the endorsements. This is where a ZKP could remove this required trust in the peer and replace it with the trust in the mathematics behind the proof.

Recent advances in the cryptographic notably from [20] saw the development of ZK-SNARKs (see Sec. 3.3.1) and ZK-STARKs (see Sec. 3.3.2) both of which are ZKPs. Both approaches may lead to improvements to blockchains in regard to privacy and scalability [20]. Both techniques can be categorized as *universal proof compilers*. In brief, a

proof can be generated for any arbitrary program written in a general purpose language [17]. Hence, the applicability of such proofs is not limited to hand crafted special use cases, and they can be applied to a far broader field. Importantly, the mathematical intricacies are not the focus. Rather, the possible application in regard to blockchain is analyzed.

In brief, endorsements could be enhanced with a proof of the correct execution of a smart contract. Currently HLF allows for a multitude of endorsement policies. For instance, endorsement policies where only one endorsement is necessary are possible. Importantly, how can other peers trust that the endorsed transaction is actually the result of a smart contract invocation? Barbara [17] calls this a computational integrity and privacy (CIP) problem statement. In this context *computational* refers to the fact that the system should handle any general-purpose computation. *Integrity* is the property that the output of the computation must be somehow bound to the computation itself. *Privacy* is the last property. It means that the output and the computation itself can be revealed but the input itself remains private. In brief, some ZKPs offer a solution to these problems. Further, Sec. 3.3.3 shows how a ZKP is used in HLF already to provide more anonymity for transactors. Lastly, 3.3.4 concludes whether this technology is applicable for the use case or not.

3.3.1. ZK-SNARK

This section briefly outlines ZK-SNARKs. A more technical view can be found in [36, 20]. The zero-knowledge succinct non interactive argument of knowledge (ZK-SNARK) was published by Ben-Sasson et al. in the year 2013 [20]. It allows the verification of a NP statement by an untrusted computationally bound prover. NP problems are "the class of problems recognized by nondeterministic Turing machines which run in polynomial time" [48]. The generated proof is publicly verifiable **without** further interaction with the prover. Importantly, ZK-SNARKs need a setup phase that is carried out by a **trusted** third party. This party generates a verification key and a proving key. The published paper focused on verifying the correct computation of a program written in the general-purpose language C [20].

Consequently, a smart contract could be written in C and a trusted third party could provide a proving and verification key. Further, this proof could be used by other peers on the HLF network to verify the correct execution of a smart contract. However, the codes of chaincodes is not required to be published to every peer of a channel (see Sec. 3.2.1.4). Thus, this would have to change for peers to be able to verify the computation. The non-interactive property also means that the proving peers need not be available to conduct

the verification. Further, the ability to prove correct execution in zero-knowledge, enables that some inputs for smart contracts can remain confidential. Further, ZK-SNARKs have been employed in combination with *Bitcoin* to implement a decentralized mix [21].

To sum up, although the capabilities of ZK-SNARKs are interesting and potentially could solve some requirements from Sec. 2.3 the maturity and the ability to combine it with HLF is uncertain. Further, ZK-SNARKs are not mentioned in the documentation as of writing this thesis. In conclusion, ZK-SNARKs cannot be used for this thesis's solution, whereas they could be an integral part of blockchain's in the future.

3.3.2. ZK-STARK

A more recent publication of Ben-Sasson et al. from the year 2018 saw the release of a new zero knowledge proof system [22]. The new proof system is called zero-knowledge scalable transparent argument of knowledge (ZK-STARK). Interestingly, the proposed method is post quantum secure. Consequently, it is secure even if large scale quantum computers exist. The following section will only outline its general properties and gauge its applicability for HLF. A more detailed description of the intricacies of ZK-STARKs can be found in [22]. Nevertheless, a ZK-STARK is different from ZK-SNARKs in that there is no need for a **trusted setup**. Further, ZK-STARK is an interactive protocol. However, a non-interactive version can be produced if the existence of collision-resistant hash functions is assumed, according to Giuffra [39, 32, 22]. However, a non-interactive implementation was not available as of writing this thesis. Additionally, an *academic grade* open source implementation of ZK-STARKs is available [7]. It is written in C++ and thus not usable by HLF chaincodes out of the box (see Sec. 3.2.1.3). However, Benhamouda et al. [23] (see Sec. 3.5.2) already used a C++ library successfully within a go chaincode. Further, the interactive nature of the proof introduces assumptions about the availability of peers. Therefore, such availability assumptions might proof difficult to satisfy.

In conclusion, ZK-STARKs are an interesting new technology that could be used to substitute or enhance endorsements in HLF. However, the existing implementation is not production ready and incompatible with HLF's supported languages. Further, the interactive nature of the proof might prove to be a challenge. For now, this technology cant be used for this thesis's approach, however future developments might make it an integral part of blockchains.

3.3.3. Idemix

Further, HLF can use ZKP's to hide the transactor of transactions. The technical details are described in [12, 3]. Normally each transactor possesses their own certificate, and whenever this transactor sends transaction proposals they include this certificate and they use it to sign the transaction proposal (see Sec. 3.2.3). The result is that the specific certificate is linked to the transaction. For instance, this could allow other organizations to distinguish different employees of an organization. This might not be desirable in some circumstances. The solution to this problem can be idemix. The idemix cryptographic protocol suit can achieve **anonymity**, meaning transactions cannot be linked to the identity of the transactor and **unlinkability**, which enables the execution of multiple transactions without revealing that they were sent by the same transactor.

In brief, the transactions can only be linked to the organization rather than to individual employees and their certificates. Further, it is implemented as a supplementary membership service provider (MSP) (see Sec. 3.2.1.6), and can only be used to execute transactions but not to endorse transactions [3].

3.3.4. Conclusion

In conclusion, ZK-SNARKs and ZK-STARKs are interesting technologies that could be used in the future to supplement and improve the endorsement process of HLF's transaction flow. However, the usage poses significant challenges which are deemed too risky for this thesis. Further, idemix provides a means to hide transactors and only reveal the organizations that they are a part of. However, the requirements of Sec. 2.3 don't need this. Consequently, an idemix MSP is not used.

3.4. Homomorphic Encryption

"**Homomorphic encryption** is the encryption of plaintext to ciphertext while enabling the analysis as if it were plaintext. Mathematical operations such as multiplication and addition can be performed on the ciphertext. In mathematics, homomorphic means the transformation of one data set to another while preserving the relationships between both elements in the sets"[51]. Bernabé et al. conclude that the maturity of homomorphic encryption (HE) is high [24]. Further, Bagdasaryan et al. already used HE in combination with blockchain to provide an access control layer for patient's medical health records. However, they only use partially homomorphic encryption (PHE) which means that they were limited to multiplication operations [15]. PHE allows one operation either addition or multiplication to be performed on the ciphertext without limitations to the number

of operations. Further, somewhat homomorphic encryption (SWHE) allows different operations to be applied to the ciphertext. However, limitations apply to the number of times each operation can be used. Lastly, fully homomorphic encryption (FHE) combines the advantages of PHE and FHE in that an unlimited amount of any operation can be applied without limitations to the number of operations [10]. Consequently, FHE could be used to provide confidential data that is encrypted to a smart contract. The result is that the computation could be replicated on multiple nodes. However, the nodes need not have access to the data to complete the computation. The result is that the addition of FHE could play an important role within HLF. The endorsement policies, private data collections, and channels reduce the amount of organizations that confidential data is exposed to. However, FHE could enable the processing of confidential data by unauthorized organizations while simultaneously maintaining the confidentiality of data assets. Sec. 3.4.1 will introduce examples where HE has been used in combination with HLF. Further, Section 3.4.2 will show how HE could be included in HLF chaincodes. Finally, Sec. 3.4.3 will conclude if this technology is applicable for this thesis's approach.

3.4.1. HE and Hyperledger Fabric

Importantly, Hyperledger Fabric does not mention HE in its documentation [3]. Nevertheless, multiple papers have been published that use HLF and some form of HE. For instance, Chen et al. have shown how privacy related data of citizens can be stored on the blockchain [33]. They used the Paillier algorithm which is an asymmetric homomorphic encryption algorithm [61]. However, it only support addition which means it is only PHE. Nevertheless, the privacy related data such as height, age, and gender were stored as integers. Consequently, statistics such as the average age or gender ratio of the citizens can be calculated. For instance, the encrypted heights of all citizens could be summarized. Next, this sum could be decrypted and divided by the number of citizens. The result is the average height. Importantly, this average can be calculated without revealing the height of an individual citizen. The implementation details were omitted in the paper which makes an adoption more difficult [33]. Then again, it shows that HE and HLF can be used together.

Further, Ghadamyari and Samet used a similar approach [38]. They proposed a novel solution that enables the statistical analysis of patient's private health data. The authors also used the Paillier algorithm which was employed by [33]. Their solution involved off chain key generation, the homomorphic combination of different results with smart contracts, encrypted storage of results on chain and access control using access control lists. Further, they used a preexisting java script library for the Paillier algorithm [38].

Consequently, the authors demonstrated that javascript based chaincodes can be used for HE.

3.4.2. Using HE

The ability of HLF to use various general-purpose languages to implement smart contracts enables it to use preexisting libraries for HE (see Sec. 3.2.1.3). A paper from the year 2017 surveyed 6 general purpose libraries HE which were written in C and C++ for [31].

Further, Brenner et al. mention that HE is built around the low-level circuits which can be compared to programming in assembly language [26]. Further, they describe a possible programming model in which the application developer writes their algorithm in a high-level programming language and then relies on a compiler to produce a so-called homomorphic encryption assembly language [26]. This has not been used in combination with HLF as of writing this thesis.

Lastly, *Microsoft SEAL* is an open source fully homomorphic encryption (FHE) library written mostly in C++. It supports additions and multiplications on integers or real numbers. Other operations like sorting or comparisons are possible but deemed not feasible [69]. Considering the implementation in C++ and the lack of publications that use both HLF and Microsoft SEAL it can be concluded that a practical usage of HE with SEAL poses a significant challenge.

3.4.3. Conclusion

In brief, HE has been used in combination with HLF in an academic setting. The usage of a production ready library in combination with HLF has yet to be discovered. In addition, the use case that is introduced in Sec. 2.1 does not involve statistical analysis of the shared artifacts which has been shown to work in combination with HLF [33, 38]. Further, the programming model introduced by Brenner et al. [26] could be used to compile smart contracts that work with confidential data into a homomorphic encryption assembly language which could be executed on the blockchain while preserving confidentiality. However, this has not been done before with HLF. Consequently, this is deemed too great of a risk and further investigations in this direction were omitted.

3.5. Secure Multiparty Computation

"Protocols for secure multiparty computation (SMPC) enable a set of parties to interact and compute a joint function of their private inputs while revealing nothing but the output" [49]. Also, the abbreviation MPC is also used according to Lindell. Importantly,

the joint function must be agreed upon by the participants [37]. Therefore, the function must be visible to all participants. The technology was first introduced by Yao [82] in the year 1982 where he solved the problem of *two millionaires* who wanted to ascertain who is the wealthiest among them without revealing their wealth to each other. Basically, secure multiparty computation (MPC) processes data that is protected by encryption or a similar method. Protocols that can handle *turing-complete* languages for the joint function are also called *programmable* MPC protocols according to Archer et al. [13]. These protocols are the focus of this section because they might be able to handle the joint execution of smart contracts who are written in general purpose languages. The difference between MPC and HE is that in the former everyone is a data owner that participates in the execution protocol whereas the latter has only one data owner and a different party executes the function [37]. Importantly, HE can be used if the function must remain private. In comparison, the joint function must be distributed to all data owners in MPC. Next, Sec. 3.5.1 will review Enigma which uses MPC in combination with *blockchain*. Lastly, Sec. 3.5.2 will show how MPC is used in combination with HLF.

3.5.1. Enigma

Enigma sets out to be a decentralized computing platform with guaranteed privacy without a trusted third party. It is similar to Ekiden (see Sec. 3.1.5.4) in that it enables the verification of computations without disclosing private inputs. Importantly, MPC does not require trust in a third party like the trusted execution environment (TEE) software guard extension (SGX) that Enigma uses (see Sec. 3.5.1). In brief, private data is split into multiple shares and then distributed to various nodes. The process is also known as *secret sharing*. Secret sharing is a threshold cryptosystem. A secret s is divided into n shares and distributed to n parties. The original secret s can only be reconstructed if at least t shares are combined. If less, then t shares are combined no intelligible result can be obtained. This threshold cryptosystem has been introduced by Shamir in the year 1979 and is also known as *Shamir secret sharing* [72, 45]. Further, these shares retain *homomorphic* properties. Meaning arithmetic circuits can be evaluated on these shares. Further, the recombination of the evaluated shares yields the same result as if the arithmetic circuit were evaluated on the original values [45]. Importantly, there exists a great variety of MPC algorithms which cannot be analyzed in this thesis. Archer et al. provides an overview of different MPC algorithms [13]. Nevertheless, Enigma uses this technology to outsource the computation for smart contracts to so called *computing nodes*. These nodes each get a share of the private inputs and the smart contract code is in the form of an arithmetic circuit. Consequently, the private inputs are safe as longs as less than

t nodes are malicious. Further, the number of computation nodes can be much smaller than the number of participants. This could potentially reduce the amount of computation that has to be done because not every participant has to recompute each transaction. This may reduce resource consumption and enables more extensive computations in smart contracts [85].

Conclusion The paper from Zyskind et al. [85] which describes *Enigma* also provides performance improvements for MPC. Yet, actual benchmarks or sample implementations are omitted. Therefore, any assumptions about restrictions and possible use cases are infeasible on this basis. In 2020 Zyskind wrote a blog post about the future of enigma [84]. This future will focus on TEE instead of MPC. The possible future addition of MPC was only briefly hinted at. In brief, *Enigma* provides an interesting concept of how MPC can be used to implement smart contracts which work on private data.

3.5.2. HLF and SMPC

Benhamouda et al. explored the possibility of adding MPC to HLF version 1.1 [23]. Their work focused around storing encrypted private data on chain. This data would be processed with MPC whenever the transaction used that private data. Their paper described a demo implementation where a seller could list assets with a reserve price. Buyers could publish their private bids on chain. In addition, a smart contract was implemented that used a MPC protocol. To elaborate, a symmetric encryption was done by so called *privileged clients* of each buyer before including the data in the proposal. The private data collections introduced in Sec. 3.2.4 where first included in version 1.2. Hence, the authors could not have taken advantage of this feature and had to rely on symmetric encryption. The MPC implementation used the *EMP-toolkit* that is written in C++. To support the usage of the C++ library in smart contracts that are written in Go the authors opted to use the simplified wrapper and interface generator (SWIG) which automatically creates the bindings between C or C++ and other common programming languages including Go. Furthermore, the authors had to alter the Fabric SDK and include a customized build environment which included SWIG and the *EMP-toolkit*. Importantly, the MPC protocol requires communication between the computing peers during its execution. However, HLF does not enable the communication between peers that execute chaincodes out of the box. Consequently, the authors added a *helper server* to facilitate this communication. Importantly, the client that triggers the auction must propose the transaction to all buyers and the seller as they need to communicate with each other during the endorsement process. In addition, a *local configuration* was added to the peers to enable the storage

of the symmetric keys used to decipher the encrypted private data. The authors are currently working on implementing MPC for HLF although as of April 2021 no additions are mentioned in the documentation [23, 3, 19]. In conclusion, it is possible to use a MPC protocol in combination with HLF through the addition of multiple components. However, no implementation is published as of writing the thesis. Therefore, this approach cannot be used for this thesis.

3.6. Trusted Execution Environments

A "trusted execution environment (TEE) is a tamper resistant processing environment that runs on a separation kernel. It guarantees the authenticity of the executed code, the integrity of the runtime states (e.g. CPU registers, memory and sensitive I/O), and the confidentiality of its code, data and runtime states stored on a persistent memory. In addition, it shall be able to provide remote attestation that proves its trustworthiness for third parties. The content of TEE is not static; it can be securely updated. The TEE resists against all software attacks as well as the physical attacks performed on the main memory of the system. Attacks performed by exploiting backdoor security flaws are not possible" [65]. Consequently, a TEE can be used to execute code and verify its correct execution. For instance, smart contracts could be executed inside a TEE and the correct execution and results could be attested to on the blockchain. In addition, private inputs could be provided to the TEE which would enable confidential inputs to smart contracts while the computation takes place on an untrusted node. Section 3.6.1 introduces an implementation of a TEE. Next, Sec. 3.6.2 presents a paper where HLF and TEE are used in combination. Finally, Sec. 3.6.3 provides a conclusion.

3.6.1. Software Guard Extension

The software guard extension (SGX) is an implementation of a TEE from Intel [83]. SGX enables computational integrity and confidentiality for security-sensitive computations. This is especially useful if privileged software such as hypervisor or kernel are potentially malicious [35]. Hence, it is useful for blockchains where nodes are untrusted and the trustworthiness of the kernel or hypervisor cannot be assumed.

Further, SGX uses so called *enclaves* which are secure containers which contain private data and code which is executed. Importantly, the code and data inside the enclaves cannot be tampered with. For this reason, the CPU produces a cryptographic hash called *mrenclave*. It is computed with the code and data that has initially been loaded into the enclave. Also, the enclave can prove that a specific application and data is loaded.

This process is called *remote attestation*. A user with prior knowledge of the *mrenclave* sends a challenge to the enclave which returns a proof which is called *attestation report*. The user forwards this report to the Intel attestation service (IAS) which verifies it using enhanced privacy ID (EPID) group signatures which are described in [27] and responds with the result. Further, SGX enclaves support a mechanism to recover the internal state of an enclave. Basically, data is encrypted before it leaves the enclave with keys that are only available to the enclave itself [25, 83]. The prior description only briefly outlines the concepts of SGX. A more detailed description is omitted as an extensive portrayal of SGX can be found in an article from Costan and Devadas [35].

3.6.2. TEE for Hyperledger Fabric

Brandenburger et al. investigated the possibility of using the TEE from Intel SGX in combination with HLF [25]. In brief, they implemented a sealed bit auction where only a trusted auctioneer can learn all the bids to evaluate a winner. The authors introduced four new components to HLF because it does not support TEE out of the box [3, 25].

- *Chaincode enclave:* It is the enclave that executes a chaincode. A *chaincode library* acts as intermediary between the enclave and the endorsing peer.
- *Ledger enclave:* This enclave maintains the ledger with additional integrity-specific metadata that represents the most recent blockchain state. The *chaincode enclave* uses this enclave to verify the correctness of the data retrieved from the blockchain state.
- The *enclave registry* is a list of all existing *chaincode enclaves*. It enables peers and clients to first inspect attestations of enclaves before invoking chaincode operations or committing state changes.
- *Enclave transaction validator:* It is responsible for validating transactions produced by the *chaincode enclaves*. Specifically, it checks if the transactions were produced by registered chaincode enclaves.

