UNIVERSITÀ DEGLI STUDI DELL'INSUBRIA

DOCTORAL THESIS

# Blockchain-based framework for data-centric and user-centric privacy protection

*Author:*
Federico DAIDONE

*Supervisor:*
Prof. Barbara CARMINATI
Prof. Elena FERRARI

*A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy*

*in the*

Strict SociaLab
DiSTA

January 21, 2022

# *Acknowledgements*

I would like to express my deep and sincere gratitude to my mentors, Professor Elena Ferrari and Professor Barbara Carminati. I learned a lot of things from them during my PhD thanks to their immense knowledge, vision and patience. It is a great honor for me to work and study under their guidance.

I also thank my thesis reviewers, .. and ..., for their instructive inputs and comments on my work.

Equally, I would like to express my thanks to all STRICT Social Lab and DiSTA members, students and supervisors, for the educational information and knowledge sharing we have had.

Finally, I want to thank my family as well as my friends.

UNIVERSITÀ DEGLI STUDI DELL'INSUBRIA

# *Abstract*

Computer Science and Computational Mathematics
DiSTA

Doctor of Philosophy

**Blockchain-based framework for data-centric and user-centric privacy protection**

by Federico DAIDONE

Today, most online services acquire more and more information about users, to intercept their habits and preferences for diverse purposes. The end-user is often unaware that he/she is giving away valuable sensitive information, which can harm his/her privacy. Therefore, he/she is forced to submit to the management of information by the service provider, unable in any way to express his/her wishes. Furthermore, considering that many of the services are based on a collaborative paradigm (e.g., social media services, IoT-based services, etc.), a mechanism for exchanging user information is engaged. This places serious security and privacy threats to the data each service provider exposes during the collaboration.

A promising approach to addressing these problems is to allow users to express their wishes about data processing and to enforce that preferences via blockchain, even when it comes to collaborative processes. This brings the benefits of trust decentralization, transparency, and accountability of privacy enforcement mechanisms in collaborative and non-collaborative contexts.

In this thesis, we propose a framework to deal with the challenges in executing privacy enforcement and secure off-chain resource sharing on the blockchain. In particular, we focus on inter-organizational business processes, IoT domain, and scientific processes. Finally, we show the results of our proposal and outline possible future research directions.

# Contents

# List of Figures

# Chapter 1

# Introduction

With technological advancement, people and businesses increasingly use online services, making them an essential part of our lives and society. With the benefits, our digital activities also bring some risks. The amount of data these services collect and exchange daily on a single individual is enormous. Often these data contain sensitive information, which could damage users' privacy. To limit these risks, several countries have enacted their own privacy regulations, such as General Data Protection Regulation (GDPR) [35], California Consumer Privacy Act (CCPA) [20], and Lei Geral de Proteção de Dados (LGPD) [58]. Each of them regulates privacy according to its own directives and implements different prevention and protection measures. In any case, to be compliant with privacy regulations (such as GDPR, CCPA), as well as *privacy by design* and *privacy by default* paradigms [51], each provider has to include a privacy-preserving component to protect user data. This protection can be achieved by exploiting different techniques [80, 78, 13], varying from access control mechanisms to anonymization algorithms. These techniques are applied based on protection scenarios. In this thesis, we consider two main scenarios, namely: *user-centric* and *data-centric* protection.

**User-centric protection.** Thanks to privacy regulation like GDPR [35], each service provider has to clearly express its *privacy policy* stating how the collected user data will be processed. The policy has to specify the purpose for which the data is collected, how long the data will be stored in the provider's server and if the data will be transferred to third parties. In this view, a user has to read and accept the provider's privacy policy before running a service or installing an app. However, this brings the user to accept all policy's terms and conditions to use the service/app or not to use it at all. In this scenario, adopting a user-centric privacy protection implies giving more control to users, as they should be able to state their preferences on how their personal data have to be managed at provider side. As an example, a user must be able to state for which purpose his/her data can be collected (such as geolocalization and not analysis of habits) and/or for how long these can remain in provider' servers. Therefore, it is essential to start developing services following the privacy by design/default model and, in particular, being able to enforce user privacy preferences on the collected data [76].

**Data-centric protection.** According to a data-centric view, security mechanisms have to address issues concerning the analysis, management, processing of data. For this purpose, several well-known techniques can be applied, such as authentication protocols, access control mechanisms, encryption schemes. However, new challenges arise in the case of *collaborative data processing*, which is an emerging paradigm adopted by several organizations. In this view, involved organizations agree on a common *business process* carried out through a set of interdependent tasks, which collectively realize a common business objective (e.g., insurance risk analysis, travel planning). Each task is assigned to one or more organizations; thus, we refer to

*inter-organizations business process*. Every organization has to execute the task locally, and this may involve requesting input data, consuming local resources, or producing an output. A notable example of an inter-organizations business process is the one implementing scientific workflows [10]. Here, organizations (e.g., research labs, data owners) collaborate on conducting scientific experiments (e.g., geological hazard estimation, bioinformatics, etc.). As this reference scenario shows, the workflow execution may include an extensive exchange of highly sensitive information. Thus, organizations may be reluctant to share their data during the collaboration without proper security mechanisms, ensuring a controlled data usage. To ensure a *data-centric privacy protection*, these data have to be shared according to the workflow execution. This implies that data have to be accessible only to the organization that has to carry out the task and only when the task has to be executed.

This thesis proposes a privacy-preserving solution that can cope with both these data protection scenarios. A common point in these two scenarios is the lack of trust among involved actors. Indeed, in user-centric scenario, end-users do not trust that his/her preferences will be considered during the data processing carried out by the service providers. Similarly, in the collaboration setting (i.e., data-centric scenario), organizations do not trust that their data will be used only according to workflow execution.

To overcome this issue, we exploited blockchain technology [82]. Indeed, thanks to its design and consensus algorithm, blockchain can create a trustworthy infrastructure, which is tamper-proof, replicated among the nodes of the network, and allows the partners involved in the collaboration/process to monitor and perform audits on its execution.

Thus, this thesis aims to exploit blockchain as a secure and reliable platform for executing operations in an untrusted distributed and collaborative scenario. In particular, the proposed blockchain-based framework aims to ensure controlled data usage, both with respect to possible authorizations related to the application scenario (e.g., collaborative process), and compliance with privacy regulation and users' privacy preferences.

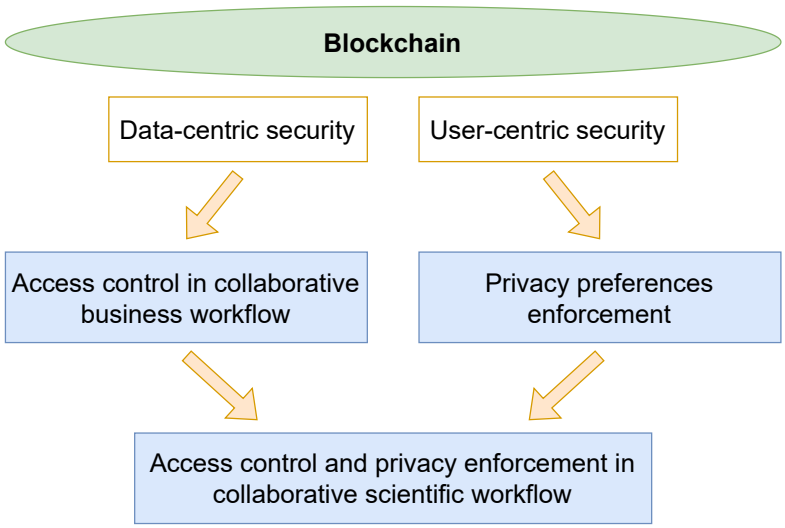



FIGURE 1.1: Thesis's research path

As depicted in Figure 1.1, in this thesis, we first proposed a solution for a data-centric scenario by designing an access control mechanism for a collaborative business process; and for a user-centric scenario by ensuring users' privacy preferences

enforcement.

A third step of the thesis implied merging the two scenarios to obtain a framework capable of simultaneously managing access control and privacy enforcement in a particularly stressful scenario such as that of collaborative scientific workflows. Scientific workflows, instead of others (e.g. business), are characterized by large amount of data that organizations and services deal with. The successful use of the blockchain in the management of workflows, however, clashes with the current limitations of the technology itself. The blockchain is not able to manage large amounts of data on-chain, nor to process huge amount of controls for the release of data. In particular, the high number of privacy preferences requires the use of different solutions than the single compliance check. The approach presented in this thesis aims to overcome the limitations of the blockchain, optimizing the management of controls and offering a blockchain-based system for capillary privacy verification. We achieve this through indexing techniques that allows us to operate with a much smaller data structure in terms of memory and management, limiting the number of compliance checks, and at the same time providing an environment that is reliable, immutable, and capable of carrying out checks efficiently guaranteeing the minimum overhead of resources.

## 1.1 Thesis' main contributions

In summary, this thesis provides the following research contributions:

- *A blockchain-based framework for ensuring a secure off-chain data exchange during the execution of inter-organizational business process.* Thanks to its design, blockchain is used as a reliable infrastructure that allows partners involved in the collaboration to monitor and perform audits on the workflow transitions [22, 60]. The business process can be modeled through a workflow, defined via the Business Process Model Notation (BPMN) [93]. When we consider a scenario consisting of web services, according to the service-oriented architecture (SOA), then the workflow can be defined via the Business Process Execution Language (BPEL) [83]. In general, existing blockchain-based workflow management solutions focus mainly on workflow coordination. However, a challenging characteristic of some workflows is that they require the exchange of a considerable amount of data that has to be managed off-chain, directly exchanged between data producer and consumer. This off-chain data sharing should follow the workflow execution. For this purpose, a first contribution, we proposed a mechanism for controlled off-chain information sharing via smart contracts in [71]. In particular, smart contracts are designed to coordinate the workflow execution and deploy a set of authorizations granting access only to the task executor and only to those resources needed for task execution and only during the task activation. We have also run a set of experiments to show the feasibility of our approach.

- *A blockchain-based framework to enforce user's privacy preferences.* As second contribution, in [30], we exploited the blockchain to perform privacy preferences' compliance check in a decentralized fashion. In this proposal, the compliance check is executed via smart contracts and validated thanks to a distributed consensus among the parties. In particular, we considered the IoT domain, as this technology pervades our lives every day and has given end users the opportunity of accessing personalized and advanced services based on the analysis of the sensed data. However, IoT services are also characterized by new challenges related to security and privacy because end users often share sensitive data with different consumers without precise knowledge of how they will be managed and used. To cope with this issue, we propose to complement data streams generated by IoT devices with proper metadata, storing information on device owner's privacy preferences. These metadate are then exploiting on the blockchain to enforce privacy preferences compliance via smart contracts. The system has been entirely developed and tested with different scenarios to show its performance and feasibility.

- *A blockchain-based framework to enforce user's privacy preference in inter-organizational scientific process.* Scientific processes differ from business ones in terms of the goal to be achieved. The scientific goal is often represented by the search for a solution to a scientific problem. Hence, this requires services that process huge data sets and perform rather complex operations, such as High Performance Computing (HPC) simulations. Data sets can be made up of user data, as in the case of human genomics research, and therefore their privacy must be protected. Using a user-centric approach, we must define a privacy preference for every single piece of data concerning the data owner. Considering the

blockchain as a trusted distributed system, we have developed methods to enable privacy enforcement on large volumes of data. The experiments confirm the goodness of the solution in comparison to the naive method.

## 1.2 Thesis organization

The dissertation is organized into 8 chapters, briefly described in the following:

- **Chapter 2:** we first provide background information on the blockchain technology such as the basic concept, cryptography techniques, components, and consensus mechanism.
- **Chapter 3:** for each of the previous challenges, we provide an overview of related works by analyzing what is missing and what has to be improved.
- **Chapter 4:** we present the proposed blockchain-based framework for ensuring a secure off-chain data exchange during the execution of inter-organizational business process.
- **Chapter 5:** we introduce the blockchain-based framework to enforce user's privacy preferences in IoT domain.
- **Chapter 6:** we illustrate the blockchain-based framework to enforce user's privacy preference in inter-organizational scientific process.
- **Chapter 7:** we conclude the paper highlighting the results obtained and the future directions.
- **Appendix A:** we provide information about two of the most used blockchain at the time of writing: Ethereum and Hyperledger Fabric.
- **Appendix B** we provide information about Chapter 5 on MUD protocol and theorem proofs.

## 1.3 Related publications

The research activities explained in this thesis have brought to the following publications:

- Christian Rondanini, Barbara Carminati, Federico Daidone, and Elena Ferrari. “Blockchain-based controlled information sharing in inter-organizational workflows,” in the proceedings of the 2020 International Conference on Services Computing (SCC), in Beijing, China. (2020, October). IEEE. [71]
- Federico Daidone, Barbara Carminati, and Elena Ferrari. "Blockchain-based Privacy Enforcement in the IoT domain," in the IEEE Transactions on Dependable and Secure Computing (2021). [30]
- Federico Daidone, Barbara Carminati, and Elena Ferrari. "Blockchain-based Privacy Enforcement in Scientific workflow."

# Chapter 2

# Background

The work conducted on this thesis aims at enhancing data and user privacy in decentralized and distributed applications and services. In this chapter we introduce the concept of blockchain technology, and privacy enforcement in order to help the reader to better understand the thesis.

## 2.1 Blockchain

The blockchain has its roots in the concept of distributed ledger. A distributed ledger is a tamperproof sequence of data shared and synchronized among serveral nodes, maintained by a peer-to-peer network [85]. The nodes can belong to different geographically distributed organizations, and each one contains a copy of the data, accessible by several people. Anyone who has access to the data can make changes or additions, reflecting the changes in all other copies. The novelty introduced by blockchain is its ability to achieve coordination and verification of each operation carried out on the platform by different parties. This eliminates the need for a centralized and trusted authority, achieving full decentralization in running applications with coordinated and autonomous operations. In the blockchain, information is exchanged through transactions between the nodes of the network. Transactions have input and outpoint data, and a unique id that represents them within the blockchain. They are sorted and collected in blocks which are then broadcast to the other nodes, using public key cryptography. its roots in the concept of distributed ledger. A distributed ledger is a tamperproof sequence of data shared and synchronized among serveral nodes, maintained by a peer-to-peer network [101]. The nodes can belong to different geographically distributed organizations, and each one contains a copy of the data, accessible by several people. Anyone who has access to the data can make changes or additions, reflecting the changes in all other copies. The novelty introduced by blockchain is its ability to achieve coordination and verification of each operation carried out on the platform by different parties. This eliminates the need for a centralized and trusted authority, achieving full decentralization in running applications with coordinated and autonomous operations. In the blockchain, information is exchanged through transactions between the nodes of the network. Transactions have input and outpoint data, and a unique id that represents them within the blockchain. They are sorted and collected in blocks which are then broadcast to the other nodes, exploting public-key cryptography. Thus, within a block we can find a set of transactions with their summary reference, a timestamp and a reference to the previous block. The summary reference of the block is obtained by means of an authenticated data structure (e.g., a Merkle tree), which ensure the integrity of the block. The previous block reference identifies the position of the block with respect to the previous one, using a hash function and constituting

the so-called immutable block chain. The first block is called the genesis block and is the reference point to measure the distance to another block, called the block height. An example of a blockchain with 4 blocks is shown in Figure 2.1.

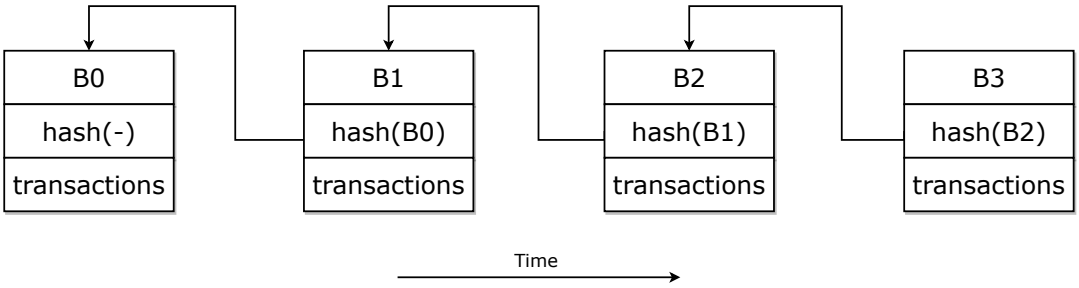



FIGURE 2.1: Example of blockchain

### 2.1.1 Smart contracts

The smart contract concept was introduced by Szabo [66] as: "a computerized transaction protocol that executes the terms of a contract". Today, the concept differs slightly from the initial definition, thanks to the introduction of Ethereum's smart contracts, which are Turing-complete and allow us to define any behavior, even not tied to the terms of a contract. Now, smart contracts are small programs stored on blockchain and therefore immutable. They have a unique address on the blockchain and can be invoked through transactions, even passing parameters. Their execution is deterministic because the execution of the smart contract, at the same state, must always return the same output for validation reasons. Each blockchain platform uses its own language for smart contracts and its own consensus system to verify the correctness of the execution.

### 2.1.2 Consensus

In general, any distributed system allows for the interaction between different processes that must be performed properly. Processes must agree on how to collaborate, reaching consensus on a common outcome after their operations [19]. At this stage we can encounter several problems, such as malicious manipulations, propagation delays, fail or crash that can hinder agreements. Over the years many solutions have been proposed for these problems, leading to the definition of different consensus models and protocols [19, 25, 57, 62].

Referring to the blockchain, each node can perform operations on the ledger. Operations on the blockchain take place through a distributed coordination system in which a consensus must be reached. Consensus protocols ensure that information on the blockchain are updated to the latest approved ledger status and that each node has the same copy. In the following subsections we analyze the most representative consensus protocols used in the blockchain.

#### Proof of Work (PoW)

The Proof of Work (PoW) consensus mechanism is based on the consumption of hardware and energy resources to prove that the task has been completed regularly. In Bitcoin [65], the miners, ie the block generators, solve a cryptographic puzzle to generate a valid block that can be spread over the network. The generation of the

block, called mining, is a difficult operation, usually it consists in adding a nonce to the block such that the generated hash is greater than a certain threshold. On the other hand, checking the correctness of the block is a simple operation because it involves making a hash. This imbalance between the two operations allows us to increase the block generation times so as to reduce conflicts such as the sybil attack and double spending. A valid block must then be approved by the majority (i.e. 50% plus 1 of all peers), to be included in the blockchain. If the majority is not honest, more malicous blocks will be accepted in the blockchain than honest ones, which violates the correctness of the consensus.

#### Proof of Stake (PoS)

The PoW algorithm requires a lot of economic and energy resources to be implemented. In 2012, a user of the Bitcointalk forum proposed an algorithm, the Proof of Stake (PoS), to limit the carbon footprint of the PoW, estimated at an energy consumption of 29.05 TWh, approximately 0,13% of the world's annual energy consumption. The PoS is based on the principle of "stake", which is the amount of cryptocurrency or token owned by a user. The larger the stake owned, the less likely the owner is to engage in malicious behavior for the system. Participants with the largest stake are selected pseudorandomly, to validate the block and add it to the blockchain. During the minting of the block, no cryptocurrecy is generated, therefore the only incentive for the miners are the transaction fees. Despite this, PoS is an expensive system, because the owner of the cryptocurrency has to invest a large initial capital to use in the algorithm.

#### Byzantine Fault Tolerant (BFT)

The Byzantine Fault Tolerant (BFT) protocols try to solve the problem of the correct execution of Byzantine processes. They are malicious processes that can arbitrarily fail at any time with respect to the execution of the algorithm or task imposed by a controlling process [19]. Processes may not communicate with each other or exchange arbitrary unsigned messages. In a distributed system, designed to be BFT, it must be able to operate correctly and reach consensus even in situations where processes behave in an arbitrary way. For example, considering a communication system, Byzantine processes, during operation, may arbitrarily no longer follow protocol, not responding to messages, or selectively delete them, or propagate false responses, and so on. For an asynchronous system, where messages can have infinite delays, to reach consensus, we must assume that at most fewer than one-thirds of all processes are Byzantine [19]. If this is not done, consensus is not reached. Today, these protocols are applied in blockchains such as Hyperledger Fabric [7] and Tendermint [17]. Specifically, they intervene in the verification of a block, where the participants must agree on its validity. The peers on the network update their local copy of the blockchain with the proposed new block only if at least two-thirds plus one of the peers has successfully approved this block.

### 2.1.3 Blockchain platforms

There are many blockchain platforms, each with their own characteristics, capabilities and fields of application. We briefly give a classification and then discuss those relevant to this thesis.

A first classification is made on the blockchain read/write permissions granted to participants:

- Permissionless, each node of the network can validate transactions and participate in the consensus algorithm;
- Permissioned, only nodes to which particular permissions have been granted can participate in the consensus and validate transactions.

A further subdivision is represented on the membership of the data infrastructure:

- Public, each node can be part of the network without a specific identity;
- Private, only selected nodes can participate in the network to which an identity is associated.

We can also find combinations of blockchains, such as Bitcoin and Etehereum which are public and permissionless, or Hyperledger Fabric which is private and permissioned.

## 2.2 Privacy enforcement

Privacy and data protection are considered increasingly important due to the emergence of new social and economical online activities. These interests have pushed the definition of new privacy laws and a new privacy models.

### Privacy regulations

Only 66% of the countries in the world have defined a full regulation to protect personal data, the others are working on it (10%) or do not have any at all (24%)[1]. Now we introduce the most relevant privacy regulations.

#### General Data Protection Regulation (GDPR).

The GDPR is the most detailed and comprehensive law currently in force for regulating how the data of EU citizens must be protected by companies that use them. The law was approved in 2016 and came into force in 2018, replacing the Data Protection Directive 95/46/EC. The regulation is also applied to any non-EU company that offers services or goods to citizens residing in the EU. For this reason, the GDPR has a global effect and influences the regulatory models of other nations. Of the 11 chapters and 99 articles, we mention only the founding principles. Article 4 defines personal data "any information relating to an identified or identifiable natural person.", And the term "identifiable" refers to the possibility of identifying a natural person with "all means reasonably likely to be used" (Recital 26) . From this, a company that does not directly identify natural persons, but holds data that could identify a person, they are considered personal data and therefore subject to the GDPR regulation. Article 5 establishes the following principles on personal data:

- "lawfulness, fairness and transparency principle", the data must be processed according to the regulations and in a transparent way;

[1]Statistics available at UNCTAD: https://unctad.org/page/data-protection-and-privacy-legislation-worldwide

- "purpose limitation principle", the data must be collected only for specific and limited purposes, their processing cannot go beyond the declared purposes;
- "data minimization principle", the data collection is limited only to those necessary and relevant for the declared purposes;
- "accuracy principle", the data must be accurate and kept up to date;
- "storage limitation principle", data retention is permitted only for the minimum necessary time for their processing in accordance with the declared purposes;
- "integrity and confidentiality principle", the data must be processed securely, ensuring that personal data are not manipulated or disclosed.

Furthermore, we mention the obligation for companies to notify their users of any data breaches incurred, and the right to be forgetten for users who wish to be deleted from the companies' databases.

**California Consumer Privacy Act (CCPA).**

The California Consumer Privacy Act (CCPA) is a privacy law enacted by California in 2018 and entered into force in 2020. It complements the jagged legal system of the United States, giving California residents new rights on personal data and imposing new rules on data processing. The GDPR spurred the generation of the CCPA, with which it has many overlaps. In particular, the CCPA defines the concept of personal information, regulates the procedures and practices to be followed for the protection of information and confirms the obligation to notify in the event of a data breach. Similar to the GDPR, the CCPA also extends to foreign companies operating in the California territory to protect the privacy of their residents.

**Lei Geral de Proteção de Dados (LGPD).**

Brazil's Lei Geral de Proteçao de Dados (LGPD) is also based on the GPDR, with which it shares the same applicability and scope. The main difference is related to the lower penalties in case of violation of the regulations, to adapt to the economic conditions of Latin America. They entered into force in 2020.

### 2.2.1 Privacy model

In general, a privacy policy is specified by a consumer to mainly state which personal data it collects from individuals, for which purpose, for how long, and whether the collected data will be released to third-parties. On the other hand, data owners can specify their privacy preferences as constraints on each single privacy policy component (e.g., purpose, retention time, third-party release). The literature presents several models to represent user privacy preferences (see e.g., [1]). Although the proposed framework can work with several privacy models, we choose to adopt the model presented in [73], since this has been designed for the IoT scenario. This is an expressive privacy model that, in addition to standard privacy-related elements (e.g., purpose), also supports a set of features tailored to the IoT domain to allow data owners to limit how and which data can be derived during IoT analytic processes. Moreover, it allows the automatic generation of privacy preferences for newly derived data (e.g., information resulting from data fusion). In this

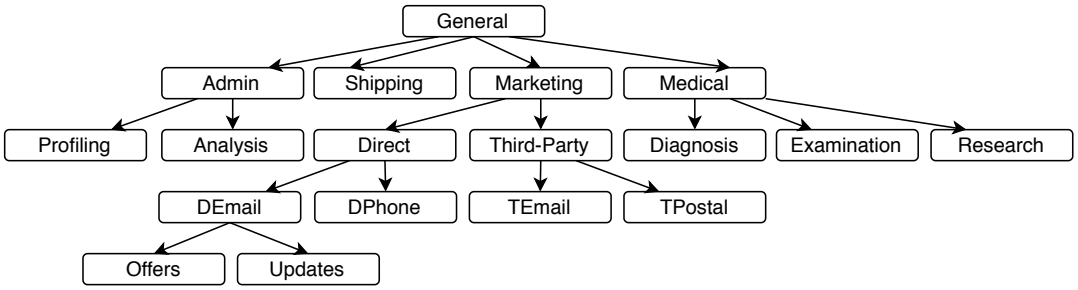



FIGURE 2.2: An example of purpose tree

proposal, as a first step, we have considered a lighter version of [73], such to focus mainly on the traditional privacy preference components (e.g., purpose, retention time). We postpone as future work to extend the framework to have full compliance with the model in [73].