Next, the chaincode execution is briefly summarized. A client sends a invoke chaincode proposal to a peer. The peer forwards the proposal to the responsible chaincode enclave. The enclave evaluates the proposal and returns a response back to the client. Further, the client uses *remote attestation* to make sure it is communicating with the chaincode enclave. The client can now encrypt the actual transaction proposal and send it to the peer which hands it to the chaincode enclave. Importantly, only the chaincode enclave can decrypt the transaction proposal. Also, the client can provide a encryption key with the transaction proposal which can be used by the enclave to encrypt the transaction

response. Next, the client decides if it wants to submit the transaction to the ordering service after it received the proposal response from the chaincode enclave. This process is analog to the standard transaction flow (see Sec. 3.2.3.1). The ordering service assigns the transaction to a block and broadcasts the block to all peers. The validation step is extended. It now checks if the transaction was produced by the correct chaincode enclave and updates the blockchain state accordingly. An in-depth analysis of this process is provided in [25]. In brief, the committing peers can now verify that the transactions were actually the results of smart contract executions. Consequently, a manipulation of the proposal responses by malicious peers is not possible anymore.

3.6.3. Conclusion

SGX aims to protect the confidentiality of computations and data inside the enclave from software attacks and a limited set of physical attacks. However, Costan and Devadas [35] state that their security analysis revealed that a security conscious software developer cannot in good conscience rely on SGX for secure remote computation. Among other vulnerability physical attacks, privileged software attacks, memory mapping attacks, software attacks on peripherals, cache timing attacks and software side-channel attacks, are possible according to [35]. A detailed survey of these attacks is out of the scope of this thesis. Further, the evaluation of [25] showed that the combination of SGX and HLF produced an overhead of no more than 20% compared to an unsecured implementation. Nevertheless, the results have not been included in HLF as of version 2.2 [3]. Consequently, there is no known implementation of HLF which utilizes TEE. Thus, it cannot be considered for this thesis.

4. Proposed Design - Fabrication Stage

This chapter will lay out the general concept for an abstract multilateral distributed use case. Importantly, the concept is realized as a design for the fabrication stage use case that was introduced in Chap. 2. First, Sec. 4.1 provides an overview of the concept for the fabrication stage use case. Next, the separation of the workflow into different smart contracts will be detailed in Sec. 4.2. Then, Sec. 4.4 introduces formal description of a multilateral workflow and outlines a general solution. Further, Sec. 4.3 will detail the structure of the HLF channels. Also, Sec. 4.5 will establish the ordering service's structure for the channels and discusses their advantages and disadvantages. Lastly, Sec. 4.6 will present the detailed process flow and lists all interactions that happen between the smart contracts and the actors.

4.1. Overview

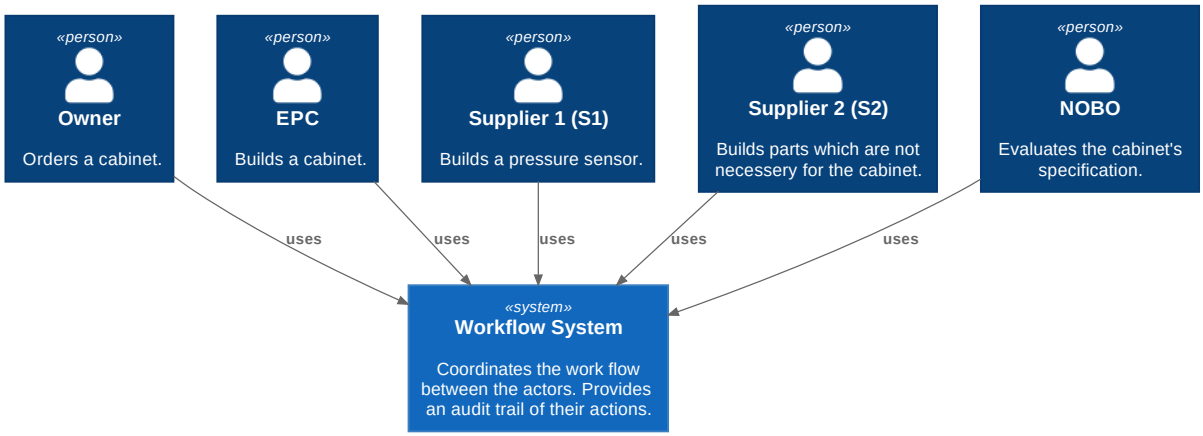

Figure 4.1.: Context diagram for the Workflow System.

Figure 4.1 shows that 5 different actors interact with a workflow system. The owner will own the cabinet, the EPC will build the cabinet, the supplier 1 will built a pressure sensor that will be integrated into the cabinet, the supplier 2 is not directly involved in the cabinet's production, and NOBO will certify that the cabinet that was produced actually satisfies its requirements. The workflow system is the central system which orchestrates the workflow between the various actors. Further, it stores the relevant data assets, enforces access control according to the access control list (see Tab. 2.1), it enforces that

the specific steps in the workflow can only be triggered by authorized organizations, and it will provide an audit trail.

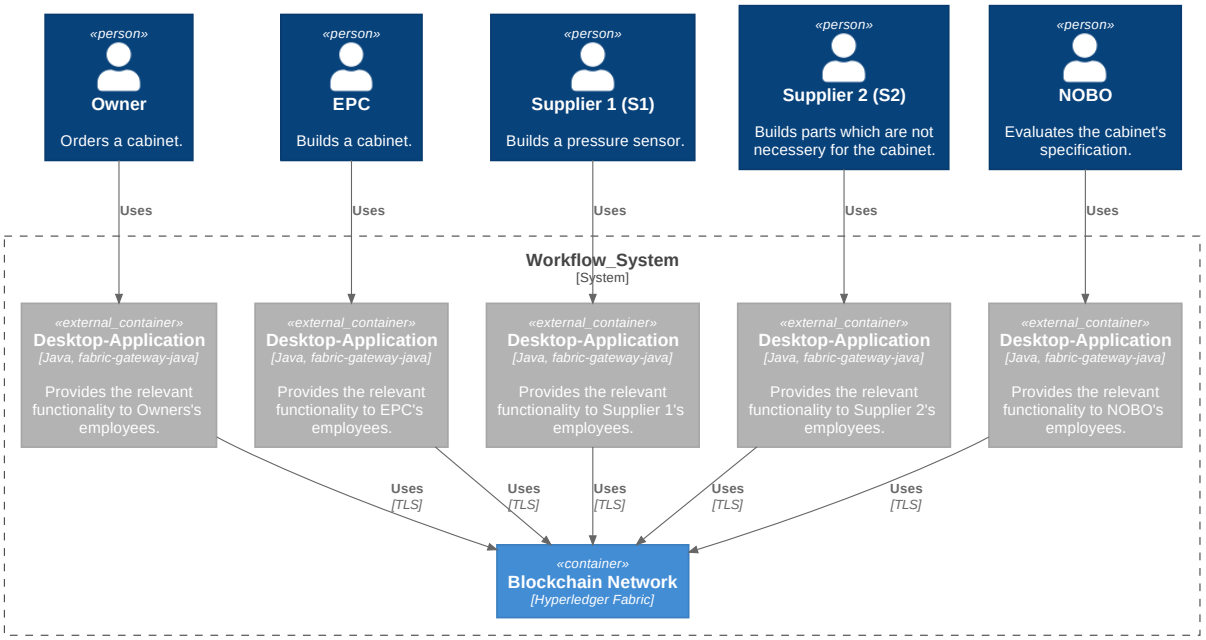

Figure 4.2.: Container diagram for the Workflow System.

Next, Figure 4.2 illustrates the containers that make up the workflow system. Importantly, in this context container is used to describe code running on some computing node. The most important part is the *Blockchain Network* for which Hyperledger Fabric (HLF) is used. In addition, the five external containers represent the client applications which each organization will be using. Further, Figure 4.2 shows Desktop Applications which represent an example and any frontend such as mobile applications or, web applications can be used. Nevertheless, these applications can leverage the fabric software development kit (SDK) to interact with HLF. The SDK is available for Java and Node.js for HLF version 2.2 [3]. Importantly, the SDK is used to trigger transactions in the blockchain network and evaluate queries to smart contracts. This simplifies the implementation for the client applications because they don't have to implement the communication with the peers from scratch. Also, the communication with the blockchain components is secured with TLS.

Figure 4.3 illustrates the logical components, of the HLF networks. The focus of the illustration is the logical structure of the network. In particular, the network is separated into 5 logical components. Importantly, these are logical components and the physical deployment will be described in Sec. 4.3. The logical components are:

- Smart Contracts: 3 different smart contracts were created.

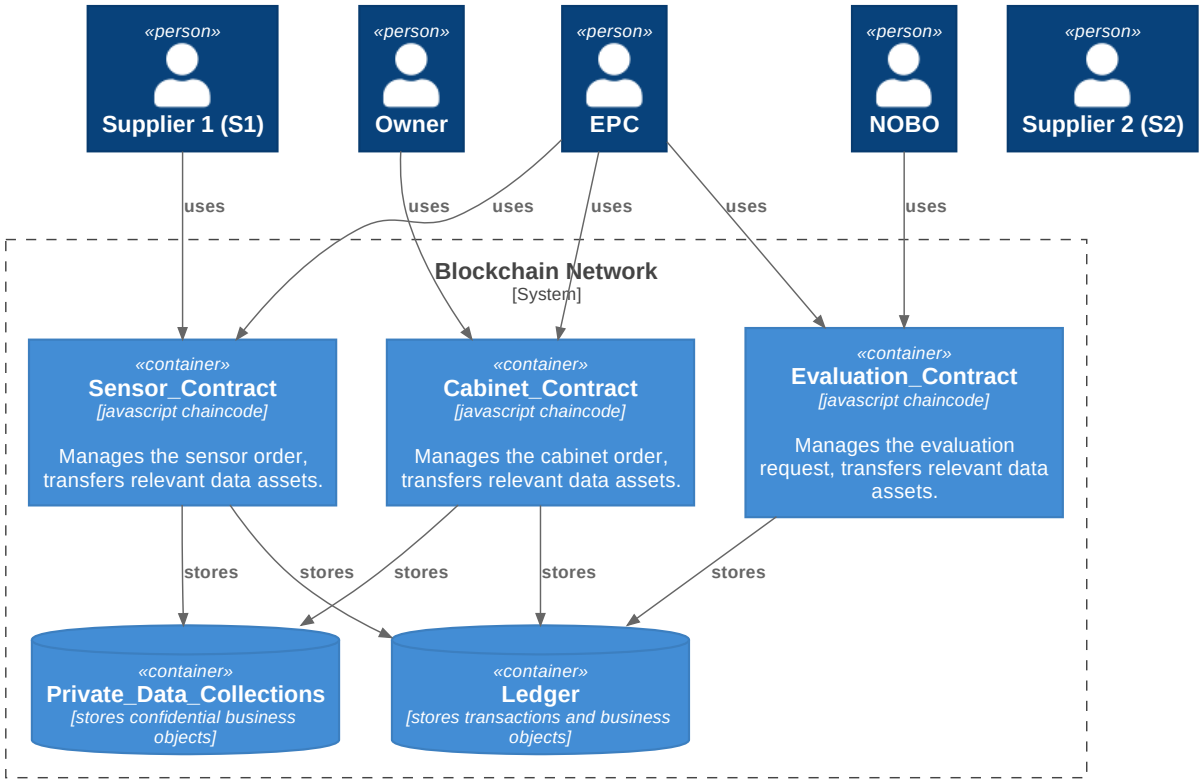

Figure 4.3.: Component diagram for the Workflow System.

1. *Cabinet Contract*: This smart contract is responsible for the sub workflow between the Owner and the EPC. Considering that the order of the cabinet is strictly handled between the Owner and the EPC it is encapsulated in one smart contract.
2. *Sensor Contract*: The sensor contract is responsible for the order of the pressure sensor. Only the EPC and the Supplier 1 are participating in this sub workflow.
3. *Evaluation Contract*: The EPC and NOBO use this contract to manage the sub workflow that leads to the creation of the audit report.

- Private Data Collection: The PDCs are a feature of HLF (see Sec. 3.2.4). They enable the sharing of confidential data assets between a subset of actors.
- Ledger: The ledger is also a feature of HLF (see Sec. 3.2.1.5). It enables the sharing of data assets within the HLF blockchain.

In brief, three different smart contracts are created where each manages a bilateral sub workflow between the **EPC** and one other organization. Importantly, the multilateral workflow can be split up into multiple bilateral sub workflows where each sub workflow is managed by an individual smart contract. In addition, the private data collection (PDC) is used to share confidential data assets because using the channel's ledger would share

the asset with every channel member. However, only the *Cabinet Contract* and the *Sensor Contract* use a PDC. Importantly, the whole workflow could also have been implemented as a single smart contract. However, the separation into 3 different contracts is necessary. The separation enables a deployment of the smart contracts which keeps the business logic of the contracts confidential. In particular, only the participants of each smart contract know its contents and this is enabled by the chaincode lifecycle (see Sec. 3.2.1.4).

4.2. Smart Contracts

Section 4.4 showed that three smart contracts are used to implement the complete workflow. These resulting smart contracts and their class diagrams are illustrated in Fig. 4.4. It shows which workflow steps are a part of each smart contract. Also, smart contracts do not have attributes like the classes of object-oriented programming languages. Therefore, the attributes that are shown in the class diagram are only showing the association of data assets with the responsible smart contract. Table 4.1 illustrates this as well. Importantly, the separation into 3 different smart contracts results in 2 *redundancies* which are caused by separating the workflow into different smart contracts. It can be argued that the Cabinet Contract and the Evaluation Contract could be combined. However, the workflows are separated to keep the business logic confidential between the participants of the smart contracts:

1. D_{CDS}: The cabinet design specification is created by the Owner. The EPC and NOBO need it to perform their tasks. The EPC needs it to gather the information needed to produce the cabinet and to create an offer for the cabinet production. NOBO needs it to evaluate whether the cabinet satisfies its requirements or not. However, NOBO does not directly interact with the Owner. Hence, the D_{CDS} is deployed to the *Cabinet Contract* and to the *Evaluation Contract*. The separation of the smart contracts is the cause for these redundancies. In conclusion, it is the *cost* of the confidential smart contracts.
2. D_{AR}: The audit report is the result of the Evaluation Contract and the sub workflow between the EPC and NOBO. However, the Owner needs to access this audit report before it can decide if it accepts the cabinet. The Owner does not *need to know* of the evaluation contract and the sub workflow between the EPC and NOBO. Hence, the EPC makes the audit report accessible in the Cabinet Contract.

Objects	Smart Contracts	Cabinet	Sensor	Evaluation
D_{CDS}	**Cabinet Design Specification**	✓		✓
C_{CO}	Cabinet Offer	✓		
D_{PSS}	Pressure Sensor Specification		✓	
C_{PSO}	Pressure Sensor Offer		✓	
D_{PSFS}	Pressure Sensor Fact Sheet		✓	
D_{EAS}	EPC Acceptance Sheet		✓	
D_{CFS}	Cabinet Fact Sheet			✓
D_{AR}	**Audit Report**	✓		✓
D_{OAS}	Owner Acceptance Sheet	✓		

Table 4.1.: Association of data assets and the responsible smart contracts.

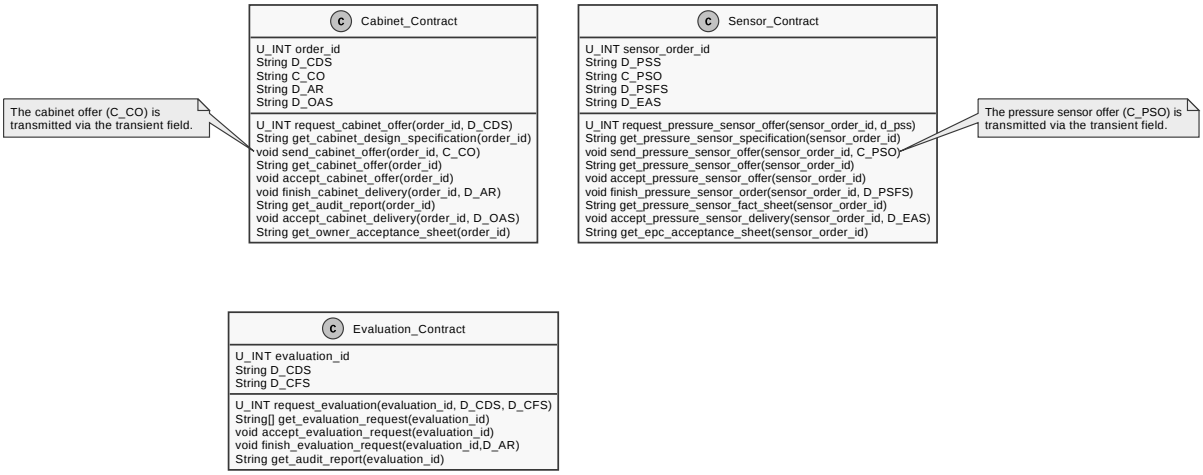

Figure 4.4.: Class diagram for the smart contracts of the fabrication stage use case.

4.3. Channels

HLF offers the channel feature (see Sec. 3.2.1.7) which allows the separation of the participating organizations into channels. The result is that the transactions are kept private from non-channel members. The number of possible channels is huge. If the set of all participants is $Members = \{\text{EPC, Owner, NOBO, S1, S2}\}$ than the total number of possible channels (including the empty channel) is equal to the *cardinality* of the *Power set* of the *Members*: $|Members| = 5, |P(Member)| = 2^5 = 32$. Nevertheless, the concept only uses 2 different channels. Figure 4.5 shows both channels, organizations, and smart contracts. However, the amount of information density in this diagram is high. Thus, Table 4.2 shows which organization participates in which channel. Basically, a channel is created for each distinct access control list AC_i (Tab. 4.5). The access control list of the Cabinet and the Evaluation sub workflow are the same. The result is the Channel 1. Further, Channel 2's participants are the members of the access control list for the Sensor

sub workflow.

Channel \ Member	EPC	Owner	NOBO	S1	S2
Channel 1	✓	✓	✓		
Channel 2	✓		✓	✓	✓

Table 4.2.: Channel participation matrix.

Importantly, why is the access control list of the Sensor sub workflow different form the other two sub workflows. The reason is that the *Owner and Suppliers* are to be separated. The eventual Owner of the cabinet does not *need to know* who the suppliers of the EPC are. Furthermore, the knowledge of the suppliers may reveal business relations which the EPC does not want to disclose and this may cause the EPC to not participate in the workflow system at all. Also, the suppliers do not *need to know* what their products are eventually used for. Consequently, the suppliers are separated from the Owner. Further, Table 4.3 shows to which channel each contract is deployed too. This can also be seen in Fig. 4.5.

Channel \ Contract	Cabinet	Sensor	Evaluation
Channel 1	✓		✓
Channel 2		✓	

Table 4.3.: Contract deployment matrix.

- Cabinet Contract: It can only be deployed to the channel 1 because it needs the Owner and the EPC to operate. Both of which are only participating in the channel 1 together.
- Sensor Contract: It can only be deployed to the channel 2 because it needs the EPC and the S1 to operate. Both of which are only participating in the channel 2 together.
- Evaluation Contract: It could be deployed to both the channel 1 and the channel 2. This is because the necessary participants are are a part of both channels. However, the access control list of the evaluation contract states that only the Owner and not the suppliers may access this contract. Consequently it is deployed on channel 1.

Further, the deployment diagrams A.1 and A.2 illustrate which component is deployed on which peer in addition to their channel participation.

Next, Table 4.4 lists the chaincode level and collection level endorsement policies (see Sec. 3.2.2) that are used for the smart contracts and private data collections. Importantly,

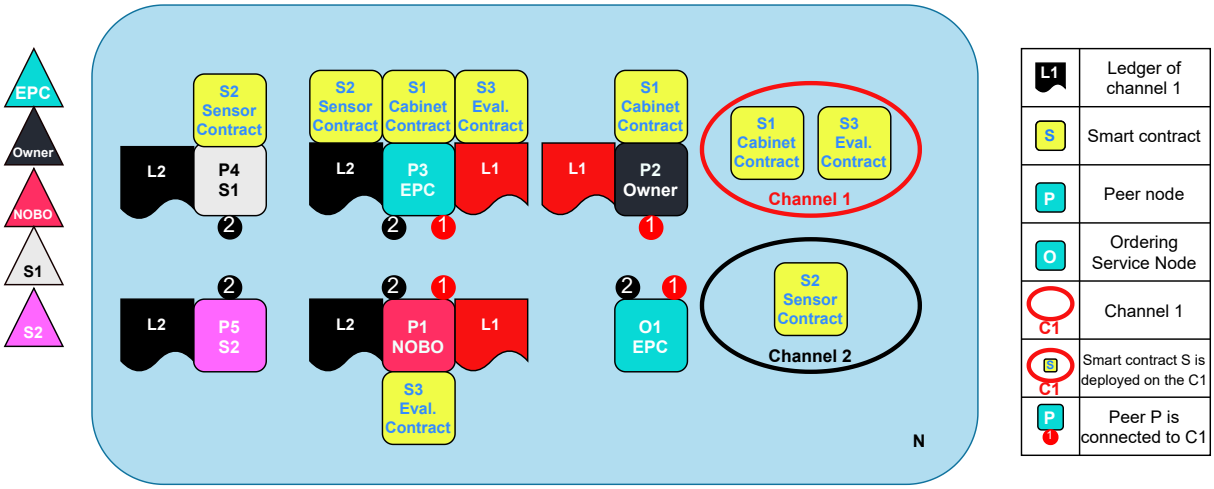

Figure 4.5.: Channel diagram for the Workflow System.

it is required that a *peer* must sign endorsements. The reason for this is that the peers are the components that execute smart contracts which is why they should sign the endorsements. In addition, the collection level endorsement policies and the chaincode level endorsement policies are the same. In case of the Cabinet Contract only the EPC and the Owner invoke functions that change the state of the ledger. Thus, they both have an interest in enforcing that these changes are made according to the smart contract. Hence, both must endorse every transaction whether these transactions involve confidential data or not. Likewise, the same applies to the Sensor Contract. There only the EPC and S1 invoke transactions that change the state of the ledger. Therefore, both want to enforce that the changes to the ledger are made according to the Sensor Contract. Hence, both are required to endorse transactions whether they involve the PDC or not. Finally, the Evaluation Contract is only used by NOBO and EPC. Consequently, both are required to endorse every transaction. Lastly, Fig. 4.5 also shows that each organization runs one peer.

Contract	Endorsement Policies
Cabinet	**AND**("EPC.peer", "Owner.peer")
PDC1	**AND**("EPC.peer", "Owner.peer")
Sensor	**AND**("EPC.peer", "S1.peer")
PDC2	**AND**("EPC.peer", "S1.peer")
Evaluation	**AND**("EPC.peer", "NOBO.peer")

Table 4.4.: Smart contract and PDC endorsement policies.

4.4. Formal Description

Previously, a concrete solution for the fabrication stage use case was shown. However, the solution can also be applied to other multilateral distributed workflows. In general, a multilateral distributed workflow consists of the following. First, the set of all organizations is O and the number of organizations is $|O| = n$. Furthermore, the fabrication stage use case has 5 different organizations $n = 5$. Second, the overall workflow can be split up into multiple different sub workflows $w_i \in W$. The fabrication stage can be split up into 3 different sub workflows which is illustrated by Tab. 4.5.. Moreover, each sub workflow has multiple properties $w_i = (P_i, D_i, C_i, AC_i)$. $P_i \subseteq O$ is the set of participating organizations of the sub workflow. $o_j \in AC_i \subseteq O$ is an organization that is allowed to access the transactions of the sub workflow and the non confidential data assets $d_{ij} \in D_i$. Importantly, $P_i \subseteq AC_i$ which means that the participants of the sub workflow always have access to the workflow. Next, $c_j \in C_i$ is a confidential data asset of the sub workflow w_i which must only be accessible to members of P_i. Lets apply this general concept to the fabrication stage use case of Chap 2.

- $O = \{\text{EPC}, \text{Owner}, \text{NOBO}, \text{S1}, \text{S2}\}$ and $n = 5$

w_i	Name	P_i	C_i	D_i	AC_i
w_1	Cabinet	EPC, Owner	C_{co}	D_{cds}, D_{ar}, D_{oas}	EPC, Owner, NOBO
w_2	Sensor	EPC, S1	C_{pso}	$D_{pss}, D_{psfs}, D_{eas}$	EPC, NOBO, S1, S2
w_3	Evaluation	EPC, NOBO		D_{cds}, D_{cfs}, D_{ar}	EPC, Owner, NOBO

Table 4.5.: Sub workflows of the fabrication stage use case. P_i are the sub workflow participants, C_i are the confidential data assets, D_i are the data assets, and AC_i is the list of organizations that are allowed to access the data assets and see the transactions of this sub workflow.

Each sub workflow w_i will be implemented in its own smart contract. The connection between the sub workflows is done by the organizations which invoke transactions. The access control to the sub workflows is implemented through the channel mechanism (see Sec. 3.2.1.7). In particular, find the minimum number of channels while making sure that each sub workflow's access control list AC_i is satisfied. Further, a PDC is used to assure the access control of the confidential data assets. Finally, the participating organizations o_1, o_2 are forming the endorsement policy of their sub workflow. The confidentiality of the sub workflows is assured because the responsible smart contracts are only deployed on the participating organization's peers. Lastly, the nodes of the ordering service have to be distributed to organizations such that the access control lists AC_i are satisfied. Lastly, the policies of a channel on which the workflows w_i are deployed is made up

such that each organization that participates in a sub workflow that is deployed to the channel can veto any changes to the CC. Thus, all changes have to made unanimously by workflow participants. In short, the proposed design can be used for arbitrary multilateral distributed workflows.

4.5. Ordering and Policy

Next, the structure of the ordering service of the channel 1 (see Sec. 4.5.1) and channel 2 (see Sec. 4.5.2) will be looked at in this section. Moreover, the amount of nodes that participate in the ordering service is of great importance. If the amount of nodes that participate in the consensus protocol is N than it can endure the following amount of node failures.

$$F(N) = \begin{cases} N/2 & \text{if } N \text{ is odd} \\ (N/2) - 1 & \text{if } N \text{ is even} \end{cases}$$

For instance, if 3 nodes are participating than $F(3) = (3/2) = 1$ node can fail. Further, if 4 nodes participate than $F(4) = (4/2) - 1 = 1$ node can fail as well. Thus, failure handling does not improve if 4 nodes are chosen instead of 3. Therefore, it is better to choose an odd number of nodes. For this thesis we expect the ordering service to still work if at most 1 node fails. In addition, the policies (see Sec. 3.2.5) which are responsible for enforcing the governance of the channels are briefly mentioned. Further, only general terms such as, channel member governance and ordering service governance are used. Importantly, policies are highly customization in HLF, and they are stored inside the channel configuration (CC). However, a detailed look at policies is out of the scope of this thesis.

4.5.1. Channel 1

Figure 4.6 illustrates the ordering service for the channel 1. Importantly, the ordering service nodes (OSN) are provided by organizations that participate on the channel. The Raft implementation for the ordering service is used (see Sec. 3.2.1.2). Further, the number of ordering service nodes (OSN) in the consenter set is **three**, and the EPC, NOBO, and the Owner each contribute an OSN to the consenter set. The result is that $F(3) = 1$ which means that this configuration can tolerate the failure of 1 node. In addition, the workflows w_1 and w_3 run on this channel which means that the EPC, NOBO and the Owner are participants in sub workflows on this channel. Thus, all of them have an interest to participate in the shared governance of the ordering service which is satisfied by this design. Lastly, the ordering service and channel members are governed

by the policy **AND**(EPC.admin, NOBO.admin and Owner.admin). Hence, an unilateral agreement among the participants of sub workflows on the channel has to be reached in order to change the relevant policies.

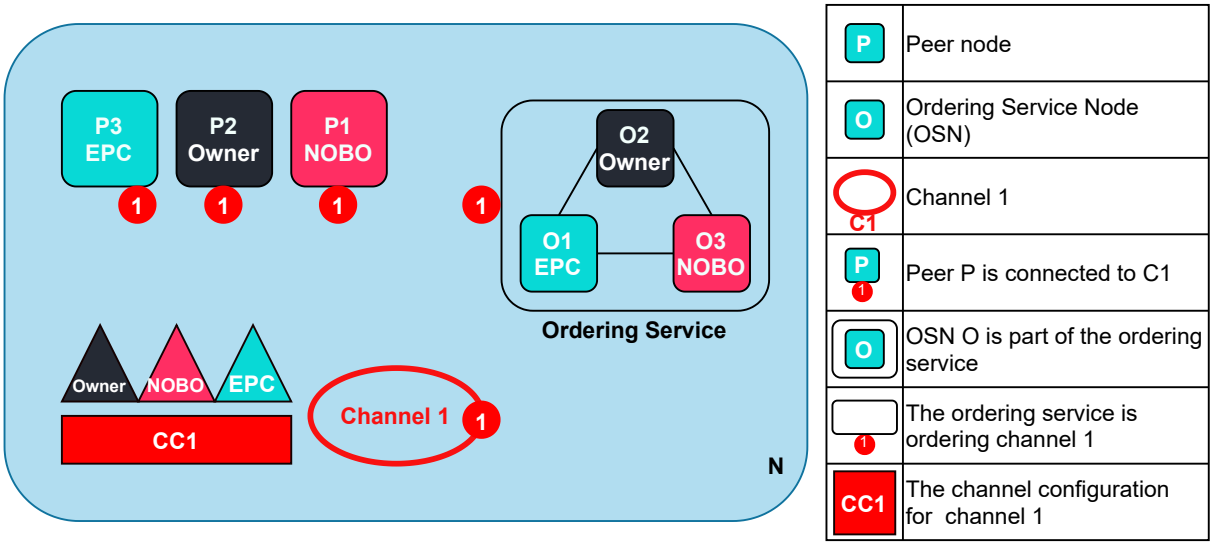

Figure 4.6.: Configuration for the ordering service for channel 1.

4.5.2. Channel 2

Next, Figure 4.7 illustrates the ordering service for the channel 2. Importantly, the channel is used for the sub workflow w_2 where the EPC and S1 participates. However, if 2 nodes are used for the ordering service than $F(2) = 0$ failures can be tolerated. Thus, more nodes need to be added to the ordering service in order to at least tolerate 1 failure. One might think that 2 nodes form the EPC and 1 node from S1 would be the solution. However, such a configuration would enable the EPC to operate a correct ordering service without S1 which would give the EPC total control over the ordering service. The result is that one other organization from the channel will be contribute an OSN. In this case either S2 or NOBO could have been chosen. Figure 4.5.2 shows that S2 has been have been picked for the design. The result is that $F(3) = 1$ which means this design can tolerate 1 failure and the governance of the ordering service is split between S1, NOBO, and the EPC. Further, only w_2 is run on this channels and the EPC and S1 are its participants. The result is that the policies which govern the channel members and the ordering service is **AND**(EPC.admin, S1.admin). Thus, only an unilateral decision from the EPC and S1 can change the relevant policies.

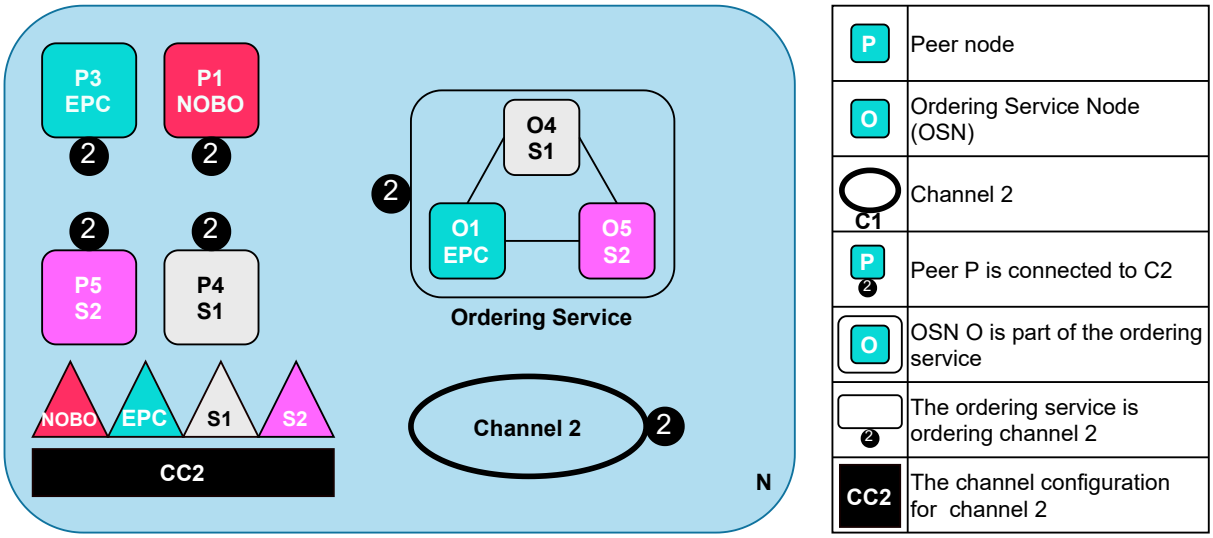

Figure 4.7.: Configuration for the ordering service for channel 2.

4.6. Process Flow

Figure B.2 illustrates a more detailed sequence diagram of the fabrication stage use case. This diagram can be seen as an extension of Fig. 2.4. Importantly, the new diagram shows the smart contracts which are used to invoke each step workflow. In addition, it also illustrates which actor invokes which function of which smart contract. In addition, the diagram also demonstrates which data assets are transferred in each transaction. Further, the sequence diagram is defines four distinct groups. These groups correspond to 3 different bilateral sub workflows:

1. **EPC** $\Longleftrightarrow$ **Owner**:
   - The *Cabinet Order* is illustrated in more detail in the sequence diagram Fig. B.3. Notably, this is the first step that happens in the distributed workflow.
   - *Finish Cabinet Delivery* is described more accurately in the sequence diagram Fig. B.6. Interestingly, this is the last step that takes place. In brief, the *Cabinet Order* and the *Finish Cabinet Delivery* are a part of the same sub workflow w_1 between the EPC and the Owner.
2. **EPC** $\Longleftrightarrow$ **Supplier 1**: The *Pressure Sensor Order* is illustrated more detailed in the sequence diagram Fig. B.4.
3. **EPC** $\Longleftrightarrow$ **NOBO**: The *Cabinet Evaluation* workflow is depicted in the sequence diagram Fig. B.5.

Importantly, the more detailed sequence diagrams show more implementation specific information. For instance, they show where the data assets are stored. Further, it can be

seen whether data assets are stored on the ledger or in the a PDC. In addition, they show an enumeration of the committed transactions that are necessary for the workflow. This is also shown in the Tab. 4.6. Also, they show smart contract queries which are used by the actors to read data assets which they need to be able continue the workflow.

Nr	Contract			Smart Contract Function	Data	Confidential Data
	C	S	E			
1	✓			request_cabinet_offer	order_id, D_{CDS}	
2	✓			send_cabinet_offer	order_id	C_{CO}
3	✓			accept_cabinet_offer	order_id	
4		✓		request_pressure_sensor_offer	sensor_order_id, D_{PSS}	
5		✓		send_pressure_sensor_offer	sensor_order_id	C_{PSO}
6		✓		accept_pressure_sensor_offer	sensor_order_id	
7		✓		finish_pressure_sensor_order	sensor_order_id, D_{PSFS}	
8		✓		accept_pressure_sensor_delivery	sensor_order_id, D_{EAS}	
9			✓	request_evaluation	evaluation_id, D_{CDS}, D_{CFS}	
10			✓	accept_evaluation	evaluation_id	
11			✓	finish_evaluation	evaluation_id, D_{AR}	
12	✓			finish_cabinet_order	order_id, D_{AR}	
13	✓			accept_cabinet_delivery	order_id, D_{OAS}	

Table 4.6.: Fabrication stage use case transactions. The C stands for Cabinet, S for Sensor, and E for Evaluation Contract.

5. Implementation

The prototype was implemented for the concept that has been introduced in Chap. 4. The machine that was use to develop this prototype had the following specs:

- Operation System: Ubuntu 18.04 (bionic)
  - Kernel: 5.4.0-66-generic
- CPU: Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz
- RAM: 10000 MB

Further, various tools were used to facilitate the development of the prototype. Most notably, the development environment Visual Studio Code (VSC) [55] in the version 1.5.3 was used. The ability of VSC to provide various extensions was the main reason for this decision. Importantly, the extension *IBM Blockchain Platform* [54] is available in VSC. Further, this extension simplifies the development process of HLF solutions. In brief, features such as the creation of smart contracts, unit tests, integration tests, packaging of smart contracts, deployment of smart contracts and the invocation of smart contract functions are included. Also, *Microfab* was utilized to simplify the creation of HLF networks. Micofab "is a containerized Hyperledger Fabric runtime for use in development environments" [74]. It can be configured with a plain configuration file that allows for a quick creation of various HLF networks. The configuration that was used for the implementation is shown in Lst. C.1. Importantly, it lists the number of channels, the participants of each channel, and all considered organizations.

Further, Sec. 5.1 describes the most important implementation details surrounding the smart contract development. Finally, Sec. 5.2 analyzes the implementation of the audit trail.

5.1. Smart Contracts

The three smart contracts *Cabinet Contract*, *Sensor Contract*, and *Evaluation Contract* were implemented according to the concept of Chap. 4. The contracts were developed using *javascript*. Importantly, the *IBM Blockchain Platform* extension of VSC was used to create the general folder structure. Figure 5.1 shows a simplified folder structure of

the *Cabinet Contract* project. The *Evaluation Contract* and the *Sensor Contract* have similar folder structures.