According to [73], purposes are hierarchically organized into a tree structure $PT$, as shown by the example in Figure 2.2. Such tree structure is used to limit the number of purposes that need to be specified into a privacy preference. Indeed, a preference with the purpose $purp$ also implies the authorization for all the purposes rooted at $purp$ in $PT$. However, the model also allows the specification of exceptions to this propagation, as formalized by the following definition.

**Definition 1** (Intended purpose [73])**.** An intended purpose $ip$ is a pair $(Aip, Exc)$, where $Aip$ (allowed intended purposes) is a set of purposes belonging to a purpose tree $PT$ and $Exc$ (exceptions) is a set of purposes that descend from elements in $Aip$. $ip$ authorizes the access for all purposes that descend from[2] the elements in $Aip$, except for those that descend from any element in $Exc$.

Now, it is useful to define the set of purposes implied by $ip$, denoted as $\overrightarrow{ip}$, to check the privacy compliance. This is the set of $purp$ in $Aip$ including all their child nodes in $PT$. From this set, we remove all $purp$ in $Exc$, plus their child and parent nodes in $PT$.

Formally, let $\overrightarrow{Aip} = \bigcup_{purp \in ip.Aip} purp^{\downarrow}$[3], where $purp^{\downarrow}$ is the set composed of $purp$ and all purposes descending from $purp$ in $PT$, and $\overrightarrow{Exc} = \bigcup_{purp \in ip.Exc} purp^{\updownarrow}$, where $purp^{\updownarrow}$, the set composed of $purp$ and all purposes descending and ascending from $purp$ in $PT$, $\overrightarrow{ip} = \overrightarrow{Aip} - \overrightarrow{Exc}$.

A privacy preference is in turn defined as follows.

**Definition 2** (Privacy preference [73])**.** A privacy preference is a tuple $pp = \langle \alpha, consumer, ip, rt, tpu \rangle$, where $\alpha$ is the considered data tuple to which the preference refers to, $consumer$ specifies the set of consumer's identities to which $pp$ applies, $ip$ specifies the intended purposes for which $\alpha$ can be collected and used by any entity in $consumer$, $rt$ specifies the retention time, and $tpu$ the third party usage.[4]

**Example 1.** Let us consider the scenario of a smart home, equipped with medical IoT devices for the remote monitoring. The basic symptoms monitoring system

[2]Hereafter, we assume this relationship as reflexive, meaning that every element descends from itself.

[3]In what follows, we use the dot notation *dataStructure.element* to indicate an element inside a tuple or a data structure.

[4]For simplicity, in what follows, we assume that the retention time is expressed in days and the third party usage $tpu$ assumes one of two values, namely *shareable* or *unshareable*.

is composed by sensors for acquiring the respiration rate, heart rate, temperature, and oxygen saturation [52]. The sensed data must reach the hospital, namely the consumer, who takes care of the treatments. A patient can define a privacy preference $pp$ for each data stream generated by the monitoring system. Consider, for instance, the heart rate monitoring. Suppose that a user wants to share the sensed data only with his/her hospital and only for administration and medical purposes, except for research. Moreover, he/she allows a retention time of 90 days and prohibits the dissemination of such information to third parties. To model these requirements, the user can specify $pp$ as follows: $pp = \langle heart\text{-}rate, \{hospital\text{-}company\}, \langle \{admin, medical\}, \{research\} \rangle, 90d, unshareable \rangle$. By considering the purpose tree in Figure 2.2, the set of purposes authorized by $pp$ is $\overrightarrow{ip} = \{admin, profiling, analysis, diagnosis, examination\}$.

Moving on to the privacy policy, it is modeled as a tuple $\langle \alpha, up, dataRet, dataRel \rangle$, where $\alpha$ is the attribute object of the policy, $up$ is the data usage purpose, $dataRet$ is the data retention time, and $dataRel$ is the third party usage. During the privacy compliance check, the privacy policy must respect the wishes of the data owner described in the privacy preference. In particular, if the $p.up$ are contained in $pp.\overrightarrow{ip}$, if $p.dataRet$ is less than or equal to $pp.rt$ and if the same data release flag is present, then the privacy enforcement is successful. We use the notation $PE(pp, p)$ to indicate a privacy enforcement between a privacy preference $pp$ and a privacy policy $p$. The result of privacy enforcement can be $PE(pp, p) = true$ if $p$ respects the wishes of $pp$, otherwise $false$.

# Chapter 3

# Review of Literature

The blockchain is one of the disruptive technologies of recent years that has introduced new paradigms for the solution of various problems. To better understand the evolution of solutions, for each application field discussed in the thesis, we have carried out literature reviews for blockchain and non-blockchain solutions.

This chapter begins with the review of collaborative processes and the secure exchange of data in Section 3.1, and then examines them in a blockchain key. Subsequently, the solutions for privacy enforcement in IoT processes are studied in Section 3.2, also taking into account those based on blockchain. Moving on scientific workflows in Section 3.3, we have reviewed the non-blockchain and blockchain solutions.

## 3.1 Collaborative processes and secure data exchange

Literature offers several proposals dealing with data confidentiality and privacy issues in web services, as well as in service composition (see [9] for a survey). Several proposals exist in the literature addressing the issue of achieving secure resource sharing in inter-organizational workflows, by leveraging on access control mechanisms. [84] reviews and compares different access control models in the context of collaborative systems, pointing out that RBAC should be extended in order to support Computer-Supported Cooperative Work as roles can be easily applied for resource management but are difficult to be applied to human collaborations. Other work, such us [55], propose approaches for access control in inter-organizational workflows, where data access is mediated by a server. While some of the concerns considered in these work are the same considered in this thesis, the framework on which we deployed the collaboration (aka blockchain) imposes to address new requirements, such as how to integrate the blockchain in a collaborative process execution.

### 3.1.1 Blockchain-based collaborative processes and secure data exchange

The idea of exploiting blockchain for inter-organizational business process execution has been investigated in several work. To the best of our knowledge, the first approach has been proposed in [89], where it has been discussed the idea of having the smart contract as a mediator to control the collaborative process. A further step has been done in [37], where an optimized method for executing business processes defined in the standard Business Process Model and Notation (BPMN) has been proposed. Furthermore, [60] proposes an open-source Business Process Management Systems (BPMS) with the goal of combining the blockchain benefits (e.g, tamper-proofness) with the development convenience of a BPMS. [69] proposed a fully-functional prototype for a blockchain-based runtime verification of process instances using the Bitcoin blockchain, thus to show the suitability of blockchains as

trust basis for decentralized and indisputable runtime verification for choreography-oriented BPMs. The work is extended in [70].

While the above-mentioned proposals show the increasing interest in having business processes deployed on the blockchain, none of them has taken into account data confidentiality issues. The only proposal we are aware of dealing with data confidentiality, is the one in [21], where a framework has been proposed, which takes into account the confidentiality issues of business processes deployed on the blockchain, by leveraging on encryption techniques. However, [21] does not address resource access control and secure data exchange, which is one of the goals of this thesis.

The literature offers several proposals exploiting blockchain to deal with access control (see [72] for a survey). For instance, [63] proposes a system, extending Bitcoin, where users can transparently observe access control policies on resources. The thesis leverages on attribute-based access control and eXtensible Access Control Markup Language (XACML) to define policies and store arbitrary data on Bitcoin. In their next study, the authors [32, 31] consider smart contracts to enforce access control policies. They have implemented a proof of concept prototype using XACML policies and the Ethereum platform. [68] describes a cryptocurrency blockchain-based access control framework for IoT devices, called *FairAccess*, in which a different smart contract is created to regulate accesses, for every resource-requester pair. [88] introduces an access control framework, leveraging on InterPlanetary File System (IPFS) as decentralized storage system, Ethereum blockchain, and Attribute-Based Encryption (ABE). In [97] a transaction-based access control (TBAC) framework has been proposed. It consists of a platform that integrates ABAC and blockchain, combining four types of transactions and Bitcoin-type cryptographic scripts, corresponding to subject registration, object holding and publication, access request, and authorization grant. [54] suggests using blockchain as a decentralized infrastructure for access control management. A proof of concept prototype has been implemented using a Multichain platform and ciphertext-policy attribute-based encryption (CP-ABE). However, the above mentioned work does not consider the scenario of resource protection for inter-organizational collaboration, which is the focus of our thesis, as such they do not provide support for the temporal activation of authorizations and dynamic resource allocation during the workflow execution. Other related work are those leveraging on blockchain for scientific workflow management. However, most of the proposals (e.g., [26, 96]) mainly focus on data provenance issues.

## 3.2 Privacy enforcement in IoT processes

The IoT field has experienced considerable development and, as a result, security and privacy have been studied deeply. In what follows, we briefly review those proposals closer to our proposed framework, considering the IoT scenario as a collaborative process among data owners and consumers. Many papers have addressed the issue of privacy preferences enforcement in the IoT domain. Over the years we have seen a gradual migration of the policy enforcement process from the cloud to the fog. The authors in [3] present the Policy Enforcement Fog Module (PEFM), achieving better privacy enforcement results with less overhead than a cloud-based solution. The work presented in [73], from which we borrow the privacy model adopted in our framework, introduces a decentralized privacy enforcement framework for the IoT scenario. Users can specify privacy preferences on local gateways which control

the release of the data to the consumers. [74] presents the Privacy Preference for IoT (PPIoT) Ontology and provides methods and examples to apply it in the IoT context in light of the GDPR. The use of machine learning is also increasingly present in the privacy enforcement process, for instance [5] demonstrates this trend. The authors proposed how machine learning can use data sources for more robust privacy management. An overall view of the different technologies and methodologies for IoT privacy enforcement is given by a recent comparative study [56]. It provides an analysis of the state-of-the-art of the main IoT frameworks and focuses on GDPR support. The results show that many frameworks are still not GDPR-compliant, and there are still open challenges, such as usable user interfaces, lack of privacy risk analysis and so on. Our framework has a number of innovative features compared to the abovementioned proposals, the main is that it is a fully decentralized solution for privacy preferences' enforcement, leveraging on blockchain, that also guarantees transparency of the enforcement process for all the involved parties without the need of a trusted entity.

### 3.2.1 Blockchain-based privacy enforcement in IoT processes

IoT processes have benefited from blockchain technology for solving different security and privacy issues. [33] proposes a "miner", that is, a high resource device, installed in every smart home for communication between IoT devices and blockchain. Here, the blockchain is used to store policies and for access to resources, adding and removing the devices authorized for communication. In the medical field, healthcare organizations need data storage systems that protect patient privacy but, at the same time, enable data sharing for medical research. Therefore, blockchain has been investigated for its characteristics of transparency, trust, and immutability of data, as in [64] and [27], where the authors propose that sensitive patient information be encrypted and stored in the blockchain. Unlike our system, data are not shared as private data, but stored permanently in the blockchain. The general-purpose architecture proposed in [75] uses Hyperledger Fabric to protect communication between IoT devices, using asymmetric encryption. The blockchain includes malicious actor detection and guarantees non-repudiation and privacy. The work presented in [4] implements a reliable ownership management system for medical IoT devices that uses Ethereum and smart contracts instead of Trusted Third Parties. In the review study presented in [81], the authors analyzed the vulnerabilities of medical IoT devices and then proposed a solution for the exchange of patients personally identifiable information, leveraging on blockchain. [34] presents a system for the exchange of patient data via blockchain, where data are encrypted, signed with the digital ring model, and then stored on the cloud. [40] focuses on wearable medical devices only, where a smart device, such as a smartphone, collects raw data and communicates with the blockchain. At the end, it sends an alert to the hospital to report the presence of data on the blockchain. Another work closely related to ours is [53], a platform for monitoring patient vital signs using smart contracts on Hyperledger Fabric. PATRIoT [61] is an Ethereum IoT data sharing platform. They use an ontology-based privacy model called LIoPY. Unlike our platform, their privacy enforcement process is done off-chain, using the blockchain only to organize the data exchange between data producer and consumer.