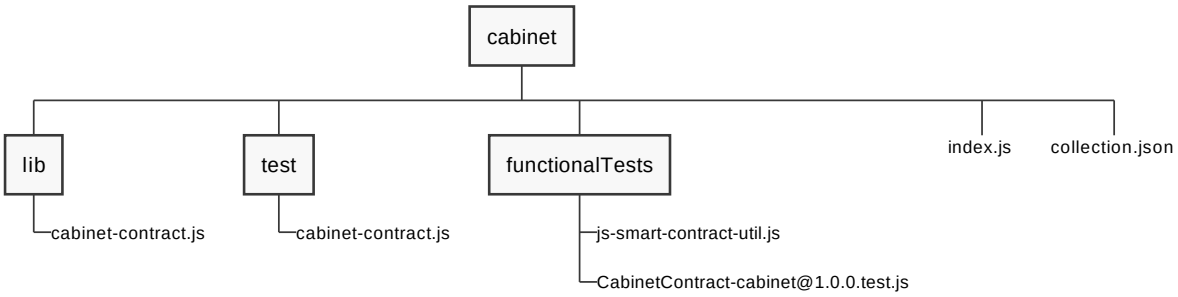

Figure 5.1.: Folder structure of the Cabinet Contract project.

- lib/cabinet-contract.js: This file contains the Cabinet Contract class. The business logic that is needed for the distributed workflow and the audit trail is implemented in here.
- test/cabinet-contract.js: The unit tests for the Cabinet Contract class are in this file.
- functionalTests/:
  - js-smart-contract-util.js: Some helper functions that are used by the integration tests are situated in here. It is automatically generated by the *IBM Blockchain Platform* extension. These functions simplify the interaction with the smart contracts.
  - CabinetContract-cabinet@X.Y.Z.test.js: The integration tests for the Cabinet Contract of the version X.Y.Z are implemented here. These tests directly interact with the smart contracts that are deployed on the *Micofab* network.
- index.js: This file exports all of the smart contracts that are to be included in the cabinet chaincode. In this case only the Cabinet Contract smart contract is a part of the cabinet chaincode. However, multiple smart contracts may be a part of the same chaincode [3] (see Fig. 3.5). However, this prototype uses 3 chaincodes for 3 smart contracts.
- collection.json: This file contains the collection definition that is needed for private data collection (see Sec. 3.2.4). The collection.json for the Cabinet Contract contains the definition for *PDC1* (see Lst. C.2). The collection.json for the Sensor Contract defines *PDC2*. Importantly, the Evaluation Contract does not have a PDC and thus no collection.json.

Further, Sec. 5.1.1 describes how the smart contracts interact with the world state. Next, Sec. 5.1.2 demonstrates how a private data collection (PDC) can be used in practice. Lastly, Sec. 5.1.3 details how access control is implemented in HLF for transactors.

5.1.1. World State

Smart contracts are able to directly interact with the world state which is a part of the ledger (see Sec. 3.2.1.5). The so-called *transaction context* is utilized for this purpose. "It provides access to a wide range of Fabric APIs that allow smart contract developers to perform operations relating to detailed transaction processing" [3]. Importantly, the getState and putState operations are used to read from and write to the world state.

1. *getState*: This function can be used to retrieve an object of the key value store. For instance, Listing 5.1 illustrates its usage. The code snipped is taken from the get_cabinet_design_specification function from the Cabinet Contract. The function *getState* is called with the parameter order_id in line 1. Importantly, the *getState* function is a member of the *transaction context*. Line 2 parses the returned value and converts it into a usable object. The result is that an object from the world state has been read by the smart contract.

```
1  const buffer = await ctx.stub.getState(order_id);
2  const object = JSON.parse(buffer.toString());
```

Listing 5.1: World State read access example from the get_cabinet_design_specification function.

2. *putState*: Listing 5.2 shows how objects can be written to the world state. The buffer contains the object that will be written to the world state. Next, the *transaction context* is used to call the *putState* method. It will write the value *buffer* to the key *order_id*.

```
1  const buffer = Buffer.from(JSON.stringify({ d_cds: d_cds}));
2  await ctx.stub.putState(order_id, buffer);
```

Listing 5.2: World State write access example from the request_cabinet_offer function.

5.1.2. Private Data Collections

The interaction with a private data collection (PDC) can be separated into 4 parts. First, the PDC has to be defined. This is done through the collection.json file (see Sec. 5.1). Second, the confidential data has to be sent to the smart contract. Importantly, this the special *transient field* (see Sec. 3.2.4) is utilized for this purpose. Next, values can be read from a PDC. Lastly, private values can be written to the PDC.

1. *Collection definition*: The collection definition is included in the chaincode definition and will be committed to the blockchain during the chaincode lifecycle (see Sec. 3.2.1.4). Listing C.2 shows the collection definition that is used for the *PDC1* which is used for the Cabinet Contract. Importantly, it can be seen that the policy states: **OR**(EPCMSP.member, OwnerMSP.member). It defines that only the EPC and the Owner are authorized to read and write the *PDC1*. The result is, that only the EPC and the Owner can endorse transactions that use the *PDC1*.

2. *Transient field*: The transient field is used to provide inputs to smart contract functions that are not a part of the transaction that persists on the blockchain. Hence, data that will be written to private data collections should usually be transferred through this mechanism. Importantly, the smart contract function can expect a transient field and use its contents. However, the transient field layout or contents are not defined anywhere. For instance, the helper function *get_transient_field* is used to access the transient field (See Lst. 5.3). Line 2 accesses the transient map through the *transaction context* and stores it. Lines 3-5 check if the Map is not empty and if the requested field is inside the map. Line 6 returns the contents of the map. The result is, that the developer has to document the requirements for transient fields on their own.

```
1 async function get_transient_field(ctx, fieldname) {
2 const transientData = ctx.stub.getTransient();
3 if (transientData.size === 0 || !transientData.has(fieldname)) {
4 throw new Error(`The transient field is missing the ${fieldname}
    field`);
5 }
6 return transientData.get(fieldname).toString();
7 }
```

Listing 5.3: Example for the access of the transient field.

3. *putPrivateData*: Listing 5.4 shows an example from the *send_cabinet_offer* function where the confidential data asset *cabinet offer (c_co)* is stored in the PDC1. The main difference between private data and world state write access is that a *collection name* has to be specified in the putPrivateData function. Line 1 sets the name to *PDC1*. Notably, this name is defined in the *collection definition* of the Cabinet Contract (See Lst. C.2).

```
1 const collectionName = 'PDC1';
2 await ctx.stub.putPrivateData(collectionName, order_id, Buffer.from(
    JSON.stringify(c_co)));
```

Listing 5.4: Private Data write access example from the send_cabinet_offer function.

4. *getPrivateData*: Listing 5.5 shows an example from the *get_cabinet_offer* function where the *cabinet offer (c_co)* is read from PDC1.

```
1  const collectionName = 'PDC1';
2  const privateData = await ctx.stub.getPrivateData(collectionName,
       order_id);
3  return JSON.parse(privateData.toString());
```

Listing 5.5: Private Data read access example from the get_cabinet_offer function.

Importantly, the queries for PDCs only work as intended if the peer that is executing the chaincode is also *authorized* to store the PDC. For instance, NOBO is not authorized to access PDC1. Consequently, an Error would be thrown whenever the send_cabinet_offer function were proposed to NOBO's peer.

5.1.3. Authorization

Importantly, each step of the workflow should only be executable by an authorized subset of organizations. Hence, each function of a smart contract should be able to specify a set of authorized transactors. For instance, the *request_cabinet_offer* function should only be able to be executed by the *Owner*. This can be achieved through the *transaction context*. Notably, the **transactor** can be accessed through the *transaction context* from inside the smart contract.

```
1 const identity = await ctx.clientIdentity.getMSPID();
2 authorize_transactor(['OwnerMSP'], identity);
```

Listing 5.6: Authorization example from the request_cabinet_offer function.

Listing 5.6 shows an example of this. Line 1 reads the transactor's identity from the context. Importantly, this is derived from the signature of the transaction proposal (see Sec. 3.2.3.2). The getMSPID() function returns the name of the MSP (see Sec. 3.2.1.6) as a String. The helper function *authorize_transactor* (see Lst. 5.7) then checks if the given MSP is authorized to execute the function. Importantly, it can be seen that the list of authorized transactors is hard coded in this example. The string "OwnerMSP" is static. This is a simplification and it is not recommended for production implementations. However, for a demo it is sufficient. Further, Sec. 5.2.2.1 shows how a dynamic access control list can be implemented. This is recommended for a production environment.

```
1 function authorize_transactor(list_of_allowed_msps, transactor) {
2 const authorized = list_of_allowed_msps.includes(transactor);
3 if (!authorized) {
4 throw new Error(`This function may only be called by ${list_of_allowed_msps
    .toString()} you are ${transactor}`);
```

```
5 }}
```

Listing 5.7: authorize_transactor function.

5.2. Audit Trail

Further, the execution of the distributed workflow has to produce an *audit trail*. Importantly, the private data that is a part of the PDCs must not be leaked with the audit trail. The transaction log of the ledger (see Sec. 3.2.1.5) contains all the transaction that are necessary for the audit trail. The result is that it is possible to query all blocks from the peers and then subsequently create a list of transactions which correspond to a certain workflow. Then, these transactions could be queried and filtered to find the desired data for an audit trail. However, this was not implemented in this prototype. The audit trail was implemented with the smart contracts themselves. This has the advantage that the authorization of the smart contract functions can be used for the audit trail as well. Two different options were explored. First, Sec. 5.2.1 explores a *manual* implementation of the audit trail. Second, Sec. 5.2.2 shows the *automatic* implementation.

5.2.1. Manual

Manual approach means that every function that shall add data to the audit trail will be altered *manually*. Consequently, a considerable amount of additional development effort is required. In brief, an additional field called *audit_trail* is added to the corresponding object. For instance, the cabinet contract object receives an additional field called *audit_trail*. This can be seen in Lst. 5.8 in line 1-3.

```
1 const audit_trail =
2 {request_cabinet_offer: `function : request_cabinet_offer (order_id = ${
    order_id}, d_cds = ${d_cds})
3 transactor = ${ctx.clientIdentity.getMSPID()}` };
4 const buffer = Buffer.from(JSON.stringify({ d_cds: d_cds, audit_trail:
    audit_trail }));
5 await ctx.stub.putState(order_id, buffer);
```

Listing 5.8: Manual audit trail example from the request_cabinet_offer function.

The audit trail will be extended by each subsequent transaction listed in Tab. 4.6. Further, the audit trail is distributed across all three contracts. The result is that the audit trail is governed by the participants of the workflow per design. Regardless, Line 4 shows that the d_cds and the audit_trail are written to the world state. The audit trail includes the name of the function, the parameters with which the function was called and the

transactor. However, it can be extended to include the same fields that the *automatic* implementation uses.

```
1 const c_co_hash = crypto.createHash('sha256').update(c_co).digest('hex');
2 audit_trail.send_cabinet_offer =
3 `function : send_cabinet_offer (order_id = ${order_id}, Hash(c_co) = ${
    c_co_hash}) transactor = ${identity}`;
```

Listing 5.9: Audit trail example with private data from the send_cabinet_offer function.

Next, Listing 5.9 shows an example from the *send_cabinet_offer* function where confidential data assets are used. Importantly, the *cabinet offer c_co* must not be accessible through the audit trail. Consequently, Line 1 creates a hash from the *cabinet offer* which was received through the transient field. Also, a salt should be added to protect against rainbow tables which were investigated by Oechslin [59]. Further, Line 2-3 adds the audit trail for the key *send_cabinet_offer* which includes the hash of the cabinet offer.

5.2.2. Automatic

The second version if the audit trail is *automatically* created whenever a transaction is executed. Importantly, the *automatic* audit trail is implemented once and then creates the audit trail for every transaction of a smart contract. In contrast, each function has to be altered for the *manual* version. The mechanism that is used for this is called *transaction handlers*. Notably, three different handlers are available [3]. Each smart contract can implement these handlers.

- *beforeTransaction*: This handler gets called before a transaction is executed on a peer. It can also access the world state.
- *afterTransaction*: The afterTransaction handler is invoked whenever a transaction finished its execution. Further, it receives the result of the transaction as an additional input in contrast the beforeTransaction handler does not have access to the result.
- *unknownTransaction*: Whenever a transaction is invoked that is not implemented this handler is invoked.

The implementation uses the *beforeTransaction* handler to implement the *automatic* audit trail. Listing C.3 shows the implementation of the beforeTransaction handler. Most notably, it creates an audit trail object in the world state with the key ('audit' + order_id) in case of the Cabinet Contract. Each transaction that uses a specific order_id results in an additional entry in the *audit trail object* for this order_id. Whenever, the *transient field* is used to transfer private data to a smart contract function than a hash of the transient

data is added to the audit trail object. This can be seen in the lines 19-20 and 31. The following assumptions are made in order for the *automatic* audit trail implementation to work.

1. The first argument of all functions is always the id of the object. For instance, the order_id is used in case of the Cabinet Contract.
2. The key of the audit trail object will not be written to in the actual smart contract function. If the smart contract function writes to the audit trail object than this will negate the changes of the beforeTransaction function.
3. The ID's of the actual business objects don't start with 'audit'. This would overwrite the audit trail objects.

5.2.2.1. Authorization

The participants of each smart contract are authorized to access the audit trail. This is the default configuration. Table 5.1 illustrates the default read access rights for the audit trails. This authorization is enforced through the method mentioned in Sec. 5.1.3. However, this method only handles static access control. Further, the audit trail might have to be shared with other organizations. Consequently, an additional *dynamic access control* method was also implemented. In brief, an access control list is maintained for each audit trail object. The participants of the smart contract are authorized to add and revoke access dynamically.

```
1  const identity = ctx.clientIdentity.getMSPID();
2  const audit_trail_id = 'audit' + order_id;
3  const buffer = await ctx.stub.getState(audit_trail_id);
4  const object = JSON.parse(buffer.toString());
5  authorize_transactor(object.access_control_list, identity);
```

Listing 5.10: Dynamic access control example from the function get_audit_trail_automatic_access_control_list.

Listing 5.10 shows how the access control list is used to dynamically enforce access control. First, the transactor is read from the transaction context. Next, the audit trail object is read from the world state. Finally, the access control list inside the audit trail object is used to check if the transactor is authorized. Further, the function defined in Lst. C.4 implements the revocation of access for a specific audit trail. Lastly, Listing C.5 is the implementation which is used to grant access to an audit trail object. Importantly, the revocation and granting of access to audit trail objects is managed by the participants of the smart contract. For instance, the access to audit trail objects of the Cabinet Contract

Actors \ Audit Trail	Cabinet	Sensor	Evaluation
EPC	✓	✓	✓
NOBO			✓
Owner	✓		
S1		✓	
S2			

Table 5.1.: Default access control list for the audit trail.

is managed by the EPC and the Owner. Each of them can revoke and grant access to other organizations. Importantly, this mechanism can be adapted to manage access to (non) confidential data assets, audit trails and workflow steps. In addition, the revocation and granting of access could also be governed by a dynamic access control list. However, this adds complexity which is not needed for the prototype.

5.2.3. Conclusion

The previous sections showed two different implementations for the audit trail. The first involves more code that has to be added manually. The second involves only one function that automatically creates the audit trail. However, it makes certain assumptions (see Sec. 5.2.2) which must hold for the audit trail to be correct. In addition, Sec 5.2.2.1 showed how the access to the audit trail can be statically and dynamically controlled with access control lists.

6. Evaluation

The following chapter conducts a requirement mapping in Sec. 6.1. Further, it evaluates whether the requirements for the fabrication stage use case that have been set up in Sec. 2 are satisfied by the proposed approach. In addition, a feature mapping is conducted in Sec. 6.2. It shows which features of HLF or the proposed approach are responsible for satisfying which requirement. Next, Sec. 6.3 answers the research questions raised in Chap. 1. Lastly, Sec. 6.4 proposes attacker models which are applied to the design.

6.1. Requirements Mapping

ID	Name	Solution	SAT
FA01	Asset Creation	The smart contracts which are deployed to both channels allow actors to create one or multiple assets depending on the workflow.	✓
FA02	Access Control on Assets	Read access to data assets can be enforced through the usage of a PDC or multiple channels. Write access is is enforced through the smart contracts themselves.	✓
FA03	Revoking Access	Write access rights can be revoked through the smart contracts. Read access rights can be revoked if organizations are removed from channels or the data asset is moved to a PDC.	✓
FA04	Transf. Assets	The right to manage access through the smart contract can be transferred trough a transaction, and it is enforced by the smart contract itself.	✓
FA05	Access Traceability	It is **not** possible to reliably trace access to data assets. The transaction flow (see Sec. 3.2.3.2) shows that clients can decide whether to submit transactions for ordering or not. Consequently, queries don't leave traces on the blockchain if the client decides not to submit the transaction.	X
FA06	AC auditability	The *access control list* is stored in the world state. Consequently, changes made to this list require ordered transactions. Thus, it is known who changes what access control list for which data asset. An audit trail can be implemented to make these changes more accessible (see Sec. 5.2).	✓

Table 6.1.: Requirements mapping for the functional requirements for data assets of the fabrication stage use case. (satisfied (SAT))

ID	Name	Solution	Satisfied
SP01	Actor Auth.	Each participant in the HLF network must have a MSP. Thus, they can authenticate themselves and have a verifiable identity.	✓
SP02	Transaction Auth.	The transaction flow of HLF requires that each transaction proposal is signed by the transactor.	✓
SP03	Identity Management	Organizations can use their existing PKI infrastructure.	✓
SP04	Non-Repudiation	Every transaction that is submitted to the ordering service will be a part of a block in the blockchain. Consequently, each "interaction" that changes the state of the ledger leaves tamper proof evidence. In addition, the audit trail makes these interactions more easily accessible.	✓
SP05	Accountability	Section 5.2 shows how an audit trail can be provided. The limitation is that only write access can be traced reliably.	✓
SP06	Data in Transit Conf.	Clients, peers and the ordering service can be configured to use transport layer security (TLS) with mutual authentication [11, 3]. Hence, the data in transit is cryptographically protected.	✓
SP07	Data at Rest Conf.	Confidential data is only stored in a PDC. These collections are only replicated on authorized peers. Consequently, if the access to these peers is controlled than the data at rest confidentiality is assured.	✓
SP08	Access Control	Channels and a private data collection (PDC) can be used to control the access to data assets on a peer level. Authorization can be used to control access on a transaction or client level.	✓
SP09	Integrity	The audit trail can be used to verify the integrity of confidential and not confidential data assets (see Sec. 5.2).	✓
SP10	Organizations	Channels enforce that only authorized participants can take part in the system (see Sec. 3.2.1.7).	✓

Table 6.2.: Requirements mapping for the security and privacy requirements for the fabrication stage use case.

Importantly, this Section checks whether the requirements of Sec. 2 are satisfied. Further, each requirement will be classified into being satisfied(✓) or not satisfied(X). Also, a

few sentences explain this classification for each requirement. Table 6.1 addresses the functional requirements for data assets from Tab. 2.2. Next, Table 6.2 addresses the security and privacy requirements from Tab. 2.3.