Our proposed framework greatly differs from the abovementioned proposals in that none of them: (1) is highly user-centric with the possibility of specifying fine-grained data owner privacy preferences (2) is designed for the massive audit of the privacy preference enforcement process.

## 3.3 Privacy enforcement in scientific workflow

Scientific workflows have been under study for many years now. Unlike business workflows, they pose greater privacy challenges. The many data involved are often linked to individuals and their sharing between different services complicates the management of privacy.

A first approach to ensuring user and data privacy was to define an ontology-based privacy model that is incorporated into scientific workflows [38]. The privacy policies are defined during the design and development of the workflow, making the model fine-grained. It also allows you to express some corrective actions if privacy is violated, while the privacy enforcement is carried out centrally by a reasoner.

Another way is that taken in [12], where the authors propose a method to manage sensitive data without altering the workflow. The workflow engine, running in a trusted environment, is free to examine the data. The processed data, before being published, go through an anonymizer to ensure the privacy of the data owners. An administrator sets the sensitive parameters to be intercepted and a desired degree of anonymity.

Another work brings us back to the ontology-based privacy model, but this time with a greater degree of integration than previous works. The authors, in [14], integrate the privacy model with the Business Process Model and Notation (BPMN) language for clinical workflow. A reasoner then deals with privacy enforcement, capable of detecting privacy violations.

The studies examined entrust the compliance check to a centralized entity and focus heavily on the definition of privacy policies during the development of the workflow. This results in less flexibility of the workflow that often has to be rewritten and in a restricted adaptability to the wishes of the data owners, that can also change at run time.

### 3.3.1 Blockchain-based privacy enforcement in scientific workflow

Moving on to distributed ledger technologies, such as blockchain, we notice that most of the work focuses on provenance data.

The most relevant work that combines provenance data and privacy is ProvChain [59], even if not expressly addressed to scientific workflows. They protect the privacy of users by applying anonymization techniques so that the rest of the blockchain nodes do not know their true identity. Here, the blockchain does not play an active role in anonymization but is used as an element to attest the provenance data, as a means of ensuring the reproducibility, integrity and trustworthiness of data in collaborative environments.

An evolution is represented by BlockFlow [29], a blockchain-based platform for scientific provenance data, capable of capturing prospective, retrospective and evolution of data, storing them in an immutable way on blockchain and allowing their audit. Through query it is possible to carry out targeted searches and audits, for example it can return all the programs executed during an experiment between two dates.

Another study focuses on genomic data sharing. This is an application that requires great attention because traditional privacy models offer limited protection. The authors, in [15], expose the problems of genomic data sharing, evaluating how an attacker can learn sensitive information about individuals by associating publicly available information with genomic data. The paper proposes a taxonomy of privacy attacks, and an overview of the techniques used to preserve privacy, such as

access control, homomoprhic encryption (HE) and secure multiparty computation (SMC). From this it is clear that the major problems are related to the technology to control data flow and data sharing, essential for scientific progress.

From the presented studies, we note that there is not yet a work that brings run time privacy enforcement to scientific workflows in distributed scenario. Our platform aims to be the glue between centralized privacy enforcement models for scientific workflows and blockchain applications, such as data provenance, guaranteeing fine-grained user privacy.

# Chapter 4

# Blockchain-based controlled information sharing in inter-organizational workflows

Nowadays, organizations need to set higher and higher business goals in order to cope with market requirements. Indeed, a widespread strategy for organizations is to join in inter-organizational processes, which set collaborations and resource sharing among involved organizations. However, the possible lack of trust among the organizations poses relevant issues on the processing of sensitive resources. A promising approach to cope with this issue is leveraging on blockchain technology. Thanks to its design and consensus algorithm, blockchain provides a trustworthy infrastructure that allows partners involved in the collaboration to monitor and perform audits on the workflow transitions. In general, the focus of the existing blockchain-based workflow management solutions is mainly workflow coordination. However, a challenging characteristic of some workflows is that they require the exchange of a big amount of data that has to be managed off-chain, that is, directly exchanged between data producer and consumer. This off-chain data sharing should be secured and controlled such to follow the workflow execution.

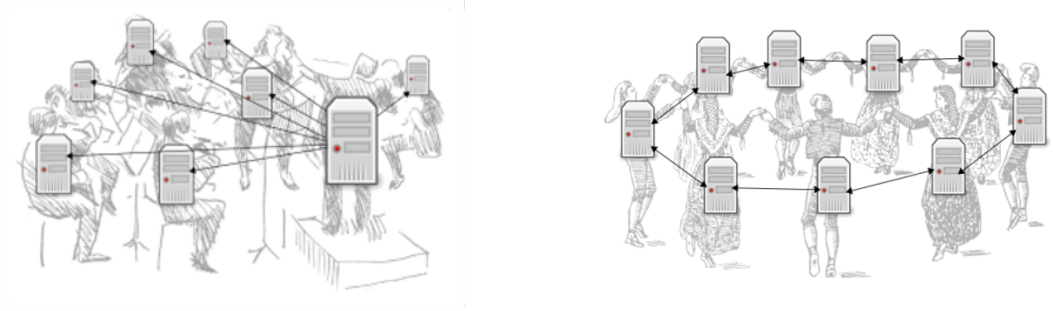
To cope with this challenge, in this Chapter, we propose a controlled information sharing in inter-organizational workflows enforced via smart contracts. These are designed to coordinate the workflow execution, as well as to deploy a set of authorizations granting access only to the task executor and only to those resources needed for task execution and only during the task activation. We have also run a set of experiments to show the feasibility of our approach.

The Chapter is organized as follows. Section 4.1 introduces the considered scenario and its main requirements with reference to resource sharing in Section 4.2. Section 4.3 presents the proposed approach and the key ideas behind it. Section 4.4 shows experimental results.

## 4.1 Inter-organization process

Collaboration among different organizations becomes essential to achieve complex goals. Before actively collaborating, organizations define a common process to follow. This is modeled as a set of activities that executed, following a predetermined order, leads to the achievement of the objective. The goal can be business, such as a marketplace (sellers, payment systems and logistics), or scientific, such as genetic research. In both cases, the activities can be modeled as a flowchart, where there

are inputs, outputs, and decision points to compose a workflow. For example, Business Process Modeling Notation (BPMN) is a modeling technique and notation for business processes.



(A) Orchestration

(B) Choreography

FIGURE 4.1: Coordination methods

The workflow can be coordinated following two methods. The orchestration paradigm (see, Figure 4.1a) implies following the directives of a central controller / broker that regulates the execution of tasks by partner involved in the collaboration. In contrast, choreography (see, Figure 4.1b) does not require a centralized entity, organizations invoke and execute workflow tasks themselves.

When several organizations, or even several departments of the same organization, collaborate in a process, a possible lack of trust can arise. This implies several challenges that must be addressed correctly to obtain a safe environment.

- **Data confidentially.** Sensitive information, in collaborative scenarios, often needs to be shared with other organizations. This is a major challenge, because lack of trust leads organizations to limit or refuse collaborations to avoid uncontrolled information exchange. Information leakage and dishonest behavior can severely damage an organization.
- **Secure off-chain data exchange.** During collaboration, organizations need to exchange information, such as data sets, operating specifications, control information, etc. This highlights the need for intensive communication and data sharing between collaborative participants. To avoid data disclosure that can cause significant economic and image damage for parties involved, the traditional solution is to rely on the access control mechanism in place at the data owner side. This results in a local access control implemented by data owner, which requires a significant overhead for him/her. This approach is further complicated when local authorizations need to be updated frequently to follow the execution of the dynamically and temporally evolving workflow.

## 4.2 Resources sharing in collaborative processes

Before presenting the proposed blockchain-based solution, in this section, we illustrate in more details the general flow of a collaborative process. In general, a collaborative process among a set $\mathcal{O}$ of organizations can be modelled as a workflow that requires multiple tasks to be performed. Each task $T$ has to be performed by one of the organizations involved in the collaboration, to which we refer to as *executor*, denoted as $T_{exec}$. We assume that the assignment of a task to its executor is done before the workflow execution starts and cannot change during the workflow deployment.

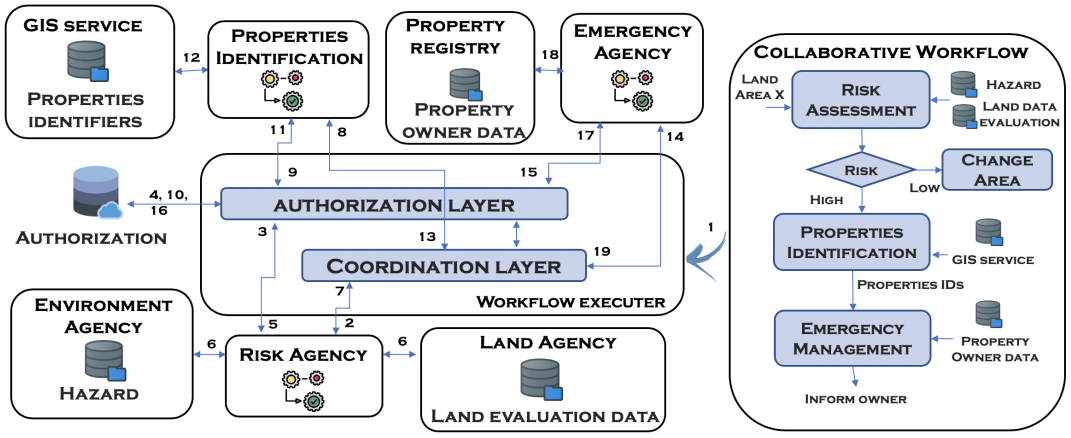



FIGURE 4.2: An example of collaborative process.

The execution of $T$ may require to perform different *actions* (e.g., elaboration, query, mining, visualization) on a set of resources, denoted as $T_{res}$. In addition to its own resources, $T_{exec}$ might need to access resources owned by other organizations in $\mathcal{O}$. For each resource $r \in T_{res}$, we indicate with $Owner_r \in \mathcal{O}$ the organization producing $r$.

**Example 2.** Let us consider a process aiming at identifying those properties that, in a given area, have a high risk to be subject to natural disasters (e.g., floods). The purpose is to inform the emergency crisis agency about the identified properties, which has to be then authorized to retrieve contact information about their owners from local registers. This process implies a collaboration among organizations performing the involved tasks (i.e., risk assessment, property identification, emergency management) and organizations that provide the needed data (e.g., land evaluation data, building information, environment agency for flood risk, property registry). A workflow modelling this process is represented in Figure 4.2, where the first task, called *risk assessment*, has to be carried out by proper experts (e.g., a risk agency). To perform this task, given an input area $X$, the risk agency has to be authorized to access information about hazards and evaluation of land $X$ from two distinct organizations participating in the collaboration (e.g., Environment agency, Land agency). If the risk assessment task returns a risk level for $X$ higher than a given threshold, the workflow executes the second and third tasks. More precisely, the *properties identification* task aims at finding the identifiers of properties built in area $X$, by inquiring a map services (e.g., GIS service). The returned ids have to be passed then to the third task, called *emergency management*, performed by the emergency crisis agency, which has then to retrieve from local registries (e.g., property registry) the contact information of the corresponding owners (step 1 in Figure 4.2).

In order to support workflow execution, there is the need of two main layers: the *coordination layer*, which regulates the execution of workflow tasks, and the *authorization layer*, which deploys the underlying controlled data sharing. The coordination layer has to invoke the execution of a task by passing to its executor the needed information, e.g., which actions/services the executor has to perform and the possible input parameters (see steps 2,7,8,13,14,19). in Figure 4.2). For instance, with reference to Example 2, the coordination layer has to interact with the risk agency to invoke its risk assessment service by passing $X$ as input area (2). Once the task has been invoked, the coordination layer waits for the results (7) and, based on these, it continues the workflow execution (8). This might imply performing other task

invocations, loops, or branches, till reaching the final state. The coordination layer has to work closely with the authorization layer. Indeed, in order to make a task executor able to perform its actions, the *authorization layer* has to set up the proper authorizations to make $T_{exec}$ able to get access to each resource $r \in T_{res}$ directly from the $Owner_r$ realm (see Figure 4.2 steps 3-5,9-11,15-17).

We identify the following requirements drawing the design principles of the coordination and authorization layers.

**Coordination layer**. Due to the possible lack of trust among the involved organizations, it is relevant to ensure the correct execution of the workflow. This can be achieved by delegating the coordination layer to a third-party (broker) that will have full control over the workflow execution (e.g., tasks, resources, etc). However, this might bring several security issues. For instance, a dishonest broker could take advantage of its position in order to favor an organization. Moreover, this solution exposes the whole process to a unique point of failure (aka the broker). A promising way to deal with this lack of trust is leveraging on blockchain to support secure collaborative processes [23]. The blockchain technology suits well, since, by relying on a distributed consensus, it guarantees the correct execution of a process (aka smart contract). In this view, the workflow could be encoded by a smart contract, executed and validated by the blockchain (see e.g., [21] as an example).