6.2. Feature Mapping

Next, Table 6.3 lists all available requirements from Sec. 2.3.2. In addition, each requirement is associated with features from HLF. For instance, MSP, transactions PDC, and channels. Also, requirements may be associated with parts of the proposed design as well. For instance, authorization (see Sec. 5.1.3 and Sec. 5.2.2.1) and audit trail (see Sec. 5.2). In brief, the association in the table illustrates which features were most important in (not) satisfying or the requirement.

ID	Authorization	Audit Trail	MSP	Transactions	PDC	Channels
FA01	✓					
FA02	✓					
FA03	✓					
FA04	✓					
FA05				(✓)		
FA06		✓				
SP01			✓			
SP02				✓		
SP03			✓			
SP04		✓		✓		
SP05		✓				
SP06				✓		
SP07					✓	
SP08	✓					✓
SP09		✓				
SP10						✓

Table 6.3.: Mapping of HLF features and implementation concepts to the fabrication stage's requirements.

6.3. Research Answers

The following Sec. answers the research questions that have been raised in Tab. 1.1.

RQ1: This thesis identified four key technologies that can enable blockchains to work with confidential data assets. Chapter 3.1 lists zero-knowledge proof (ZKP), homomorphic encryption (HE), trusted execution environment (TEE), and secure multiparty computation (SMPC) as possible technologies. Further, multiple blockchains were

also identified in this chapter. In addition, Chapter 3.2 listed Hyperledger Fabric (HLF) in more detail and showed the features which enable this technology to work with confidential data assets and workflows.

RQ2: The use of different channels and the chaincode lifecycle allows the deployment of smart contracts which are only accessible to a subset of participants. Hence, confidential workflows are possible with this design. Moreover, the possible endorsement policies are directly dependent on the deployment of the smart contracts. Thus, deployment of the smart contracts to fewer peers results in endorsement policies which can only include these few peers. Further, endorsement policies that are very restrictive may lead to less availability because the reliance on fewer peers increases the chance that one of the required peers is unavailable. This increased reliance on few peers may be compensated with an increased number of peers for that organization. In contrast, very unrestricted endorsement policies lead to an increased risk of malicious smart contracts. Meaning that peers attackers need to control less peers in order to fulfill the less restrictive endorsement policy. In conclusion, the decision where to deploy smart contracts and what the endorsement policies look like is not trivial.

6.4. Attacker /Adversary Models

The proposed solution may be a target of external or internal attackers. However, the replication of the ledger, endorsement of smart contracts, and consensus models thwart some attacks. Saad et al. mention attacks on the blockchain structure or peer to peer communication [64]. However, these attacks are general and cannot be applied to the specific use case. Consequently, they are not considered. Also, Putz and Pernul lists multiple vulnerabilities for the Hyperledger Fabric (HLF) blockchain. For instance, vulnerabilities in the framework, dependencies or in the cryptography [63]. Again, these mentioned vulnerabilities are independent from the use case. Also, Yamashita et al. lists potential risks of HLF smart contracts [81]. In conclusion, this section will define two distinct attacker models in Sec 6.4.1 that will show the impact of certain aspects of the proposed approach. Importantly, these models show the impact of certain design decisions (see Chap. 4). For instance, confidential smart contracts, splitting of smart contracts, endorsement policies, and replication of peers.

6.4.1. Attacker Models

Importantly, multiple actors are participating in the blockchain network. Any actor can act in a malicious way and become an attacker. Actors that are a part of the blockchain network are called *internal attackers*. For instance, the EPC, NOBO, Owner, S1 and S2 could be internal attackers if they were to act in a malicious way. A malicious actor that is not a part of the blockchain network is called an *external actor* [63]. However, it is assumed that participating organizations do not act in a malicious way. For example, the EPC is authorized to store PDC1. Thus, he can simply access the data assets inside of the PDC and share them without the Owner's consent. However, it is assumed that the actors don't share the data assets out of band or try to manipulate the transaction flow. This is because these organizations have a preexisting *business relationship* and trust each other to this degree. In contrast, this trust can't be given to customers in a digital rights management (DRM) use case for streaming services. However, this assumption is taken for the fabrication stage use case and only *external attackers* are considered.

Further, to show multiple attack scenarios lets first look at the different components that are involved. The **clients** that are shown in Fig. 4.2 are able *invoke* transactions. They can send transaction proposals to peers and receive proposal responses. Thus, they rely on peers to execute transactions. Hence they only provide inputs and receive outputs of transactions. Importantly, they have no further influence on the execution of the smart contract functions, and they cannot access the code of the smart contracts.

Next, **peers** which are shown in Fig 4.5 endorse transactions. Importantly, they have access to the smart contract code, the ledger, and the private data collections. Importantly, they can refuse to execute transaction proposals and send malicious proposal responses.

Lastly, the **ordering service** is shown in Fig. 4.6 and Fig. 4.7. It consists of ordering service nodes OSNs which all have access to the ledger and actively take part in consensus algorithm. An OSN can refuse to add transactions to blocks or stop participating in the consensus algorithm altogether. However, ordering service node attacks are not considered. This is because the *Raft* implementation is used as the consensus protocol. Most importantly, this protocol is designed to be and is crash fault tolerant (CFT) (see Sec. 3.1.2). Consequently, it is not expected to handle a malicious actor (Byzantine fault (BF)).

Further, a compromise of the certificate authorities which provide the certificates for each actor is not considered. Putz and Pernul already looked into these kind of attacks which they called *Identity Provider Compromise* [63]. In addition, the PKI is not the focus of this thesis.

6.4.1.1. Peer Attacker

The *Peer Attacker* describes an attacker that controls one or multiple peers of the blockchain network. It can control the actions of that peer. For instance, it can choose whether to endorse transaction proposals or not. In addition, it can send malicious responses as an endorsement response. The peer attacker can communicate between the malicious peers that they control. For instance, they can orchestrate 2 peers to send malicious proposal responses. The result is that the chaincode is not necessarily executed and arbitrary changes are made to the world state and private data collections. Also, the peer attacker has access to all ledgers, PDCs, and chaincodes that the controlled peers host.

6.4.1.2. Client Attacker

The *Client Attacker* describes a malicious entity that can take control of an organization's client. Thus, it can use this client to propose smart contract invocations. In addition, it can send these transaction proposals to the ordering service to change the state of the ledger. Importantly, it cannot access the blockchain, world state, chaincode program code, and CC directly. Its access is limited by the chaincode's program code. Therefore, only the data that is accessible through chaincode queries or invocations is available to the *Client Attacker*.

6.4.2. Evaluation

This section evaluates the attacker models that were defined in Sec. 6.4.1.1 and Sec. 6.4.1.2.

6.4.2.1. Peer Attacker

Table 6.4 illustrates how the *Peer Attacker* can block the execution of smart contracts. Importantly, it shows 9 different scenarios. Each scenario lists a unique set of peers that are simultaneously controlled by a *Peer Attacker*. The following enumeration comments on the most important cases.

1,3,4: NOBO's, the Owner's, and S1's peer are attacked on their own. The result is that the endorsement policies for the Evaluation Contract, Cabinet Contract, and Sensor Contract cannot be satisfied if the attacker *refuses* to endorse transactions. However, only one contract is blocked in all of these cases. For instance, in Nr.1 NOBO is attacked. Hence, the Evaluation Contract which needs endorsements from

NOBO'S peer may be blocked. However, the Sensor Contract and the Evaluation Contract work as intended because NOBO's peer does not influence their execution.

2: EPC's peer is compromised in this case. Importantly, EPC's peer must endorse all smart contracts. Consequently, the attacker can block every contract.

8: NOBO's, the Owner's, and S1's peer are marked with (✓). This is to illustrate that the attack on EPC's peer alone has the same effect on the smart contracts.

9: This case illustrates that the attacker must compromise 3 peers that are not the EPC's peer to get the same result. Thus, the concept of Chap. 4 leads peers which are more or less important for the attacker. Importantly, EPC's peer is the most valuable target if the blockage of contracts is desired.

Nr.	Attacker				Contracts blocked		
	NOBO	EPC	Owner	S1	Cabinet	Sensor	Evaluation
1	✓						✓
2		✓			✓	✓	✓
3			✓		✓		
4				✓		✓	
5	✓		✓		✓		✓
6	✓			✓		✓	✓
7			✓	✓	✓	✓	
8	(✓)	✓	(✓)	(✓)	✓	✓	✓
9	✓		✓	✓	✓	✓	✓

Table 6.4.: Peer attacker's ability to block transactions.

Table 6.5 illustrates how the *Peer Attacker's* ability to manipulate smart contracts. For instance, the Cabinet Contract requires two endorsements. One from the EPC's peer and one from the Owner's peer. Hence, the peer attacker that controls both peers can send 2 endorsements that don't reflect the actual smart contract. Hence, it can be said that the attacker can *manipulate* smart contracts. Next, the most important cases are described in more detail in the following enumeration.

1-4: The attacker only controls one peer in all of these cases. Importantly, the endorsement policies from Tab. 4.4 always require **two** endorsements. Thus, no manipulation of smart contracts is possible whenever only **one** peer is compromised.

5-7: In these scenarios the endorsement policies of one contract and the set of attacked peers are congruent. Thus, exactly one contract is manipulable in each of these scenarios.

8: EPC's peer is not compromised in this scenario. However, EPC's peer is a part of every endorsement policy. Therefore, the attacker cannot compromise any contract without EPC's peer.

9: If all peers are compromised that all smart contracts are compromised.

Nr.	Attacker				Contracts manipulated		
	NOBO	EPC	Owner	S1	Cabinet	Sensor	Evaluation
1	✓						
2		✓					
3			✓				
4				✓			
5	✓	✓					✓
6		✓	✓		✓		
7		✓		✓		✓	
8	✓		✓	✓			
9	✓	✓	✓	✓	✓	✓	✓

Table 6.5.: Peer attacker's ability to manipulate smart contracts.

6.4.2.2. Client Attacker

Nr	Attacker					Transaction name
	NOBO	EPC	Owner	S1	S2	
1			✓			request_cabinet_offer
2		✓				send_cabinet_offer
3			✓			accept_cabinet_offer
4		✓				request_pressure_sensor_offer
5				✓		send_pressure_sensor_offer
6		✓				accept_pressure_sensor_offer
7				✓		finish_pressure_sensor_order
8		✓				accept_pressure_sensor_delivery
9		✓				request_evaluation
10	✓					accept_evaluation
11	✓					finish_evaluation
12		✓				finish_cabinet_order
13			✓			accept_cabinet_delivery

Table 6.6.: Lists which transaction a Client Attacker can trigger with malicious intent.

All client attackers gain access to data assets according to Tab. 2.1. The attacker can query chaincode functions on peers. The peers recognizance the client's organization and authorize the access based on that. Section 5.1.3 showed how the authorization is done inside smart contracts. In addition, the clients may invoke transactions in the name of

the organizations. Table 6.6 shows all possible transactions that are directly involved in the workflow execution. Further, the client attacker may also gain access to audit trails and revoke or grant access. However, this is out of scope.

Nevertheless, the client attacker may trigger workflow steps as one organization which might have implications for other organizations. For instance, attack Nr. 11 in Tab. 6.6. The attacker compromised NOBO's client. Thus, it is able to invoke the *finish_evaluation_request* instead of an actual employee of NOBO. Thus, an audit report is committed to the ledger which may contain lies or misinformation. Further, this audit report is then made accessible to the Owner through the finish_cabinet_order transaction. The result is, that the Owner might reject or accept the delivery of the cabinet on the basis of an audit report which was not actually produced by NOBO. In brief, the compromise of a Client can have far reaching consequences based on the use case.

6.4.3. Conclusion

The evaluation of the two attacker models showed that multiple attacks are possible. Importantly, the *Client Attacker* model showed that each organization has to handle the access to their clients with care. Further, the evaluation showed that the *authorization* can enforce access control even if the client is malicious. Further, Putz and Pernul noted that a Client Attacker can flood endorsing peers with a large number of transaction proposals [63]. Each transaction proposal leads to computations on the endorsing peer. Also, the transactions might be sent to the ordering service. Consequently, they will lead to blocks being created and distributed. Therefore, low transaction throughput and high transaction latency may be the result [63].

Furthermore, the evaluation of the *Peer Attacker* showed how the interplay of endorsement policies, chaincode deployment and number of peers can lead to manipulable or blockable smart contracts. Importantly, the desire to have *confidential smart contracts* resulted in 3 different smart contracts. In addition, the deployment of these contracts was limited to the peers which endorse the transactions. Thus, the evaluation of the *Peer Attacker* in particular lead to the following insights:

- *Multiple Peers*: Table 6.4 showed that an attack on a single peer can lead to a blockage of a smart contract. Hence, it is advisable to have multiple peers for each organization. Most importantly, the EPC should maintain multiple peers because an attack on its peer might result in the blockage of all smart contracts. For instance, if the EPC has $n = 6$ peers running the Cabinet Contract than the attacker would have to compromise all of EPC's peers to block the execution of the Cabinet Contract. Hence, a peer attacker must compromise more peers if the organization is running

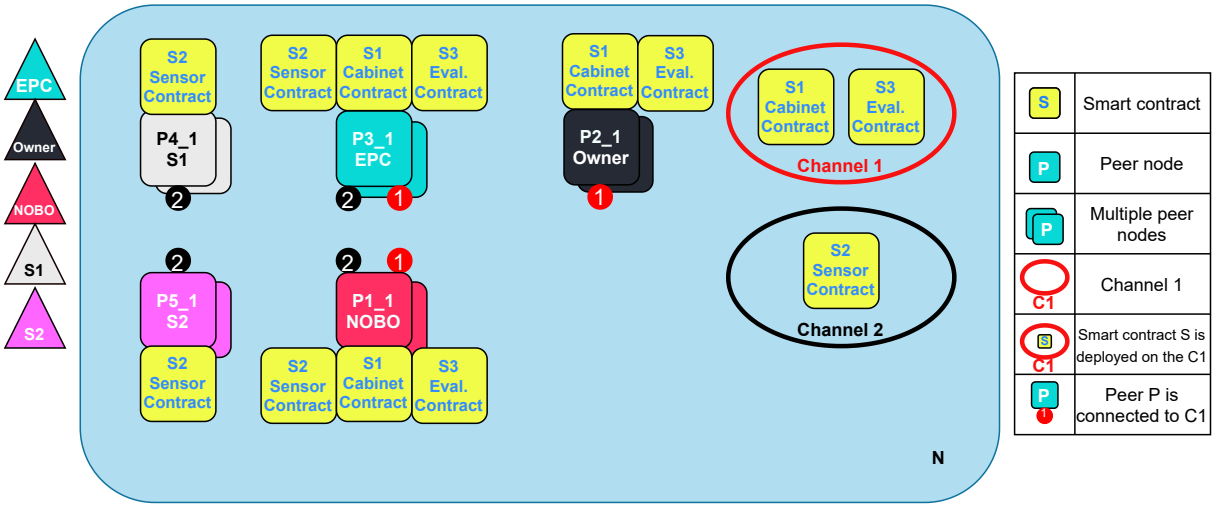

Figure 6.1.: Channel diagram for the Workflow System.

multiple peers. Thus, it can be said that more peers protect against blocked smart contracts.

- *Updated Endorsement*: Table 6.5 illustrates that if the set of peers that is attacked is congruent with the endorsement policy of a smart contract than this enables the attacker to manipulation a smart contract. Consequently, endorsement policies that require more peers to endorse lead to smart contracts which are harder to manipulated. The reason for this is that the peer attacker must compromise more peers in order to manipulate the smart contact. In contrast, less restrictive endorsement policies make it easier for the attacker to manipulate smart contracts because they don't have to compromise as many peers to satisfy the endorsement policy. For instance, the Cabinet Contract requires **2** endorsements in the concept from Chap. 4. However, an endorsement policy which requires **3** peers forces an attacker that compromises 3 peers to yield the same result.
- *Confidential Contracts*: The concept that is described in Chap. 4 showed where the smart contracts were deployed in Fig 4.5. Importantly, the workflow was split into 3 contracts to protect the business logic inside each contract. However, this resulted in the endorsement policies of Tab. 4.4 which only required 2 endorsements per contract. Therefore, a compromise of 2 peers results in a possible manipulation. However, if the smart contracts were deployed to more peers and the endorsement policy were to be altered as to require more endorsements than the attacker would have to compromise more peers to be able to manipulate smart contracts.

Now, the gained insights were used to propose an *updated* concept which aims to complicate the *Peer Attacker's* job to block or manipulate smart contracts. Figure 6.1 shows that

there might be multiple peers deployed for each organization. The exact number depends on the amount of redundancy that each organization is willing to tolerate. In addition, the addition of multiple peers may result in a less homogeneous view of the blockchain across all peers. The result may be more multi-version concurrency check (MVCC) collisions. Importantly, the *Peer Attacker* has to compromise all peers to block the smart contract execution.

Contract	Endorsement Policies
Cabinet	**AND**("EPC.peer", "Owner.peer", "NOBO.peer")
PDC1	**AND**("EPC.peer", "Owner.peer")
Sensor	**AND**("EPC.peer", "S1.peer", "S2.peer", "NOBO.peer")
PDC2	**AND**("EPC.peer","S1.peer")
Evaluation	**AND**("EPC.peer", "NOBO.peer", "Owner.peer")

Table 6.7.: Updated smart contract Endorsement Policies.

Further, the deployment of the smart contracts has changed in comparison with the original Fig. 4.5. In particular, the smart contracts are now deployed on all peers that participate in the channel. For instance, the Sensor Contract is deployed on channel 2. The result is that it is now deployed to S1's, S2'S, EPC's and NOBO's peers. Consequently, the business logic of these contracts is now *shared* with the whole channel. The result is that the Cabinet, Sensor and Evaluation Contract are not confidential contracts anymore. However, this enables the new more *restrictive* endorsement policies see Tab. 6.7. The result is that these more restrictive endorsement policies make it harder for the *Peer Attacker* to manipulate the Evaluation Contract. The Cabinet and Sensor Contract contain a private data collection (PDC). Consequently, transactions that write in the PDC only require the endorsement of these respective policies. Importantly, it is unclear whether the collection level endorsement policy can be used by a Peer Attacker to compromise arbitrary keys in the world state of the respective smart contract.

In brief, the updated proposed approach is a good solution for the fabrication stage use case. Even though attack scenarios are shown and a comparable solution that uses a single trusted central authority is more vulnerable to a peer attacker, because a single compromised node halts the complete workflow. However, the proposed solution only sees at most a halt of a sub workflow. Thus, the added complexity of the blockchain results no single point of failure which is arguably favorable.