**Authorization layer**. For each task $T$, the purpose of this layer is to deploy a set of authorizations $T_{auth}$ to enable $T_{exec}$ the access to resources in $T_{res}$. This set of authorizations and the corresponding enforcement have to satisfy the requirements discussed below.

*Temporal (activation of) authorizations*. Access to $T_{res}$ has to be allowed only during the task execution. As an example, the authorizations enabling the emergency crisis agency to access contact information about properties' owners have to be valid only if, during the workflow execution, the condition on risk level (i.e., area risk level greater than the threshold) is satisfied. This implies that authorizations cannot be deployed before the workflow execution; rather, these have to be activated only once the task is indeed invoked and have to be deactivated soon after the task termination. This temporal activation of $T_{auth}$ requires that authorization and coordination layers work closely together.

*Dynamic resource allocation and least privilege*. It is relevant to note that $T_{auth}$ cannot be prearranged before the workflow execution as resources in $T_{res}$ might be dynamically specified. Recalling Example 2, we do not know before the execution of the *properties identification* task, which are the properties considered at risk and thus for which the emergency crisis agency is authorized to access owners' information. We need therefore to provide a mechanism where resources to be allocated for a task are dynamically determined, and, as a consequence also their authorizations. Moreover, we have to consider the least privilege principle [77], that is, we need a mechanism to check whether $T_{exec}$ had required more resources of what is indeed needed for the task execution. As an example, we have to detect if the emergency crisis agency has required information of property owners that are not considered at risk.

*Access control enforcement*. In our reference scenario, resources are managed and controlled off-chain, directly by the corresponding owners. As such, a naïve place where deploying and enforcing the authorizations needed in support of workflow execution is the owner realm. To achieve this goal, we can rely on the access control mechanism in place at the owner side. This implies having the owner itself that sets its local access control mechanism to release the data to external organizations and only when these are indeed needed for a task execution. However, this might require a relevant overhead at owner side. Indeed, for each workflow, local authorizations

should be updated in order to model its information sharing needs. This is further complicated, if we consider the temporal and dynamic aspects of this information sharing. To limit as much possible this overhead, the activation of authorizations should be closely related to workflow execution. A promising option is that of having the activation of authorizations managed at the coordination layer implemented by the blockchain framework.

In the following section, we show how the above requirements have been considered in the design of the proposed architecture.

## 4.3 Workflow-based data sharing through blockchain

The key idea of our proposal is that of having both the coordination and authorization layers implemented by two smart contracts, namely, $WF_{Engine}$ and $SC_{T_{invoke}}$. Our implementation of smart contracts relies on Hyperledger Fabric [8].

We assume that, at the beginning, organizations involved in the collaboration agree on a collaborative workflow. This is then submitted to an off-chain component, called *Deployer*, in charge of the creation of the tailored $WF_{Engine}$ smart contract. The obtained smart contract is then deployed to the blockchain.[1]

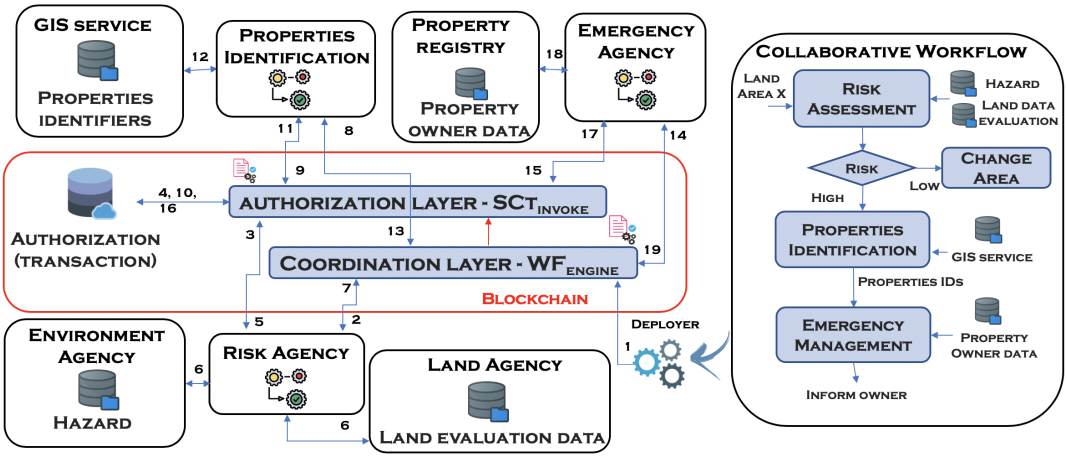



FIGURE 4.3: Blockchain-based controlled information sharing in inter-organizational workflows

Given a workflow WF, the $WF_{Engine}$ smart contract is created such to execute WF, which implies to invoke the initial task and then, based on the flow, decides which is the next task to be invoked till reaching the final state. According to the proposed solution, the core logic of $WF_{Engine}$ is mainly related to the coordination of tasks invocation, whereas each task invocation is implemented into a separate smart contract, i.e, $SC_{T_{invoke}}$. This implies that to invoke a given a task $T$, the $WF_{Engine}$ has to create and deploy a new $SC_{T_{invoke}}$ smart contract, by passing information about the invocation (e.g., information on $T_{exec}$, the possible input parameters).

Let us consider, as an example, the workflow in Figure 4.2. To invoke the first task $T1$, $WF_{Engine}$ creates a smart contract $SC_{T_{invoke}}$ where $T1_{exec}$ is the risk agency and the input parameter is "$land = X$". As such, the core logic of $SC_{T_{invoke}}$ is related to

[1] Regarding the adopted blockchain, the proposed solution works both on public and permissioned blockchain. However, the latter is more appropriate in case the information contained in the workflow (aka its instructions) is considered sensitive. In this case, the permissioned blockchain might consist only of peers made available by organizations involved in the collaboration.

task invocation which requires the following main functionalities: (1) deployment and enforcement of $T_{auth}$ authorizations, thus making $T_{exec}$ able to access resources in $T_{res}$; (2) invocation of $T_{exec}$ to perform task $T$; (3) deactivation of $T_{auth}$.

Let us now introduce how $SC_{T_{invoke}}$ implements the most critical steps, that is, (1) and (3). As introduced in Section 4.2, the idea is to have the authorizations and related enforcement directly in the blockchain, implemented into $SC_{T_{invoke}}$. Thus, by exploiting the blockchain distributed consensus, $Owner_r$ can simply check the resulting transactions as a proof of the correct enforcement of authorizations. However, as a further requirement, we have to support the dynamic allocation of resources. Indeed, according to the considered scenario, it could be the case that $T_{exec}$ should be authorized to access a resource $r$ of $Owner_r \in \mathcal{O}$ that was not determined before the task execution. As an example, the emergency crisis agency can access only the information of the owners whose properties are in a risky area. To satisfy this requirement, we assume that, when an organization $O$ agrees to participate in the WF, it exposes information about those resources that it will make available, that is, resources that $\mathcal{O}$ authorizes to release for the collaboration. In particular, we assume that, for each resource $r$, $O$ provides its profile, which can be viewed as a set of $(attribute, value)$ pairs. These profiles are designed to be used as indexes for resource retrieval during the collaboration.

**Example 3.** Let us consider again the environment agency providing information on flood risk. This organization might participate to the collaboration providing information related to a given limited area (e.g. a given province/region/state). We can assume this information includes several types of data (e.g., topography and hydrogeology properties of existing sources of water) and that these data are organized according to their GIS location. The environment agency might provide these data based on unit area. Nevertheless the unit's granularity, each unit area represents a resource and its profile consists of GIS points that delimit the unit. Assuming circles are used as unit area, each resource $unit_j$ has the profile: $\{(GIS_{center}, GIS_{data}), (radius, int)\}$.

Furthermore, we assume that, when a task $T$ is invoked, as a first step $T_{exec}$ determines its information needs and encode them as a Boolean expression on available resource profiles. As an example, the risk agency that has to execute the risk assessment for land $X$ (i.e., task $T1$), encodes its information needs for the environment agency as a Boolean expression on GPS points (e.g., to retrieve flood data on units included in land X). Once $SC_{T_{invoke}}$ receives $T_{exec}$'s information needs (aka the Boolean expression), it evaluates this expression on the set of resource profiles available for WF. At this purpose, we assume that each organization stores their resource profiles in the blockchain into a set of transactions. We exploit a component made available by the target blockchain for an efficient transactions retrieval. It is relevant to note that the strategy to store and query transactions in the blockchain may be different. [2]

This evaluation determines which resource $T_{exec}$ is indeed authorized to access. As an example, for task $T1$, $SC_{T_{invoke}}$ evaluates the Boolean expression returned by the risk agency, $T1_{exec}$, on $unit$ profiles. The authorized resources are the set of units included in land $X$.

The results of this evaluation are stored into a transaction. This transaction contains information on deployed authorizations $T_{Auth}$, namely: $T_{exec}$'s identity (e.g., the

[2] In this chapter, we leverage on a component of the hyperledger Fabric [8] architecture, i.e., CouchDB, which provides to each node an indexed view of blockchain transactions.

risk agency), the boolean expression representing its information needs (e.g., GIS expression to delimit area $X$), the id of resources satisfying the boolean expression (e.g., ids of units included in area $X$). When $SC_{T_{invoke}}$ is deployed into the blockchain, the participants validate its corrected execution. Thus, once $Owner_r$ receives from $T_{exec}$ the access request complemented with the address of transactions resulting from $SC_{T_{invoke}}$ execution, it will check the smart contract evaluation to determine which subset of its resources $T_{exec}$ is authorized to access.

According to this approach, organizations involved in the collaboration have to previously authorize the access to part of their resources, by making their profiles available for the collaboration. However, it is relevant to note that the proposed solution ensures that these resources are accessed and consumed only when the task is indeed executed. Moreover, since the evaluation of $T_{exec}$'s information needs is run on smart contracts, we obtain an audit of which information is required during the workflow execution. This allows transparency, in that a-posteriori checks can be performed to validate whether an executor asked more than expected, preventing thus to break the least privilege principle.

Since, in Hyperledger Fabric it is not possible to write a smart contract that creates another smart contract, we assume that both $WF_{engine}$ and $SC_{T_{invoke}}$ smart contracts are already deployed by the *Deployer* component. In particular, to trigger the workflow execution we assume that *Deployer* invokes $WF_{engine}$. Then, for each task to be executed, $WF_{engine}$ notifies the corresponding $T_{exec}$, which has then to invoke $SC_{T_{Invoke}}$. The notification process in Fabric leverages on the event service. Once $T_{exec}$ catches the notification, it invokes the $SC_{T_{Invoke}}$ smart contract, by passing as input the Boolean expression representing its information needs. We assume this expression is encoded as a regular expression (RE). Then $SC_{T_{Invoke}}$ checks if $T_{exec}$ is the task executor for the current task. If so, it performs an evaluation on the $RE$ and resource profiles on the blockchain. This is performed by the query mechanism provided by Fabric (through couchDB), which iterates the list of resource profiles, evaluating on it the $RE$. As a result, $SC_{T_{Invoke}}$ returns to $T_{exec}$ a transaction stored on the blockchain with the following structure: $T_{auth} = \{T_{executer}, RE, AuthorizedResources, state\}$ where: *Authorizedresources* contains the ids of the resources satisfying the boolean expression, $T_{executer}$ is $T_{exec}$'s id, $RE$ is the regular expression representing $T_{exec}$'s information needs, whereas *state* contains a state variable storing the actual state of the authorization (i.e., active, inactive). When the transaction is deployed, the state variable is set to 'active'. Once the task is terminated, $WF_{Engine}$ changes this state by modifying the state variable in $T_{auth}.state$ to 'disable'. This allows to synchronize the activation/deactivation of authorizations according to task execution. As soon as $T_{exec}$ terminates the task execution, it invokes $WF_{Engine}$ with the returned values. This, as a first step, updates the state variable in the $T_{auth}$ transaction of the terminated task. As such, resource owners will be aware that this task is no more active and, as a consequence, also the corresponding authorizations. Then, $WF_{Engine}$ proceeds to the next task invocation.

## 4.4 Evaluation

In this section, we provide results of a set of experiments we carried out to test the feasibility of our proposal. At this purpose, we have implemented $WF_{engine}$ and $SC_{T_{invoke}}$ smart contracts in the Hyperledger Fabric blockchain (v1.4.6), using

Go[3](v1.12.12 linux/amd64) programming language. We set up an experimental scenario on a PC running Ubuntu 18.04 and equipped with a quad-core Intel i3 CPU 550 @ 3.20Ghz. Thanks to the Docker containerization of Hyperledger, we could emulate 2 organizations, each with 2 peers, all in the same PC. The aim of the experiments is to provide an estimation of the $SC_{T_{invoke}}$ invocation time and the data size required on the blockchain. In contrast, due to the logic of $WF_{engine}$ its invocation time is negligible. We remark that in Fabric we cannot create a smart contract within another smart contract, as such $WF_{engine}$ must be invoked by $T_{exec}$ for every task invocation. Therefore, the experiments take into account one execution of $WF_{engine}$, since the overhead will increase linearly with the number of invocations.

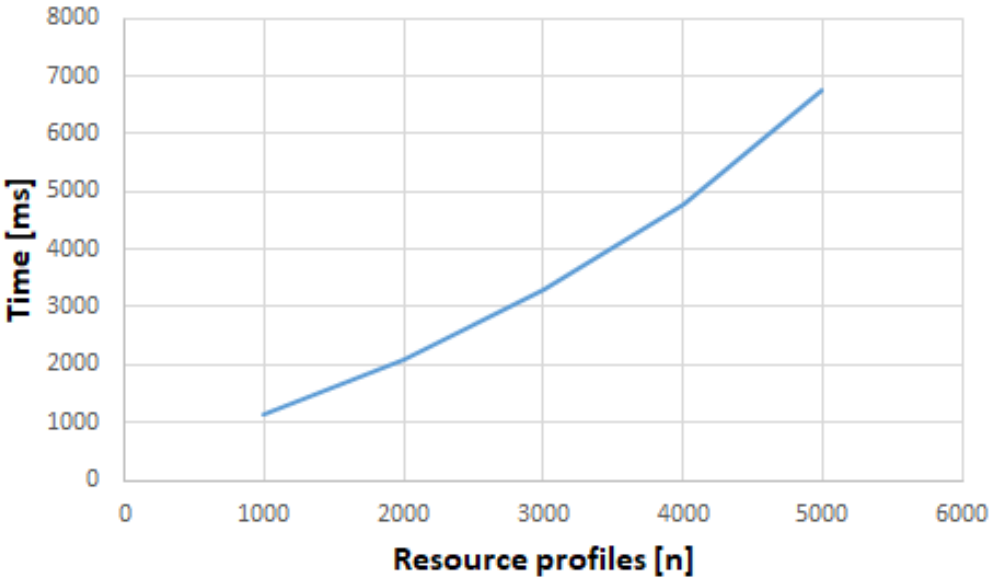



FIGURE 4.4: Invocation time by varying the number of resource profiles involved

We run several experiments varying multiple dimensions, to show how these impact the estimation. As first experiment, we have varied the number of resource profiles on the blockchain, whereas the number of attributes of each resource has been fixed to 50, and the number of clauses in the RE has been fixed to 20 predicates. Figure 4.4 shows the invocation time, expressed in milliseconds (i.e., y-axis), by varying the number of resource profiles involved (i.e., x-axis). In case of 1000 resources, the time is 1123,79ms.

In the second experiment, we have varied the number of resource profiles on the blockchain, whereas the number of attributes of each resource has been fixed to 50, and the number of clauses and predicates in the RE has been fixed to 20 predicates. Figure 4.5 shows the size of the blockchain, expressed in megabyte (i.e., y-axis) by varying the number of resource profiles involved (i.e., x-axis). In case of 1000 resources, the dimension is 21,3 MB. In the third experiment, we have varied the number of resource profiles on the blockchain, whereas the number of attributes of each resource has been fixed to 50, and the number of clauses and predicates in the RE has been fixed to 10 clauses with 2 predicates each. Also, we run the experiment with 1, 4 and 8 clients. Figure 4.6 shows the throughput, expressed in tx/s (i.e., y-axis),

[3]https://golang.org/

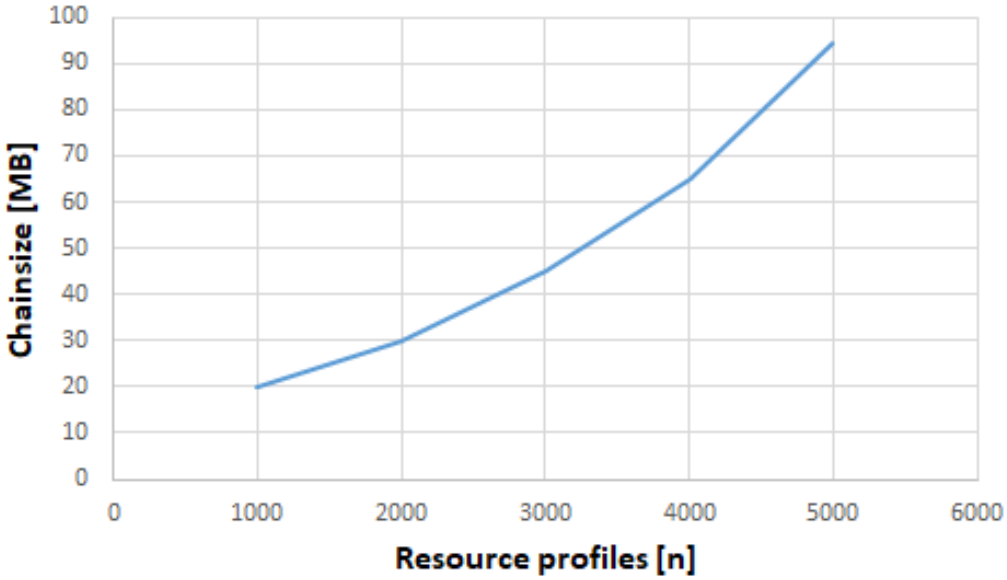



FIGURE 4.5: Size of the blockhain by varying the number of resources profiles


FIGURE 4.6: Throughput by varying the number of resource profiles

by varying the number of resource profiles involved (i.e., x-axis). In case of 1000 resources, with 8 clients, the throughput is 2,2 tx/s.

We note that these experiments are heavily dependent by the experimental scenario. However, even with the proposed experimental scenario, our proposal is able to perform the tasks with a relevant amount of resource profiles, up to 5000. Normally each peer has to be executed on a stand alone server/machine to obtain greater performance and manage more than 5000 profiles.

# Chapter 5

# Blockchain-based Privacy Enforcement in IoT domain

The Internet of Things (IoT) pervades our lives every day and has given end users the opportunity of accessing personalized and advanced services based on the analysis of the sensed data. However, IoT services are also characterized by new challenges related to security and privacy because end users often share sensitive data with different consumers without precise knowledge of how they will be managed and used. To cope with these issues, we propose a blockchain-based privacy enforcement framework where users can define how their data can be used and check if their will is respected without relying on a centralized manager. In this chapter, we introduce a blockchain-based privacy enforcement platform, including a tailor-made access control and workflow engine for IoT context.

In Section 5.1 we present the general architecture of the platform which is made up of two layers. The two layers are discussed in Section 5.2, privacy enforcement, and Section 5.3, data release, respectively. The security discussion is in Section 5.4, instead in Section 5.5 we show the evaluation of the experiments, simulating different scenarios and showing the feasibility of our approach.

## 5.1 Architecture

In this section, we introduce the overall architecture of the proposed blockchain-based privacy preferences enforcement framework, shown in Figure 5.1. The key idea is that data owners can leverage on blockchain for privacy compliance before their data are sent to consumers. We assume that both devices of data owner and consumers are registered on the blockchain. More precisely, we model a smart environment at the data owner side a set of connected IoT devices, hereafter IoT network,

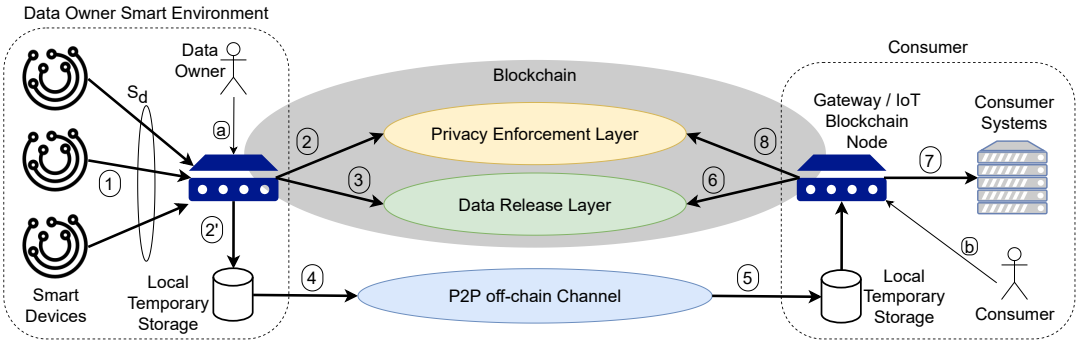



FIGURE 5.1: Blockchain-based privacy enforcement workflow

owned by a given user. The IoT network can sense data from the user environment, eventually locally elaborate them, and then send them to the consumer servers. We further assume that the IoT network is connected to consumer servers via a limited set of special IoT devices, called gateways. In our proposed solution, the gateways become the point of contact of an IoT environment with the blockchain. This implies that they also act as blockchain nodes, called IoT blockchain nodes, in addition to performing their gateway's functions. Data collected by the gateways are complemented with metadata encoding the owner's privacy preferences. Consumers register their privacy policies into the blockchain.

IoT manufacturers play a key role in the privacy enforcement process. They know their products in detail, for example, which network ports they use, which endpoints they connect to, which information they process and which instead they send to consumers, etc. Manufacturers can leverage the MUD standard to define the behavior of their IoT devices at internetworking level (e.g., MAC address, IP address, network port). With respect to data privacy, in our framework, the manufacturer can insert a by default privacy preference, called system-defined privacy preference *pps*, using the custom field provided by MUD (see, Appendix B.1). This represents a privacy preference defined with the aim of providing the first level of privacy, by design and by default, to unaware users and therefore being compliant with regulations, such as the GDPR[1]. In any case, the data owner is free to add further restrictions to system-defined privacy preferences, by specifying an owner-defined privacy preference *ppo*. In particular, the framework provides the functionalities to merge the owner and system-defined privacy preferences before performing a compliance check. As it will be described in Section 5.2, this process is carried out via a dedicated smart contract deployed in the blockchain. Hereafter, when we talk about privacy preference, we will refer to the one generated by the combination of the system-defined and the owner-defined privacy preferences.

We also rely on smart contracts for the enforcement of privacy preferences. In particular, the gateways analyze the collected data, and when they find a new privacy preference (e.g., a privacy preference that has not yet been evaluated), they trigger the execution of a smart contract implementing the privacy compliance check. This is designed to verify whether the consumer policy satisfies the constraints specified by the data owner in his/her privacy preference. Finally, we save on-chain the proof of the results returned by the compliance check for auditing purposes.

In devising our framework, we have to consider that, by design, information stored in the chain is distributed among all blockchain nodes. Each node can see what is saved on-chain as well as the smart contract contents. The privacy enforcement does not treat sensitive information, and therefore any node can execute the smart contract without affecting the data owner's privacy. Moreover, the proposed architecture leverages on a permissioned blockchain that can be configured such that only given stakeholders are authorized to join the privacy compliance check.[2] Once privacy enforcement has been executed (i.e., its smart contract has been validated), we exploit the blockchain to enforce the data release process. This implies to release owner's data only to consumers that satisfy his/her privacy preferences. Since data could be sensitive, we cannot store them directly in the blockchain. For this reason, we assume that once the data have reached the gateway, they are kept

[1] General Data Protection Regulation - EUR-Lex 32016R0679: https://eur-lex.europa.eu/eli/reg/2016/679/oj

[2] Permissioned blockchains allow us to have an integrated identity management system relying on public key infrastructure to identify authorized stakeholders.

in local temporary storage, at the gateway side, waiting to be released to the authorized consumers. To coordinate the data release, we need a communication channel able to directly connect the gateway holding the data to the blockchain node corresponding to the authorized consumer. This channel must be private, avoiding thus any other nodes in the blockchain to access exchanged data. At this purpose, we exploit the *private data* mechanism, natively supported by Hyperledger Fabric. Thanks to this, Hyperledger Fabric can create peer-to-peer links between two or more nodes to exchange off-chain information, keeping on-chain an evidence of the data exchange. In particular, all communications take place via an encryption layer, established through the Transport Layer Security (TLS). Therefore, when the data owner's privacy preferences have been verified, another smart contract takes care of data release. It is executed directly by the gateway hosting the data to be released. Its execution aims at moving the data from the temporary local storage to the authorized consumer via the P2P channel supported by Hyperledger private data. The sensitive data are processed only within the data owner's IoT blockchain node (i.e., the gateway acting as blockchain node). The remaining nodes can only see the hash of this data, stored on the blockchain as a result of the smart contract execution. To enforce the above-mentioned steps (i.e., privacy enforcement and data release), we leverage on a permissioned blockchain to create two groups of blockchain nodes, namely the privacy enforcement and the data release layer (see Figure 5.1) to which we assign different privileges on transactions/smart contracts. Nodes on the privacy enforcement layer implement the privacy compliance check, whose outcome is used by nodes on the data release layer. Both layers are discussed in detail in Sections 5.2 and 5.3, respectively.