7. Conclusion and Future Work

Multilateral business processes require organizations to communicate across their organizational boundaries. Consequently, trust in the correct execution of this process has to be established between the organizations. This thesis demonstrated that a multilateral distributed business process, the fabrication stage use case, can be realized through the utilization of Hyperledger Fabric (HLF). The solution satisfies the requirements which means that all necessary functionality is provided by HLF. Moreover, a multilateral business process may require confidential data assets, transactions, and smart contracts. Importantly, the channels, private data collections and the *execute-order-validate* paradigm of HLF enable this technology to satisfy the aforementioned requirements. This thesis presented a concept for the implementation of the fabrication stage use case. Further, a prototype of the smart contracts was created. Finally, an evaluation based on use case specific attacker models showed the risks when using a confidential smart contract and a PDC. In particular, the implementation of confidential smart contracts is prone to lax endorsement policies. Hence, the resulting confidential smart contracts can only be required to be endorsed by a relatively small amount of peer nodes. Moreover, the evaluation with the peer attacker model is capable of manipulating or blocking smart contract execution. Further, the same problem arises when a PDC is used as it also reduces the amount of peers that can be required to endorse transactions which leads to the same vulnerability. However, an updated design mitigated the risks of using a PDC and reduced smart contract deployments. In particular, the usage of multiple peers per organization will reduce the risk of blocked transactions. Also, to reduce the risk of manipulated smart contracts the deployment of smart contracts was expanded to multiple organizations which can then be required to endorse such transactions. In conclusion, the updated concept mitigates the aforementioned vulnerabilities and any single point of failure that a conventional centralized solution would have is removed. Thus, the use of HLF to implement this kind of multilateral distributed workflow is recommended.

A few questions remain unanswered and their investigation could enhance the results of this thesis. In particular, the question remains if lax collection level endorsement policies for private data collections can be utilized by an attacker to change the whole private data collection, chaincode's namespace or the whole world state. In particular, it is of interest if the weakest endorsement policy can be used by an attacker to change arbitrary

entries of the world state. Further, this may be investigated for key and collection level endorsement policies as well. The evaluation may require carrying out an actual attack on peers and creating malicious proposal responses. In addition, a comparison of HLF for workflow execution and existing workflow execution systems could result in additional arguments for or against the use of blockchain in this field.

Also, this thesis proposed that organizations maintain multiple peers to reduce the risk of an attacker blocking the execution of a smart contract by taking over peers. However, maintaining multiple peers may lead to a less homogeneous view of the blockchain. Consequently, more peers per organization could lead to more invalid transaction due to MVCC collisions in the validation phase. Hence, the impact of multiple peers per organization on the transaction throughput and on the amount of MVCC collisions should be investigated. Moreover, this information could aid the decision making when it comes to the number of peers per organization.

Bibliography

[1] Frequently asked questions, 2020. URL https://consensys.net/quorum/faq/.

[2] Hyperledger burrow, 2020. URL https://github.com/hyperledger/burrow.

[3] Hyperledger fabric 2.2, 2020. URL https://hyperledger-fabric.readthedocs.io/en/release-2.2.

[4] Consensys quorum, 2020. URL https://github.com/ConsenSys/quorum.

[5] Hyperledger fabric 2.3, 2021. URL https://hyperledger-fabric.readthedocs.io/en/release-2.3.

[6] Consensys quorum private transaction lifecycle, 2021. URL https://docs.goquorum.consensys.net/en/stable/Concepts/Privacy/PrivateTransactionLifecycle/.

[7] A library for zero knowledge (zk) scalable transparent argument of knowledge (stark), 2021. URL https://github.com/elibensasson/libSTARK.

[8] P. W. Abreu, M. Aparicio, and C. J. Costa. Blockchain technology in the auditing environment. In *2018 13th Iberian Conference on Information Systems and Technologies (CISTI)*, pages 1–6, 2018. doi: 10.23919/CISTI.2018.8399460.

[9] C. C. Agbo and Q. H. Mahmoud. Comparison of blockchain frameworks for healthcare applications. *Internet Technol. Lett.*, 2(5), 2019. doi: 10.1002/itl2.122. URL https://doi.org/10.1002/itl2.122.

[10] A. Alharbi, H. Zamzami, and E. Samkri. Survey on homomorphic encryption and address of new trend. *Int. J. Adv. Comput. Sci. Appl*, 11(7):618–626, 2020.

[11] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. D. Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolic, S. W. Cocco, and J. Yellick. Hyperledger fabric: a distributed operating system for permissioned blockchains. In R. Oliveira, P. Felber, and Y. C. Hu, editors, *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal,*

April 23-26, 2018, pages 30:1–30:15. ACM, 2018. doi: 10.1145/3190508.3190538. URL https://doi.org/10.1145/3190508.3190538.

[12] E. Androulaki, A. D. Caro, M. Neugschwandtner, and A. Sorniotti. Endorsement in hyperledger fabric. In *IEEE International Conference on Blockchain, Blockchain 2019, Atlanta, GA, USA, July 14-17, 2019*, pages 510–519. IEEE, 2019. doi: 10.1109/Blockchain.2019.00077. URL https://doi.org/10.1109/Blockchain.2019.00077.

[13] D. W. Archer, D. Bogdanov, B. Pinkas, and P. Pullonen. Maturity and performance of programmable secure computation. *IEEE Secur. Priv.*, 14(5):48–56, 2016. doi: 10.1109/MSP.2016.97. URL https://doi.org/10.1109/MSP.2016.97.

[14] N. Arvidsson. *Building a Cashless Society*. Springer International Publishing, 2019. doi: 10.1007/978-3-030-10689-8. URL https://doi.org/10.1007%2F978-3-030-10689-8.

[15] E. Bagdasaryan, G. Berlstein, J. Waterman, E. Birrell, N. Foster, F. B. Schneider, and D. Estrin. Ancile: Enhancing privacy for ubiquitous computing with use-based privacy. In L. Cavallaro, J. Kinder, and J. Domingo-Ferrer, editors, *Proceedings of the 18th ACM Workshop on Privacy in the Electronic Society, WPES@CCS 2019, London, UK, November 11, 2019*, pages 111–124. ACM, 2019. doi: 10.1145/3338498.3358642. URL https://doi.org/10.1145/3338498.3358642.

[16] A. Baliga, I. Subhod, P. Kamat, and S. Chatterjee. Performance evaluation of the quorum blockchain platform. *CoRR*, abs/1809.03421, 2018. URL http://arxiv.org/abs/1809.03421.

[17] M. A. Barbara. Proof of all: Verifiable computation in a nutshell. *CoRR*, abs/1908.02327, 2019. URL http://arxiv.org/abs/1908.02327.

[18] M. Bartoletti and L. Pompianu. An empirical analysis of smart contracts: Platforms, applications, and design patterns. In M. Brenner, K. Rohloff, J. Bonneau, A. Miller, P. Y. A. Ryan, V. Teague, A. Bracciali, M. Sala, F. Pintore, and M. Jakobsson, editors, *Financial Cryptography and Data Security - FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers*, volume 10323 of *Lecture Notes in Computer Science*, pages 494–509. Springer, 2017. doi: 10.1007/978-3-319-70278-0_31. URL https://doi.org/10.1007/978-3-319-70278-0_31.

[19] D. M. Beazley et al. Swig: An easy to use tool for integrating scripting languages with c and c++. In *Tcl/Tk Workshop*, volume 43, page 74, 1996.

[20] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. Snarks for C: verifying program executions succinctly and in zero knowledge. In R. Canetti and J. A. Garay, editors, *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II*, volume 8043 of *Lecture Notes in Computer Science*, pages 90–108. Springer, 2013. doi: 10.1007/978-3-642-40084-1_6. URL https://doi.org/10.1007/978-3-642-40084-1_6.

[21] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 459–474. IEEE Computer Society, 2014. doi: 10.1109/SP.2014.36. URL https://doi.org/10.1109/SP.2014.36.

[22] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *IACR Cryptol. ePrint Arch.*, 2018:46, 2018. URL http://eprint.iacr.org/2018/046.

[23] F. Benhamouda, S. Halevi, and T. Halevi. Supporting private data on hyperledger fabric with secure multiparty computation. *IBM J. Res. Dev.*, 63(2/3):3:1–3:8, 2019. doi: 10.1147/JRD.2019.2913621. URL https://doi.org/10.1147/JRD.2019.2913621.

[24] J. B. Bernabé, J. L. Cánovas, J. L. H. Ramos, R. T. Moreno, and A. F. Skarmeta. Privacy-preserving solutions for blockchain: Review and challenges. *IEEE Access*, 7: 164908–164940, 2019. doi: 10.1109/ACCESS.2019.2950872. URL https://doi.org/10.1109/ACCESS.2019.2950872.

[25] M. Brandenburger, C. Cachin, R. Kapitza, and A. Sorniotti. Blockchain and trusted computing: Problems, pitfalls, and a solution for hyperledger fabric. *CoRR*, abs/1805.08541, 2018. URL http://arxiv.org/abs/1805.08541.

[26] M. Brenner, W. Dai, S. Halevi, K. Han, A. Jalali, M. Kim, K. Laine, A. Malozemoff, P. Paillier, Y. Polyakov, et al. A standard api for rlwe-based homomorphic encryption. Technical report, Technical Report. HomomorphicEncryption. org, Redmond WA, USA, 2017.

[27] E. Brickell and J. Li. Enhanced privacy ID from bilinear pairing for hardware authentication and attestation. *Int. J. Inf. Priv. Secur. Integr.*, 1(1):3–33, 2011. doi: 10.1504/IJIPSI.2011.043729. URL https://doi.org/10.1504/IJIPSI.2011.043729.

[28] R. G. Brown, J. Carlyle, I. Grigg, and M. Hearn. Corda: an introduction. *R3 CEV, August*, 1:15, 2016.

[29] B. Bünz, S. Agrawal, M. Zamani, and D. Boneh. Zether: Towards privacy in a smart contract world. In J. Bonneau and N. Heninger, editors, *Financial Cryptography and Data Security - 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10-14, 2020 Revised Selected Papers*, volume 12059 of *Lecture Notes in Computer Science*, pages 423–443. Springer, 2020. doi: 10.1007/978-3-030-51280-4_23. URL https://doi.org/10.1007/978-3-030-51280-4_23.

[30] M. Castro and B. Liskov. Practical byzantine fault tolerance. In M. I. Seltzer and P. J. Leach, editors, *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999*, pages 173–186. USENIX Association, 1999. URL https://dl.acm.org/citation.cfm?id=296824.

[31] M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, J. Hoffstein, K. Lauter, S. Lokam, D. Moody, T. Morrison, et al. Security of homomorphic encryption. *HomomorphicEncryption. org, Redmond WA, Tech. Rep*, 2017.

[32] L. Chen, S. Jordan, Y.-K. Liu, D. Moody, R. Peralta, R. Perlner, and D. Smith-Tone. Report on post-quantum cryptography, 2016-04-28 2016.

[33] L. Chen, K. Chen, S. Zhong, and D. Ye. Privacy protection method of document management based on homomorphic encryption on the fabric platform. In *Proceedings of the 2019 2nd International Conference on Blockchain Technology and Applications*, ICBTA 2019, page 31–37, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450377430. doi: 10.1145/3376044.3376063. URL https://doi.org/10.1145/3376044.3376063.

[34] K. R. Choo, A. Dehghantanha, and R. M. Parizi, editors. *Blockchain Cybersecurity, Trust and Privacy*, volume 79 of *Advances in Information Security*. Springer, 2020. ISBN 978-3-030-38180-6. doi: 10.1007/978-3-030-38181-3. URL https://doi.org/10.1007/978-3-030-38181-3.

[35] V. Costan and S. Devadas. Intel SGX explained. *IACR Cryptol. ePrint Arch.*, 2016: 86, 2016. URL http://eprint.iacr.org/2016/086.

[36] I. Damgård. *Commitment Schemes and Zero-Knowledge Protocols*, pages 63–86. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. ISBN 978-3-540-48969-6. doi: 10.1007/3-540-48969-X_3. URL https://doi.org/10.1007/3-540-48969-X_3.

[37] D. Evans, V. Kolesnikov, and M. Rosulek. A pragmatic introduction to secure multi-party computation. *Found. Trends Priv. Secur.*, 2(2-3):70–246, 2018. doi: 10.1561/3300000019. URL https://doi.org/10.1561/3300000019.

[38] M. Ghadamyari and S. Samet. Privacy-preserving statistical analysis of health data using paillier homomorphic encryption and permissioned blockchain. In *2019 IEEE International Conference on Big Data (Big Data), Los Angeles, CA, USA, December 9-12, 2019*, pages 5474–5479. IEEE, 2019. doi: 10.1109/BigData47090.2019.9006231. URL https://doi.org/10.1109/BigData47090.2019.9006231.

[39] G. Giuffra. Scalable, transparent, and post-quantum secure computational integrity. 2019.

[40] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof-systems (extended abstract). In R. Sedgewick, editor, *Proceedings of the 17th Annual ACM Symposium on Theory of Computing, May 6-8, 1985, Providence, Rhode Island, USA*, pages 291–304. ACM, 1985. doi: 10.1145/22145.22178. URL https://doi.org/10.1145/22145.22178.

[41] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis. Madmax: analyzing the out-of-gas world of smart contracts. *Commun. ACM*, 63(10):87–95, 2020. doi: 10.1145/3416262. URL https://doi.org/10.1145/3416262.

[42] I. Grigg. The ricardian contract. In *Proceedings. First IEEE International Workshop on Electronic Contracting, 2004.*, pages 25–31. IEEE, 2004.

[43] T. Härder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, 1983. doi: 10.1145/289.291. URL https://doi.org/10.1145/289.291.

[44] N. Kannengießer, S. Lins, T. Dehling, and A. Sunyaev. What does not fit can be made to fit! trade-offs in distributed ledger technology designs. In T. Bui, editor, *52nd Hawaii International Conference on System Sciences, HICSS 2019, Grand Wailea, Maui, Hawaii, USA, January 8-11, 2019*, pages 1–10. ScholarSpace, 2019. URL http://hdl.handle.net/10125/60143.

[45] N. Khan and M. Nassar. A look into privacy-preserving blockchains. In *16th IEEE/ACS International Conference on Computer Systems and Applications, AICCSA 2019, Abu Dhabi, UAE, November 3-7, 2019*, pages 1–6. IEEE Computer Society, 2019. doi: 10.1109/AICCSA47632.2019.9035235. URL https://doi.org/10.1109/AICCSA47632.2019.9035235.

[46] V. Khinchi. Evaluating various transaction processing characteristics of permissioned blockchain networks. pages 1–103, 2018. doi: http://dx.doi.org/10.18419/opus-10105. URL http://nbn-resolving.de/urn:nbn:de:bsz:93-opus-ds-101223.

[47] T. Kosar and M. Livny. Faults in large distributed systems and what we can do about them. In J. C. Cunha and P. D. Medeiros, editors, *Euro-Par 2005, Parallel Processing, 11th International Euro-Par Conference, Lisbon, Portugal, August 30 - September 2, 2005, Proceedings*, volume 3648 of *Lecture Notes in Computer Science*, pages 442–453. Springer, 2005. doi: 10.1007/11549468_51. URL https://doi.org/10.1007/11549468_51.

[48] R. E. Ladner. On the structure of polynomial time reducibility. *J. ACM*, 22(1):155–171, 1975. doi: 10.1145/321864.321877. URL https://doi.org/10.1145/321864.321877.

[49] Y. Lindell. Secure multiparty computation (MPC). *IACR Cryptol. ePrint Arch.*, 2020:300, 2020. URL https://eprint.iacr.org/2020/300.

[50] X. Liu, B. Farahani, and F. Firouzi. *Distributed Ledger Technology*, pages 393–431. Springer International Publishing, Cham, 2020. ISBN 978-3-030-30367-9. doi: 10.1007/978-3-030-30367-9_8. URL https://doi.org/10.1007/978-3-030-30367-9_8.

[51] M. Louk and H. Lim. Homomorphic encryption in mobile multi cloud computing. In *2015 International Conference on Information Networking, ICOIN 2015, Siem Reap, Cambodia, January 12-14, 2015*, pages 493–497. IEEE Computer Society, 2015. doi: 10.1109/ICOIN.2015.7057954. URL https://doi.org/10.1109/ICOIN.2015.7057954.

[52] C. Ma, X. Kong, Q. Lan, and Z. Zhou. The privacy protection mechanism of hyperledger fabric and its application in supply chain finance. *Cybersecur.*, 2 (1):5, 2019. doi: 10.1186/s42400-019-0022-2. URL https://doi.org/10.1186/s42400-019-0022-2.

[53] B. Medjahed, M. Ouzzani, and A. K. Elmagarmid. Generalization of ACID properties. In L. Liu and M. T. Özsu, editors, *Encyclopedia of Database Systems, Second Edition*. Springer, 2018. doi: 10.1007/978-1-4614-8265-9_736. URL https://doi.org/10.1007/978-1-4614-8265-9_736.

[54] Microsoft. Ibm blockchain platform, 2021. URL https://marketplace.visualstudio.com/items?itemName=IBMBlockchain.ibm-blockchain-platform.

[55] Microsoft. Visual studio code, 2021. URL https://code.visualstudio.com.

[56] S. Nakamoto. A peer-to-peer electronic cash system. *Bitcoin.–URL: https://bitcoin.org/bitcoin. pdf*, 4, 2008.

[57] A. Narayanan, J. Bonneau, E. W. Felten, A. Miller, and S. Goldfeder. *Bitcoin and Cryptocurrency Technologies - A Comprehensive Introduction*. Princeton University Press, 2016. URL https://d28rh4a8wq0iu5.cloudfront.net/bitcointech/readings/princeton_bitcoin_book.pdf.

[58] P. L. Noac'h, A. Costan, and L. Bougé. A performance evaluation of apache kafka in support of big data streaming applications. In J. Nie, Z. Obradovic, T. Suzumura, R. Ghosh, R. Nambiar, C. Wang, H. Zang, R. Baeza-Yates, X. Hu, J. Kepner, A. Cuzzocrea, J. Tang, and M. Toyoda, editors, *2017 IEEE International Conference on Big Data, BigData 2017, Boston, MA, USA, December 11-14, 2017*, pages 4803–4806. IEEE Computer Society, 2017. doi: 10.1109/BigData.2017.8258548. URL https://doi.org/10.1109/BigData.2017.8258548.

[59] P. Oechslin. Making a faster cryptanalytic time-memory trade-off. In D. Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 617–630. Springer, 2003. doi: 10.1007/978-3-540-45146-4_36. URL https://doi.org/10.1007/978-3-540-45146-4_36.

[60] D. Ongaro and J. K. Ousterhout. In search of an understandable consensus algorithm. In G. Gibson and N. Zeldovich, editors, *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, pages 305–319. USENIX Association, 2014. URL https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

[61] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In J. Stern, editor, *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer, 1999. doi: 10.1007/3-540-48910-X_16. URL https://doi.org/10.1007/3-540-48910-X_16.

[62] J. Polge, J. Robert, and Y. Le Traon. Permissioned blockchain frameworks in the industry: A comparison. *ICT Express*, 2020.

[63] B. Putz and G. Pernul. Detecting blockchain security threats. In *2020 IEEE International Conference on Blockchain, Blockchain 2020, Rhodes Island, Greece, November 2-6, 2020*, pages 313–320. IEEE, 2020. doi: 10.1109/Blockchain50366.2020.00046. URL https://doi.org/10.1109/Blockchain50366.2020.00046.