## 5.2 Privacy Enforcement Layer

In this section, we illustrate the three phases carried on by the privacy enforcement layer: privacy preference enforcement, tuple grouping, and enforcement audit.

### 5.2.1 Privacy preference enforcement

As depicted in Figure 5.1, data sensed by IoT devices are sent to gateways, that act as blockchain nodes (see step ① in Figure 5.1). IoT devices send their data inside a data tuple $t_d$, formally defined as follows.

**Definition 3** (Data tuple)**.** Let $S_d$ be a stream containing data sensed by an IoT device. A data tuple $t_d$ in $S_d$ has the following structure: $\langle idS, sn, d, hash(d)\rangle$, where $d$ is the sensed data, $hash(d)$ is the hash value of $d$, $idS$ is the id of $S_d$, whereas $sn$ represents the sequence number of $t_d$ in $S_d$.

We use hashes to let consumers check that the received data are exactly those sent by IoT devices and have not been manipulated or corrupted (see Section 5.2.3). To avoid further overload of the blockchain, the hash values are directly computed by IoT devices. For instance, the tuple $t_d = \langle stream01, 32, \text{“}heart\text{-}rate : 60\text{”}, \text{“}de6960c7\text{”}\rangle$ can be used to encode heart rate information of a patient equipped with a wearable device to sense heartbeats. Before sending their data, data owners must register their data streams on gateways. Consumers can then subscribe to registered data streams. This subscription is subject to data owner acceptance. Once the data owner accepts a new subscribed consumer, he/she communicates a vector of subscribed consumers'

**consumerVector**[3] to the gateway (see step ⓐ in Figure 5.1). The vector is formed by consumer identifiers $idC$ and wrapped into a tuple $t_{cv}$ : $\langle idS, \textbf{consumerVector}\rangle$, where $idS$ is the target stream identifier. On the other hand, consumers register their privacy policies in the blockchain (see step ⓑ in Figure 5.1). A consumer's gateway sends to the blockchain the privacy policy $p$ related to a stream $idS$ through a privacy policy tuple $t_p$ with the following structure: $\langle idS, idC, p\rangle$, where $idC$ is the consumer identifier, $idS$ is the data stream identifier, whereas $p$ denotes a privacy policy specified as explained in Section 2.2.1. When the blockchain receives $t_p$, it executes a tailored smart contract to store the tuple with the identifier $idTp$ and to create an association between the consumer $idC$ and the stream $idS$.

We recall that, to make the task of privacy preferences specification easier, we assume that IoT manufacturers create system-defined privacy preferences, with the aim to also protect those data owners with little awareness of the privacy risks related to data disclosure. Such preference is stored into the MUD extension field. Moreover, the preference is also stored in the blockchain for future audits (e.g., in case the MUD file is no longer available on the manufacturer server). A skilled data owner can specify his/her own privacy preferences, called owner-defined privacy preferences. An owner-defined privacy preference complements the system-defined privacy preference, making it more restrictive. The gateway, upon receiving the data owner's privacy preference (see step ⓐ in Figure 5.1) and catching the MUD URL coming from the corresponding IoT device, sends this information to blockchain injecting into the corresponding stream a privacy preference tuple $t_{pp}$, formally defined as follows.

**Definition 4** (Privacy preference tuple)**.** Let $S_d$ be a stream containing data sensed by a smart device. A privacy preference tuple $t_{pp}$ in $S_d$ has the following structure: $\langle idS, sn, mudUrl, ppo\rangle$, where $idS$ is the id of $S_d$, $sn$ represents the sequence number of $t_{pp}$ in $S_d$, $mudUrl$ is the URL pointing to the manufacturer's MUD file of the corresponding IoT device, and $ppo$ is the owner-defined privacy preference, specified according to the model in Section 2.2.1.

In case of legacy devices, which are not compatible with the MUD standard, the $mudUrl$ field will be left blank. An empty $ppo$ field means that the data owner has not specified his/her own privacy preference. When both the $ppo$ and $mudUrl$ fields are empty, it is equivalent to not execute any privacy enforcement.

The privacy enforcement smart contract (see Pseudocode 1[4]) includes the *submitPrivacyPreference()* function that generates a unique $pp$ starting from the input privacy preferences provided in $t_{pp}$. It is started when the blockchain receives a new $t_{pp}$. As first step, it checks that the sequence number is as expected, to avoid inconsistencies and conflicts of privacy preference (see, Line 6). When the tuple $t_{pp}$ contains only the data owner-defined privacy preference, that will become the actual $pp$. The same holds if only a system-defined preference is contained into the tuple. When the privacy preference tuple contains both a system and an owner-defined preference, the resulting $pp$ is obtained by combining these two in such a way that they must be both satisfied in order to deliver the protected data to the requesting consumers. Preferences combination is implemented by the $ppCombine()$ function (Pseudocode 1). The generic "$join***(pps.*, ppo.*)$" function combines each element of the two privacy preferences. At the end, the function returns the resulting $pp$, which is saved in the blockchain as a parameter of $t_{pp}$. In addition, we also

[3] All vectors are denoted in lowercase bold.

[4] Functions "$getBc***(k)$"/"$putBc***(k,v)$" read/write the data with key $k$ and value $v$ on the blockchain, respectively.

append the system-defined privacy preference $pps$ retrieved from the manufacturer server. The execution of the privacy enforcement smart contract is then validated by the network via distributed consensus, ensuring thus the correct compliance check.

**Example 4.** Returning to Example 1, a privacy preference tuple for the heart rate can be: $t_{pp} = \langle stream01, 24, \text{“}manufacturerUrl\text{”}, dataOwnerPP \rangle$, where "stream01" is the stream id, "24" is the tuple id, "manufacturerUrl" is the url of the MUD file containing $pps$, "dataOwnerPP" is the owner-defined privacy preference, which we assume being the $pp$ of Example 1, that is: $\langle heart\text{-}rate, \{hospital\text{-}company\}, \langle \{admin, diagnosis\}, \emptyset \rangle, 90d, unshareable \rangle$. In this example the data owner grants access to his/her sensitive data. The system-defined privacy preference acts by excluding any purpose which is not GDPR compliant (e.g., to ensure that consumers cannot speculate on medical data). This can be encoded by the following system-defined privacy preference: $pps = \langle \emptyset, \emptyset, \langle \emptyset, \{marketing\} \rangle, \emptyset, \emptyset \rangle$. The combination of the two privacy preferences that will be saved on blockchain is: $pp = \langle heart\text{-}rate, \{hospital\text{-}company\}, \langle \{admin, diagnosis\}, \{marketing\} \rangle, 90d, unshareable \rangle$.

When the blockchain receives a new $t_{pp}$, the contained privacy preference $pp$ takes effect substituting the previous one. This implies to run the privacy compliance check of the new preference against privacy policies of consumers subscribed to the related stream (same $idS$). This check is performed by the *privacyComplianceChecker()* function (Pseudocode 1). The function is triggered by the *submitPrivacyPreference()* function each time a new $t_{pp}$ is received. After obtaining the privacy preference tuple and the consumer vector, *privacyComplianceChecker()* verifies which of the consumers have the authorization to receive the data in the stream to which $t_{pp}$ refers to. This is done by leveraging on the $verifyAuth()$ function (Pseudocode 1). The purpose check (see, Line 8) is satisfied if $up$ specifies an allowed purpose, that is, a purpose contained in $\overrightarrow{ip}$ (see, Section 2.2.1). Also, $dataRet$ must be less than or equal to $rt$ (see, Line 11), whereas $dataRel$ must match $tpu$ (see, Line 14). If all checks are satisfied, then the compliance check successes. The result is then stored in the blockchain. More precisely, since the function is executed for each subscribed consumer, all returned results are collected into a unique vector:

$$\begin{aligned} \textbf{check} &= (check_1, check_2, ..., check_n) \\ s.t. \quad check_j &= (idC, idTp, grant) \end{aligned} \tag{5.1}$$

where $n$ is the number of subscribed consumers, $grant$ is the value returned by $verifyAuth()$ for consumer whose id is $idC$, and $idTp$ is the privacy policy identifier. As introduced in Section 5.2.1, when $t_p$ is submitted to the blockchain, an identifier $idTp$ is generated, referenced on-chain with the keys $idC$ and $idS$. $idTp$ is retrieved from blockchain via the $getBcIdTpByIdSIdC()$ function (see, Line 19). Finally, the **check** vector is saved in the blockchain with $idCheck$ as key.

### 5.2.2 Tuple grouping

In a realistic IoT scenario often many devices in the IoT network push their tuples to the gateway simultaneously. However, blockchain might not be able to manage a high incoming rate of tuples, such as gateways might produce. Therefore, the blockchain can become a bottleneck. To avoid this, we group data tuples to which the same privacy preference applies. This selection has the purpose to lighten the workload on the smart contract and therefore on the blockchain. By grouping tuples with similar privacy protection requirements, the blockchain can process them

in one round. Indeed, when the smart contract receives a group of tuples that share the same $idS$ and $pp$, it can perform the compliance check only once and apply it to the whole group. To correctly create the tuple selection $\sigma$, we must take into account the validity range of $pp$ for a given data stream $S_d$, that is, the set of tuples to which it applies. The range of tuples to which $pp$ applies begins with the privacy preference tuple referring to $pp$ and ends with the subsequent privacy preference tuple. We call this data stream subset *privacy preference scope*, denoted as $S_{pp}$. Another important dimension is the time interval on which the selection acts. Considering different streams as input to the gateway, the selection must wait for the collection of a certain number of data tuples. We can limit the maximum waiting time and the maximum number of tuples per selection to be sent to the blockchain. By tuning these two parameters, we create queues within the gateway with different priorities and capacities. For example, in the case of streams that need low latency and small data size, such as near real-time, we can select a low accumulation time and a high number of tuples in a single selection. Otherwise, if the flow contains big size data that can be delayed, such as batch processing, the queue may have a higher grouping time than in the previous case. We have considered these two extreme scenarios, but the platform is also capable of handling mixed scenarios, by using the same approach. Obviously, different strategies must be adopted to tune the window and balance the latency according to the considered application scenario. In any case, $\sigma$ contains a certain number of data tuples collected in a time interval, that respects the constraint of having the same stream identifier and the same privacy preference applied to all its tuples. Each tuple in the selection has its own sequence number, so we can define an interval as two integers $[n, m]$, where $n$ and $m$ are the sequence numbers of the first and the last tuple, respectively. $\sigma_{[n,m]}(S_d)$ represents a selection of tuples $t_d \in S_d$ such that $t_d$ belongs to $S_{pp}$ and $n \leq t_d.sn \leq m$.

For the obvious limitations of the blockchain, we must include further optimizations with respect to the use of data hashes contained in each data tuple (see, Def. 3). Therefore, we exploit a hash function to compute and store only a digest, representative of a whole tuple selection. Let us consider a hash function $hash()$ and $n$ messages $m_1, m_2, ..., m_n$, the digest is calculated as $d = hash(hash(m_1)||hash(m_2)||...||hash(m_n))$, where $||$ denotes the concatenation operation. Keeping only one digest in the blockchain allows us to combine in a single hash all the tuples over which the control has been made, with a considerable saving of space. More precisely, from a selection $\sigma_{[n,m]}(S_d)$ we obtain the set of tuples $t_d$ composing it (see, Figure 5.2). For each $t_d$, we select only the field $hash(d)$. Finally, we compute the cascading hash on the resulting set of hash values.

After that, for each $t_d$ in $\sigma_{[n,m]}(S_d)$, we collect $d$ into a data vector $\mathbf{d}$. The data vector is saved on temporary local storage, ② in Figure 5.1, and it is not shared with anyone, but it is only accessible locally by the data owner.