[64] M. Saad, J. Spaulding, L. Njilla, C. A. Kamhoua, S. Shetty, D. Nyang, and A. Mohaisen. Exploring the attack surface of blockchain: A systematic overview. *CoRR*, abs/1904.03487, 2019. URL http://arxiv.org/abs/1904.03487.

[65] M. Sabt, M. Achemlal, and A. Bouabdallah. Trusted execution environment: What it is, and what it is not. In *2015 IEEE TrustCom/BigDataSE/ISPA, Helsinki, Finland, August 20-22, 2015, Volume 1*, pages 57–64. IEEE, 2015. doi: 10.1109/Trustcom.2015.357. URL https://doi.org/10.1109/Trustcom.2015.357.

[66] C. Saraf and S. Sabadra. Blockchain platforms: A compendium. In *2018 IEEE International Conference on Innovative Research and Development (ICIRD)*, pages 1–6, 2018. doi: 10.1109/ICIRD.2018.8376323.

[67] F. B. Schneider. Byzantine generals in action: Implementing fail-stop processors. *ACM Trans. Comput. Syst.*, 2(2):145–154, 1984. doi: 10.1145/190.357399. URL https://doi.org/10.1145/190.357399.

[68] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990. doi: 10.1145/98163.98167. URL https://doi.org/10.1145/98163.98167.

[69] SEAL. Microsoft SEAL (release 3.6). https://github.com/Microsoft/SEAL, Nov. 2020. Microsoft Research, Redmond, WA.

[70] A. Sforzin, P. Kasinathan, and M. Wimmer. D5.1 requirements analysis of demonstration cases phase1. pages 30–41, 2020. URL https://cybersec4europe.eu/wp-content/uploads/2020/05/D5.2-Specification-and-Set-up-of-Demonstration-Case-Phase-1-v1.0_Submitted.pdf.

[71] A. Sforzin, M. Wimmer, and P. Kasinathan. D5.2 specification and set-up demonstration case phase 1. pages 50–71, 2020. URL https://cybersec4europe.eu/wp-content/uploads/2020/05/D5.

2-Specification-and-Set-up-of-Demonstration-Case-Phase-1-v1.0_Submitted.pdf.

[72] A. Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979. doi: 10.1145/359168.359176. URL http://doi.acm.org/10.1145/359168.359176.

[73] T. Shimosawa, T. Sato, and S. Oshima. Bcverifier: A tool to verify hyperledger fabric ledgers. In *2020 IEEE International Conference on Blockchain, Blockchain 2020, Rhodes Island, Greece, November 2-6, 2020*, pages 291–299. IEEE, 2020. doi: 10.1109/Blockchain50366.2020.00043. URL https://doi.org/10.1109/Blockchain50366.2020.00043.

[74] L. Q. Simon Stone, Matthew B White. Github - ibm-blockchain/microfab, 2021. URL https://github.com/IBM-Blockchain/microfab.

[75] D. Skeen. Nonblocking commit protocols. In Y. E. Lien, editor, *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data, Ann Arbor, Michigan, USA, April 29 - May 1, 1981*, pages 133–142. ACM Press, 1981. doi: 10.1145/582318.582339. URL https://doi.org/10.1145/582318.582339.

[76] S. Stahnke. Workflow enforcement for certification of the construction of industrial plants. pages 1–123, 2020.

[77] P. J. Taylor, T. Dargahi, A. Dehghantanha, R. M. Parizi, and K.-K. R. Choo. A systematic literature review of blockchain cyber security. *Digital Communications and Networks*, 6(2):147 – 156, 2020. ISSN 2352-8648. doi: https://doi.org/10.1016/j.dcan.2019.01.005. URL http://www.sciencedirect.com/science/article/pii/S2352864818301536.

[78] M. Valenta and P. Sandner. Comparison of ethereum, hyperledger fabric and corda. *no. June*, pages 1–8, 2017.

[79] M. Walport. Distributed ledger technology: beyond block chain. uk government office for science, london. *Haettu osoitteesta https://www. gov. uk/government/publications/distributed-ledgertechnology-blackett-review*, 2016.

[80] D. Yaga, P. Mell, N. Roby, and K. Scarfone. Blockchain technology overview. Oct 2018. doi: 10.6028/nist.ir.8202. URL http://dx.doi.org/10.6028/NIST.IR.8202.

[81] 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), pages 1–10, 2019. doi: 10.1109/IWBOSE.2019.8666486.

[82] A. C. Yao. Protocols for secure computations (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*, pages 160–164. IEEE Computer Society, 1982. doi: 10.1109/SFCS.1982.38. URL https://doi.org/10.1109/SFCS.1982.38.

[83] F. Zhang, W. He, R. Cheng, J. Kos, N. Hynes, N. M. Johnson, A. Juels, A. Miller, and D. Song. The ekiden platform for confidentiality-preserving, trustworthy, and performant smart contracts. *IEEE Secur. Priv.*, 18(3):17–27, 2020. doi: 10.1109/MSEC.2020.2976984. URL https://doi.org/10.1109/MSEC.2020.2976984.

[84] G. Zyskind. The future of enigma: A letter from the ceo. URL https://blog.enigma.co/the-future-of-enigma-a-letter-from-the-ceo-c149cf3b0b11.

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

Appendices

A. Deployment Diagrams for the Fabrication Stage Concept

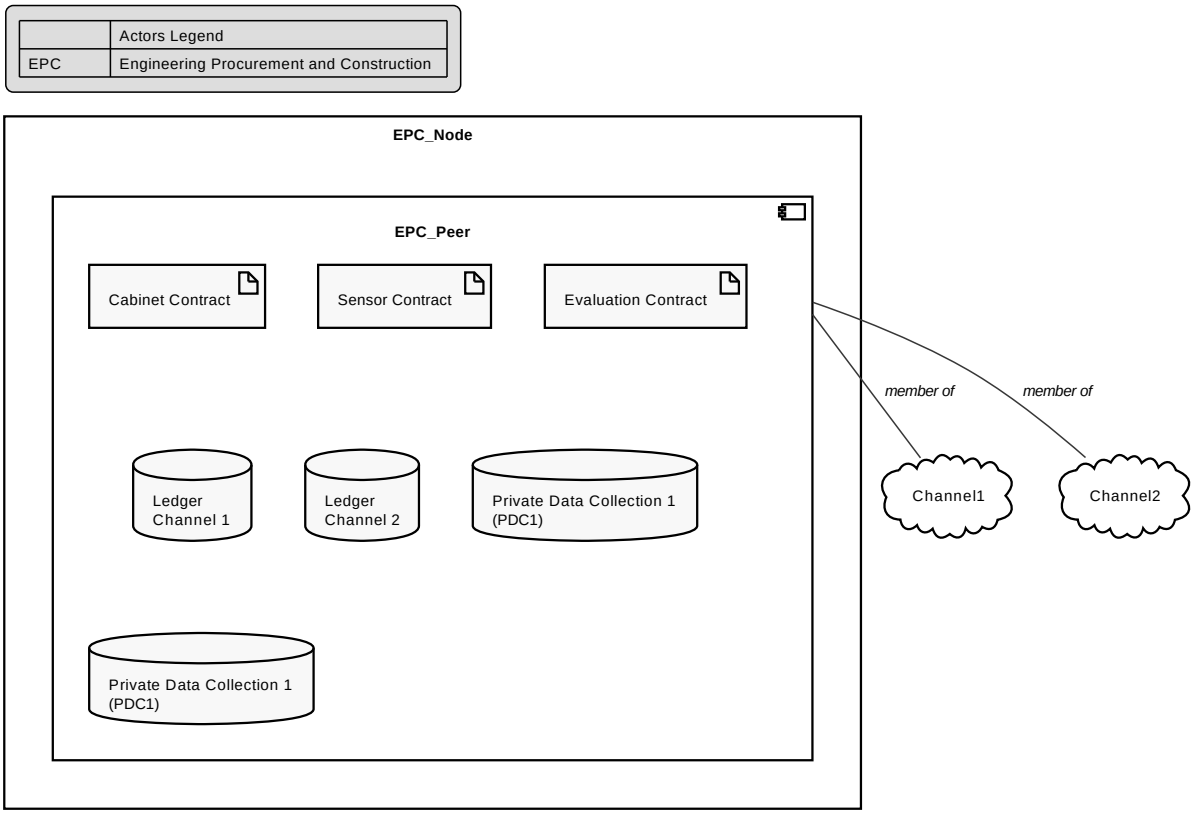

Figure A.1.: Deployment diagram for the EPC's node.

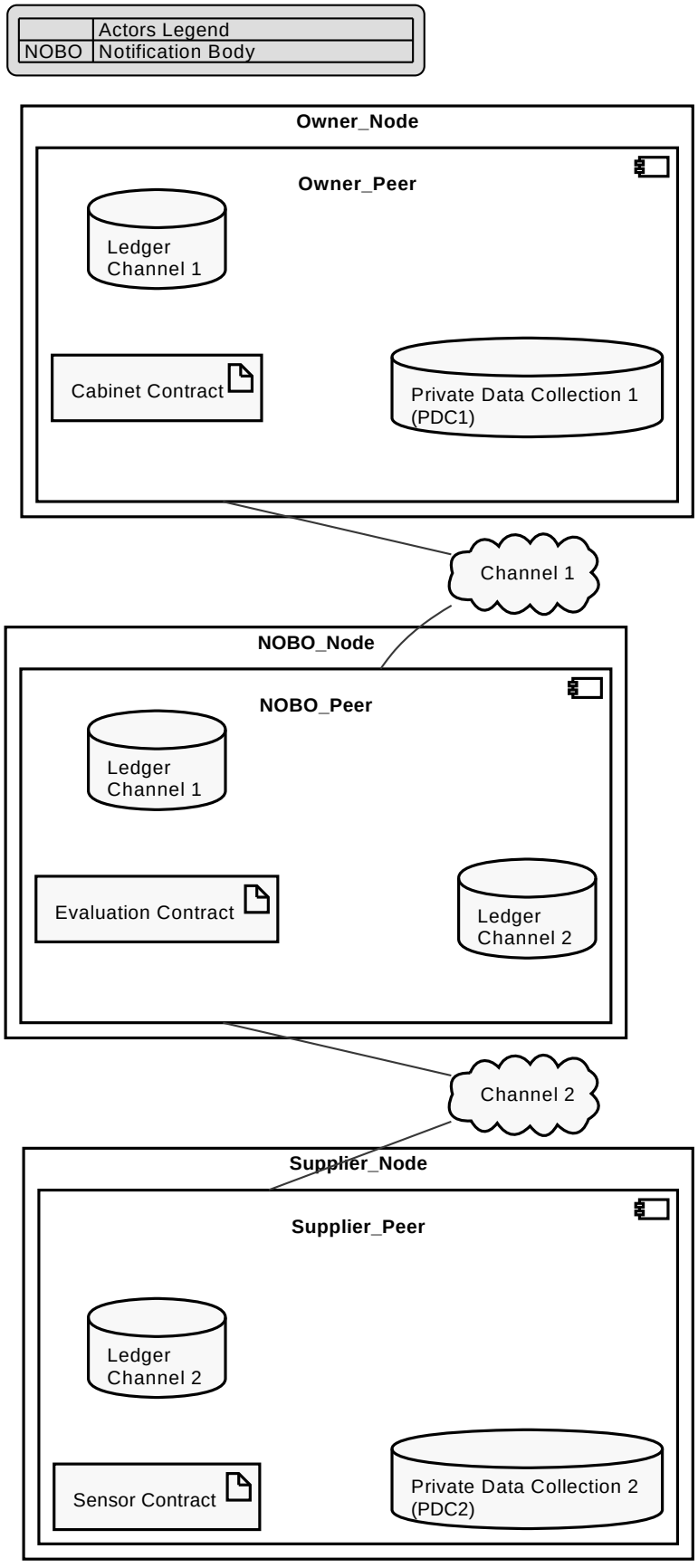

Figure A.2.: Deployment diagram for the nodes of the Owner, Supplier 1, Supplier 2 and NOBO.

B. Sequence Diagrams for the Fabrication Stage Concept

	Actors
EPC	Engineering Procurement and Construction
Owner	Owner of the cabinet
NOBO	Notification Body
S1	Supplier of the pressure sensor
S2	Supplier that is not involved in the cabinet supply chain

	Artifacts
C_	Document is confidential
D	Document is not confidential
D_CDS	Cabinet Design Specification
C_CO	Cabinet Offer
D_CFS	Cabinet Fact Sheet
D_AR	Audit Report
D_OAS	Owner Acceptance Sheet
D_PSS	Pressure Sensor Specification
C_PSO	Pressure Sensor Offer
D_PSFS	Pressure Sensor Fact Sheet
D_EAS	EPC Acceptance Sheet

	Collections	Accessible by
L1	Ledger of channel 1	Owner, EPC, NOBO
PDC1	Private Data Collection 1	Owner, EPC
L2	Ledger of channel 2	EPC, S1, S2, NOBO
PDC2	Private Data Collection 2	EPC, S1

Figure B.1.: Legend for the sequence diagrams.

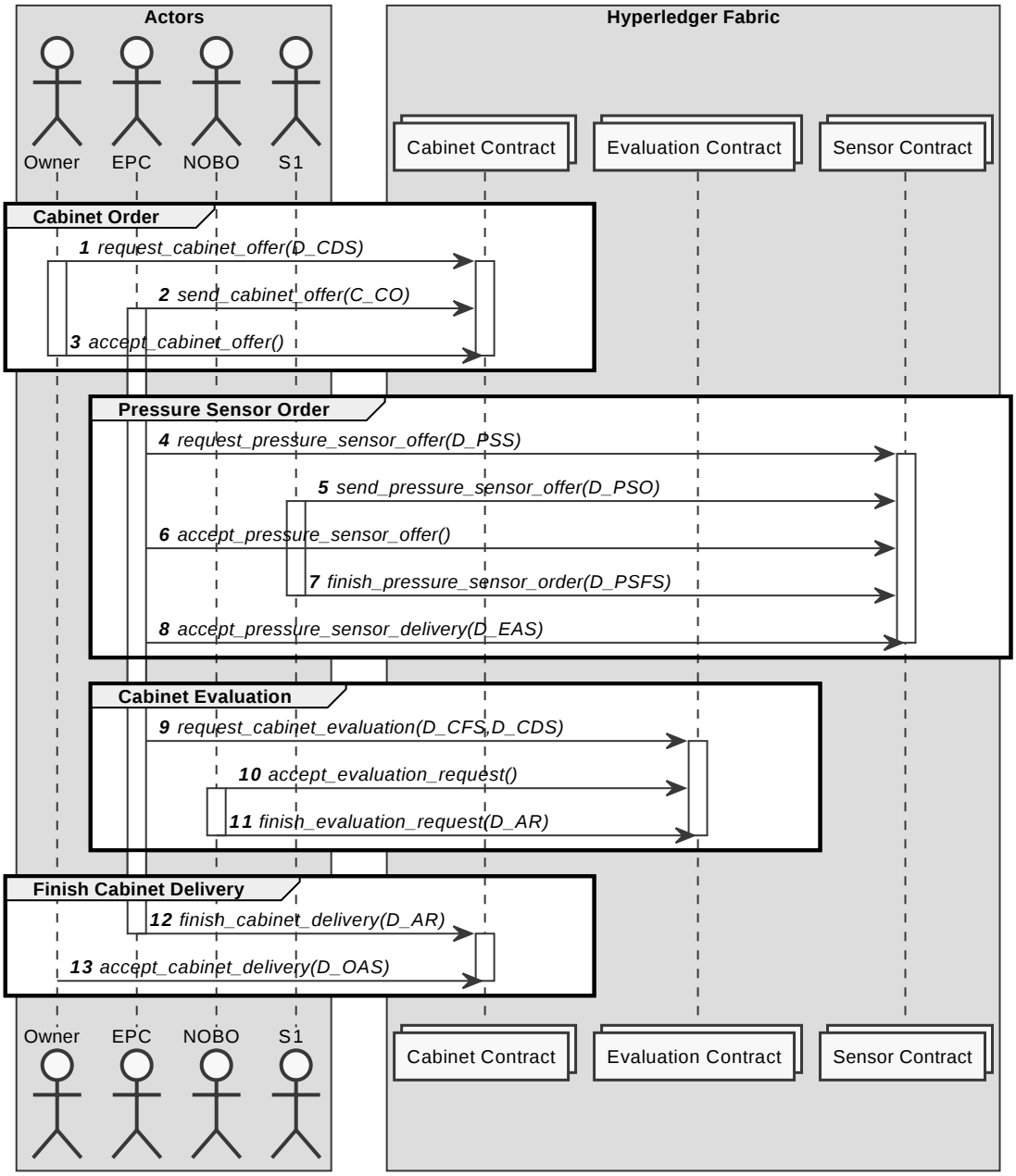

Figure B.2.: Sequence diagram for the Fabrication Stage use case.

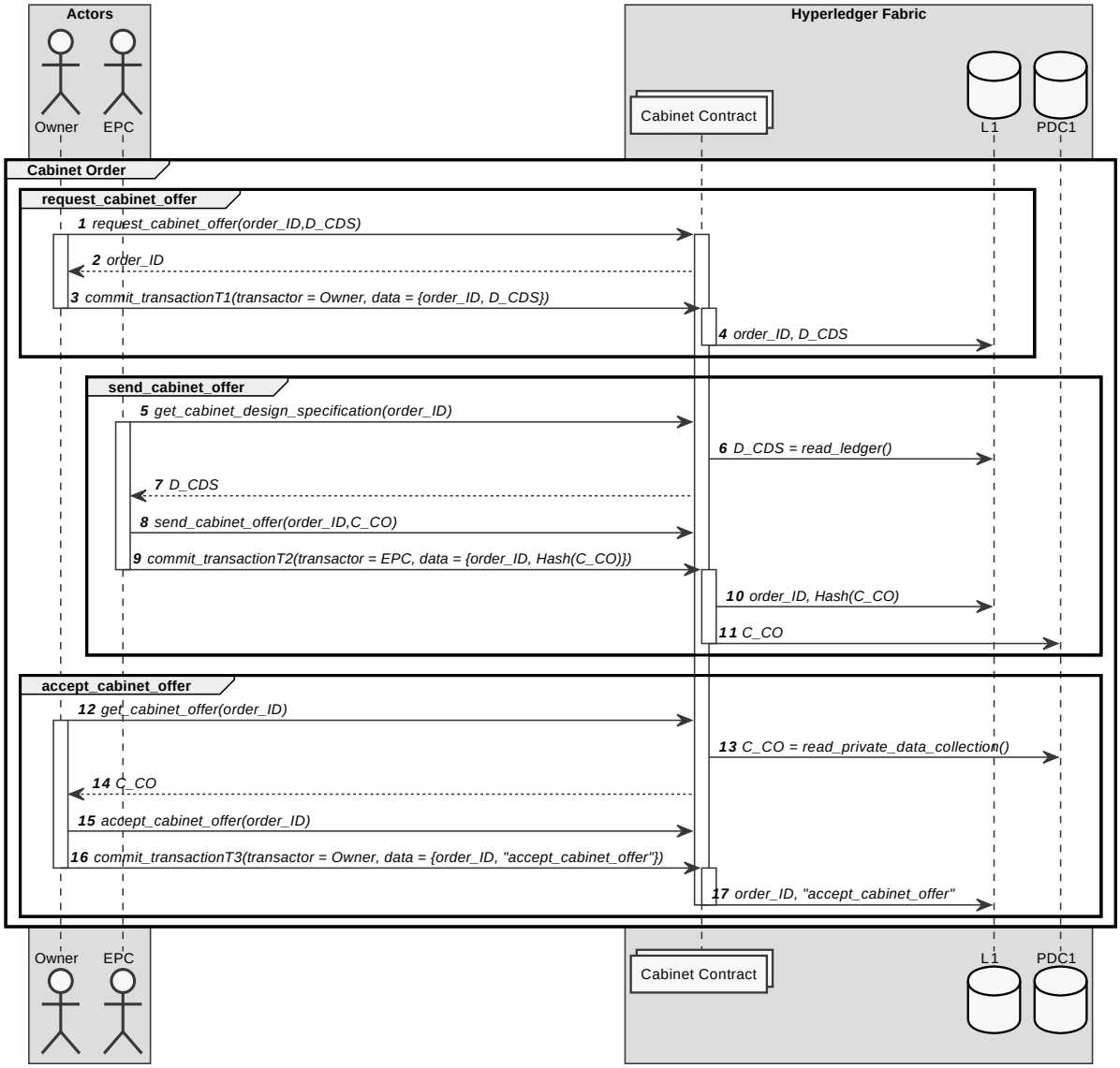

Figure B.3.: Sequence diagram for the Fabrication Stage use case.

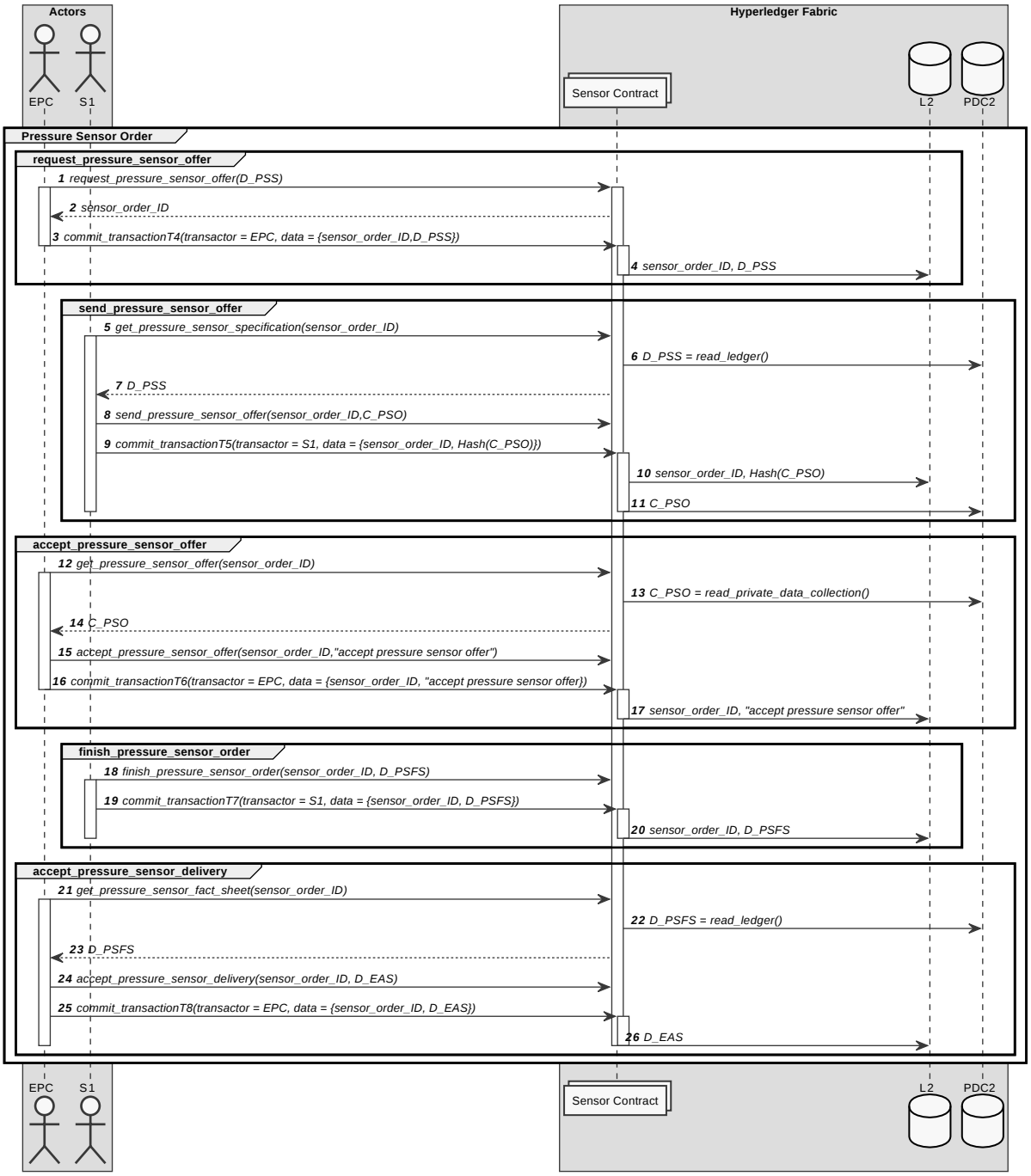

Figure B.4.: Sequence diagram for the Fabrication Stage use case.

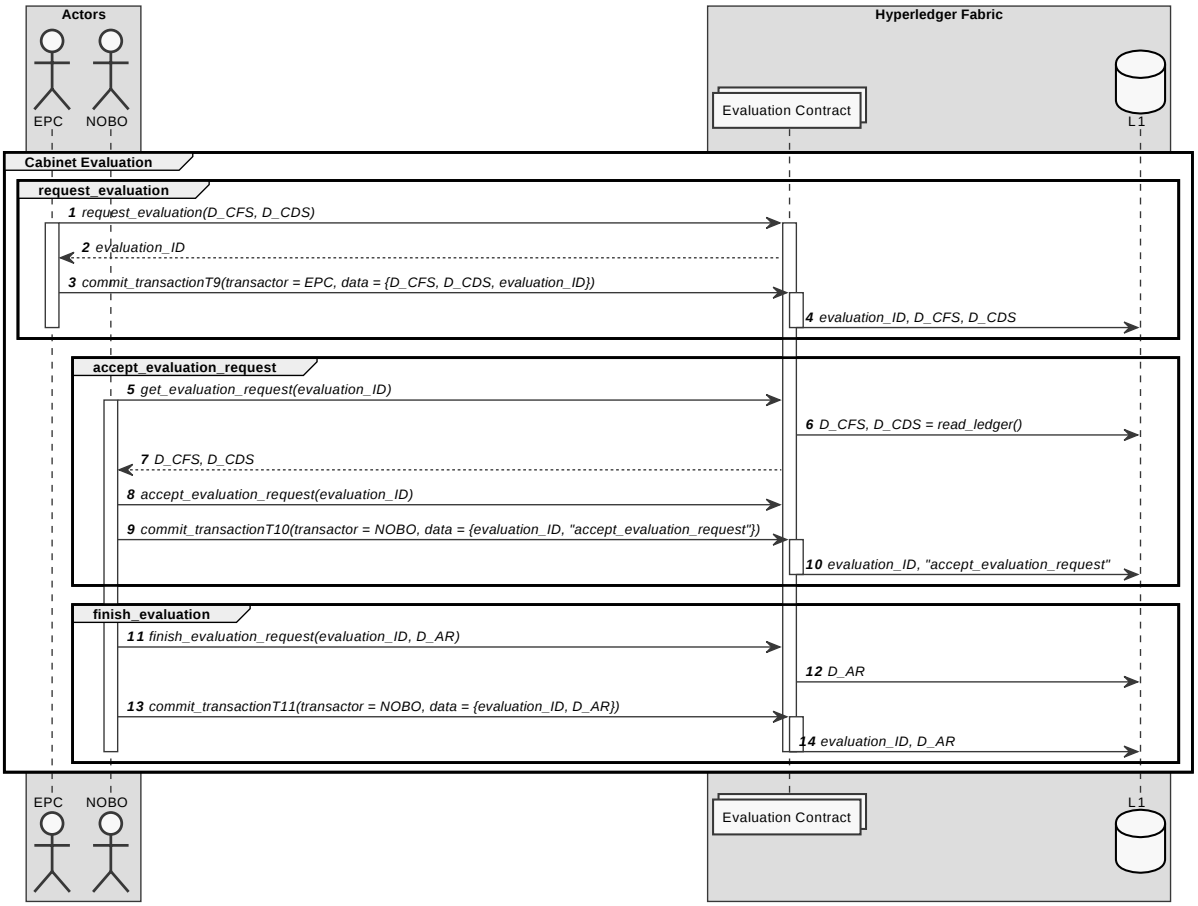

Figure B.5.: Sequence diagram for the Fabrication Stage use case.

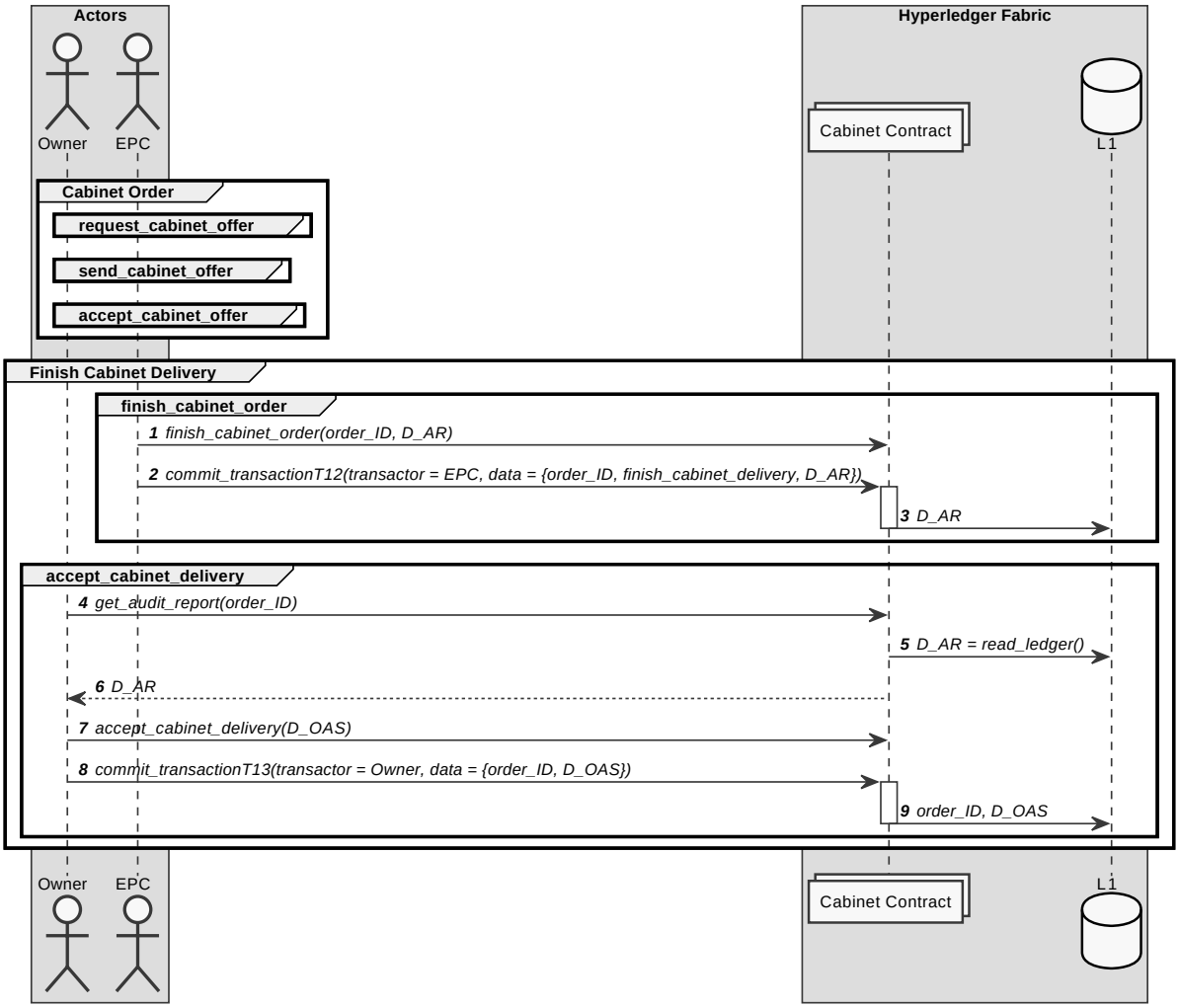

Figure B.6.: Sequence diagram for the Fabrication Stage use case.

C. Implementation

```
{
  "endorsing_organizations":[{
    "name": "EPC"
  },{
    "name": "NOBO"
  },{
    "name": "Owner"
  },{
    "name": "S1"
  },{
    "name": "S2"
  }],
  "channels":[
  {
    "name": "channel1",
    "endorsing_organizations":[
    "EPC",
    "NOBO",
    "Owner"
    ]
  },{
    "name": "channel2",
    "endorsing_organizations":[
    "EPC",
    "NOBO",
    "S1",
    "S2"
    ]
  }]
}
```

Listing C.1: Configuration for the Microfab test network that was used throughout the development process.

```
[
{
  "name": "PDC1",
  "policy": "OR('EPCMSP.member', 'OwnerMSP.member')",
  "requiredPeerCount": 1,
  "maxPeerCount": 1,
  "blockToLive":0,
  "memberOnlyRead": false,
  "memberOnlyWrite": false
}
]
```

Listing C.2: Private Data Collection Definition for the PDC1 that is used by the Cabinet Contract.

```
async beforeTransaction(ctx) {
  const args = ctx.stub.getArgs();
  const transaction_details = ctx.stub.getFunctionAndParameters();
  const audit_trail_id = 'audit' + args[1];
  const buffer = await ctx.stub.getState(audit_trail_id);
  let object;
  if ((!!buffer && buffer.length > 0)) {
    //parse the existing audit trail
    object = JSON.parse(buffer.toString());
  } else {
    //create new audit trail object, because it is the first audit trail
       entry for this order_id
    object = { audit_trail: [], access_control_list:[] };}
  const transientDataMap = ctx.stub.getTransient();
  let transient_data_entry;
  if (transientDataMap.size === 0) {
    transient_data_entry = null;
  }else{
    transient_data_entry = [];
    for (const [key, value] of transientDataMap.entries()) {
      const hashToVerify = crypto.createHash('sha256').update(value.
         toString()).digest('hex');
      transient_data_entry.push([key, hashToVerify]);}}
  const identity = ctx.clientIdentity.getMSPID();
  const formatted_timestamp = new Date((ctx.stub.txTimestamp.seconds *
     1000));
  formatted_timestamp.setMilliseconds(ctx.stub.txTimestamp.nanos /
     1000000);
  const audit_trail_entry = {
    timestamp : formatted_timestamp,
    transaction_id: ctx.stub.txId,
    function_name: transaction_details.fcn,
    arguments: args.slice(1),
    transactor: identity,
    transient_data: transient_data_entry}
  object.audit_trail.push(audit_trail_entry);
  await ctx.stub.putState(audit_trail_id, Buffer.from(JSON.stringify(
     object)));}
```

Listing C.3: beforeTransaction implementation

```
async revoke_audit_trail_access(ctx, order_id, revoke_msp) {
//assert that the cabinet already exists
await assert_cabinet_existence(ctx, order_id, true);

//assert that only the Owner and the EPC can revoke automatic audit trail access
const identity = ctx.clientIdentity.getMSPID();
authorize_transactor(['OwnerMSP', 'EPCMSP'], identity);

//retrieve audit trail object
const audit_trail_id = 'audit' + order_id;
const buffer = await ctx.stub.getState(audit_trail_id);
const object = JSON.parse(buffer.toString());

//add authorized msp to the access control list
object.access_control_list.push(revoke_msp);

//remove revoked msp
object.access_control_list = object.access_control_list.filter(function(item) {
return item !== revoke_msp
})

//update the audit trail object with the new access control list
await ctx.stub.putState(audit_trail_id, Buffer.from(JSON.stringify(object)));
return JSON.stringify(object);
}
```

Listing C.4: revoke_audit_trail_access function

```
async authorize_audit_trail_access(ctx, order_id, authorized_msp) {
  //assert that the cabinet already exists
  await assert_cabinet_existence(ctx, order_id, true);

  //assert that only the Owner and the EPC can authorize automatic audit
     trail access
  const identity = ctx.clientIdentity.getMSPID();
  authorize_transactor(['OwnerMSP', 'EPCMSP'], identity);

  //retrieve audit trail object
  const audit_trail_id = 'audit' + order_id;
  const buffer = await ctx.stub.getState(audit_trail_id);
  const object = JSON.parse(buffer.toString());

  //add authorized msp to the access control list
  object.access_control_list.push(authorized_msp);
  await ctx.stub.putState(audit_trail_id, Buffer.from(JSON.stringify(
     object)));

  return JSON.stringify(object);
}
```

Listing C.5: authorize_audit_trail_access function

D. DVD Contents

1. saved_sources.zip: Archived websites.
2. prototype.zip: Code for the prototype.

Abbreviations

ACID	atomicity, consistency, isolation, durability
AC	access control
AEP	authorized endorsing peer
BFT	Byzantine-fault tolerant
BF	Byzantine fault
CFT	crash fault tolerant
CIP	computational integrity and privacy
CC	channel configuration
CSCC	configuration system chaincode
DLT	distributed ledger technology
DSL	domain specific language
DOS	denial of service
DAG	directed acyclic graph
DRM	digital rights management
EPID	enhanced privacy ID
ESCC	endorsement system chaincode
FHE	fully homomorphic encryption
gRPC	remote procedure call
GDPR	general data protection regulation
HLF	Hyperledger Fabric
HE	homomorphic encryption
IAS	Intel attestation service
IBFT	Istanbul Byzantine fault tolerant
LSCC	lifecycle system chaincode
MSP	membership service provider
MPC	secure multiparty computation
MVCC	multi-version concurrency check
OSN	ordering service nodes
POW	proof of work
PDC	private data collection
PBFT	practical Byzantine fault tolerance
PKI	public key infrastructure
POI	proof of identity
PoET	proof of elapsed time
PHE	partially homomorphic encryption
SAT	satisfied
SGX	software guard extension
SMPC	secure multiparty computation
SWIG	simplified wrapper and interface generator
SDK	software development kit
SWHE	somewhat homomorphic encryption
TEE	trusted execution environment

TDAG transaction directed acyclic graph
TLS transport layer security
VSCC validation system chaincode
VSC Visual Studio Code
ZKP zero-knowledge proof
ZK-SNARK zero-knowledge succinct non interactive argument of knowledge
ZK-STARK zero-knowledge scalable transparent argument of knowledge