Leaving aside $\mathbf{d}$ (it will be discussed in Sec. 5.3), we represent the tuple selection $\sigma_{[n,m]}(S_d)$ on the blockchain with a small size tuple suitable for on-chain storage, called chunk tuple $t_c$, whose structure is:

$$\begin{gathered}\langle idS, idT_{pp}, [n, m], idCheck, digest(\mathbf{h})\rangle \\ s.t. \quad \mathbf{h} = \prod_{hash(d)}(\sigma_{[n,m]})\end{gathered} \tag{5.2}$$

where $\mathbf{h}$ is the projection on component $hash(d)$ of selection $\sigma_{[n,m]}$, $idS$ is the identifier of the stream to which the selection refers to, $idT_{pp}$ is the selection's privacy preference identifier, $[n, m]$ is the interval of data tuples' sequence numbers belonging to the selection, $idCheck$ is the privacy compliance check identifier, whereas

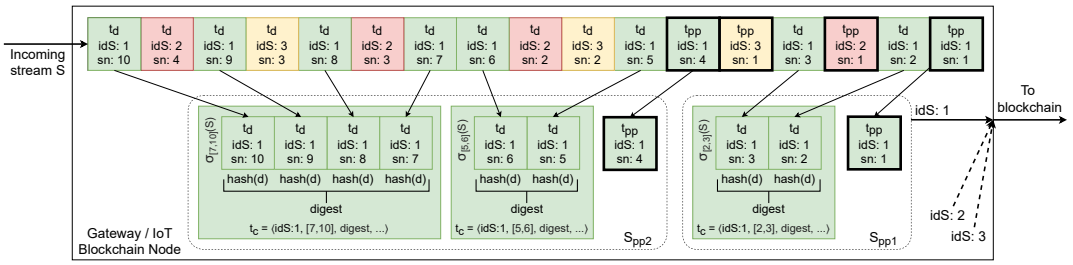



FIGURE 5.2: Tuple selection process inside the gateway

$digest()$ is the digest built on **h**. $t_c$ is created by function $dataTupleChunk()$ of the privacy enforcement smart contract (see, Pseudocode 1 and step ② in Figure 5.1). It receives $\sigma$ as input, from which it derives $idS$, **h**, and $[n, m]$. The two identifiers $idTpp$ and $idCheck$ are obtained from blockchain with key $idS$. Next, we calculate the digest (see, Line 33) and then store $t_c$, with its identifier $idTc$, on the blockchain. Figure 5.2 shows a graphic representation of the tuple selection process. At the top, each square represents a tuple, those with the bold border are privacy preference tuples, while the others are data tuples. The colors green, red, and yellow represent tuples from three different data streams. Focusing on the green stream, "$idS : 1$", we can see two privacy preference scopes $S_{pp}$, the first with a selection, and the second with two selections. For instance, the selection $\sigma_{[7,10]}(S_d)$ consists of the tuples from sequence number 7 to 10 in the stream "$idS : 1$" belonging to the privacy preference scope $S_{pp2}$.

### 5.2.3 Privacy preference enforcement audit

Our framework allows data owners to check whether the compliance of their privacy preferences with the consumers' privacy policies has been correctly carried on for a certain data tuple. Regarding consumers, they can check the outcome of their privacy policy enforcement and whether the received data set is complete.

Mainly, the data owner has the right to verify the enforcement of his/her privacy preferences over his/her data and to which consumers they have been released. To proceed with the audit verification of a specific data tuple $t_d$, the data owner needs to determine the chunk tuple $t_c$ where the outcome of the compliance check for $t_d$ is kept. Knowing $idS$, he/she can go back to $t_c$ thanks to the $sn$ component of $t_d$. Once the data owner gets $t_c$ from the blockchain, he/she can carry out the checks itself. First, the data owner can verify that the tuples contained in the range $[n, m]$ have been enforced with the privacy preference $t_{pp}$, retrieved from $idTpp$. Another check is on the vector **check**, kept with $idCheck$ on blockchain. Specifically, each consumer has his/her own $check$ inside **check** vector. By doing so, the data owner gets the list of consumers enabled to receive the data. A third more in-depth check can be carried out on each individual consumer, to re-evaluate privacy enforcement on $t_{pp}$ and $t_p$. In this case, the data owner must carry out the check on his/her own and verify that the outcome matches the one stored in the $grant$ component of the element referring to that consumer in the **check** vector.

On the other hand, the consumer receives $t_c$ and **d**. From $t_c$ he/she can see the outcome of the privacy enforcement carried on his/her privacy policy and, if he/she wants, recalculate it with $t_{pp}$ and $t_p$ for double-check, like the data owner (see, step ⑧ in Fig. 5.1). Instead, to verify the completeness of the received data set, he/she can calculate the entire digest of **d** and compare it with the one contained $t_c$. If they match, then the consumer has received all the data.

**Pseudocode 1:** Privacy enforcement smart contract

```
1  Let:
2  tpp be the privacy preference tuple;
3  idTpp be the privacy preference tuple identifier;
4  σ be the tuple selection ;
5  Function submitPrivacyPreference (tpp)
6      checkSequenceNumber(tpp);
7      mudFile = downloadMudFile(tpp.mudUrl);
8      tpp.pps = extractPps(mudFile);
9      tpp.pp = ppCombine(tpp.pps, tpp.ppo);
10     new idTpp;
11     putBcTpp(idTpp, tpp);
12     privacyComplianceChecker(idTpp);
13 end
14 Function privacyComplianceChecker (idTpp)
15     tpp = getBcTpp(idTpp);
16     consumerVector = getBcConsumerVector(tpp.idS);
17     check = new vector;
18     forall idC in consumerVector do
19         idTp = getBcIdTpByIdSIdC(tpp.idS,idC);
20         tp = getBcTp(idTp);
21         grant = verifyAuth(tpp.pp, tp.p);
22         check.add(idC,idTp,grant);
23     end
24     new idCheck;
25     putBcCheck(idCheck,check);
26 end
27 Function dataTupleChunk (σ)
28     idS = getIdS(σ);
29     idTpp = getBcIdTppByIdS(idS);
30     h = getH(σ);
31     [n, m] = getInterval(σ);
32     idCheck = getBcIdCheckByIdS(idS);
33     digest = calculateDigest(h);
34     tc = (idS, idTpp, [n, m], idCheck, digest);
35     new idTc;
36     putBcTc(idTc, tc);
37 end
38 Function ppCombine (pps,ppo)
39     new pp;
40     pp.ip = joinIp(pps.ip,ppo.ip);
41     pp.rt = joinRt(pps.rt,ppo.rt);
42     pp.tpu = joinTpu(pps.tpu,ppo.tpu);
43     return pp;
44 end
```

**Pseudocode 2:** Verify authorization function

```
1  Let:
2  pp be the data owner's privacy preference;
3  p be the consumer's privacy policy ;
4  Function verifyAuth (pp,p)
5      Let ipFlag be a boolean variable, initialized as False;
6      Let rtFlag be a boolean variable, initialized as False;
7      Let tpuFlag be a boolean variable, initialized as False;
8      if (p.up ∈ pp.ip→) then
9          ipFlag=True;
10     end
11     if (p.dataRet ≤ pp.rt) then
12         rtFlag=True;
13     end
14     if (p.dataRel = pp.tpu) then
15         tpuFlag=True;
16     end
17     if (ipFlag=True & rtFlag=True & tpuFlag=True) then
18         return True;
19     else
20         return False;
21 end
```

## 5.3 Data Release Layer

The data release layer transfers an authorized data item $d$ from the data owner IoT blockchain node to a consumer IoT blockchain node. Since $d$ may contain sensitive information, they cannot be exchanged on-chain. For this reason, we exploit a peer-to-peer (P2P) off-chain private channel to exchange data between the involved parties. For this purpose, in this chapter, we leverage on private data provided by Hyperledger Fabric. Private data allows the establishment of a P2P link between two or more nodes for data sharing, saving on the blockchain a trace of the exchange.

More precisely, the data release layer operates on the output of the privacy enforcement layer (see, Section 5.2), that is: (1) the data $d$ to be released to the intended consumers, which are saved on local temporary storage as data vector $\mathbf{d}$, and (2) the chunk tuple $t_c$, stored on the blockchain. The purpose is to send $\mathbf{d}$ to the allowed consumers. The data release is managed by a smart contract, described in Pseudocode 3, and executed by the data owner IoT blockchain node. The data release smart contract leverages the $dataRelease()$ function (see, step ③ in Fig. 5.1), which sends data to consumers according to the compliance check results. The function receives as input parameters (1) and (2) and verifies that each consumer has received a positive result from the compliance check performed by *privacyComplianceChecker()* (see, Pseudocode 1), otherwise data are not released.

First, let us get $t_c$ by $idTc$ (see, Line 5, Pseudocode 3), where is contained $idCheck$ (it represents a pointer within the blockchain to the privacy enforcement check). We use $idCheck$ to derive the vector **check** from blockchain (see, Line 6, Pseudocode 3). Inside this vector, we have the outcome of privacy enforcement per each consumer. When the smart contract finds a consumer, who is entitled to receive the data, it gets the private channel identifier and sends $\mathbf{d}$ on the P2P off-chain channel (see, steps ④ and ⑤ in Fig. 5.1) through a tuple $(idTc, \mathbf{d})$. The function $putPrivateData()$ (see, Line 10 of Pseudocode 3) takes care of sending $(idTc, \mathbf{d})$ to the consumer on an Hyperledger Fabric private channel. Since in a blockchain network any node can be vulnerable and attacked, data contained in $\mathbf{d}$ are encrypted and signed by the IoT

**Pseudocode 3:** Data release smart contract

```
1  Let:
2  idTc be the chuck tuple identifier;
3  d be the data vector;
4  Function dataRelease (idTc,d)
5      tc = getBcTc(idTc);
6      check = getBcCheckVector(tc.idCheck);
7      forall check in check do
8          if check.grant is True then
9              idChannel = getChannel(check.idC);
10             putPrivateData(idChannel, idTc, d);
11         end
12     end
13 end
```

devices that generate them. Hyperledger Fabric provides a Public Key Infrastructure (PKI) in which each node has its own identity with a public and a private key (wallet). Taking advantage of this feature, an IoT device can send through the P2P off-chain channel a symmetric key to a consumer, encrypting it with the consumer's public key. The consumer uses the Hyperledger Fabric *getPrivateData*() function in order to obtain his/her data (see, step ⑥ in Fig. 5.1). This function requires the channel identifier *idChannel* and the chunk tuple identifier *idTc* as input parameters. The consumer knows the channel identifier *idChannel* as this is released at subscription time, the chunk tuple identifier *idTc* because the P2P off-chain channel has a built-in push notification mechanism that alerts the consumer when a new chunk tuple is available. After that, **d** is sent to the consumer (see, step ⑦ in Fig. 5.1).

## 5.4 Security discussion

In this section, we discuss the security guarantees provided by each layer of our framework, as well as of the underlying infrastructure. In general, we assume that each component of our infrastructure (e.g., blockchain, gateways, and IoT devices) is untrusted and controlled by different entities that might have conflicting interests.

***Infrastructure*** To ensure secure communication among IoT devices, gateways, and blockchain peers, we leverage the authentication and encryption mechanism features provided by Hyperledger Fabric via Membership Service Provider (MSP)[5], which is based on TLS protocol and X.509 certificates.

A further key component of the proposed infrastructure is the blockchain, which might be subject to vulnerabilities and threats, e.g., double spending, smart contract coding flaws, Sybil attack, etc. In our framework, to cope with these threats we adopted the countermeasures designed for Hyperledger described in [16]. Moreover, we rely on a permissioned blockchain and, as such, blockchain's clients and peers are known, having they own identities. Therefore, they are accountable for their behaviors, and any perpetrator of abuse or malicious behavior is easily identifiable and can be banned from the network. We also use the raft consensus algorithm which is considered safe and preventing double spending by design [16]. Although the blockchain allows us to run a reliable privacy enforcement even in an

[5]Membership Service Provider - Available at https://hyperledger-fabric.readthedocs.io/en/latest/msp.html

untrusted environment, we have to consider smart contracts' security. Smart contracts are prone to various programming errors that can lead to bugs and/or vulnerabilities. They can be exploited to manipulate the workflow of the smart contract and obtain different results from those expected. To cope with this issue, we have implemented a set of strategies, such as imposing maximum time for smart contract execution, setting root privileges only where necessary, enabling access control lists (ACLs) for channel access, setting policies for chaincode lifecycle and endorsement, testing input parameters against parameter tampering, testing the chaincodes with static analysis tools (revive^CC[6] and gosec[7]).

***Privacy enforcement layer*** Attacks on the correct functioning of the privacy enforcement layer can be done by malicious IoT devices or gateways that try modifying the preference/policy tuples. Our scenario foresees different IoT devices each with its own tailored privacy preference, specifically the system-defined privacy preference *pps* (cfr. Section 5.1). For the sake of scalability, in the default setting of our framework IoT devices generate and sign privacy preference tuples. So it is possible that a compromised IoT device, generates a fake privacy preferences, in conflicts with the real owner preferences. However, the data owner is able to verify on the blockchain the stored privacy preference, sent by the IoT device, and detect the attack. This solution could fit many IoT deployment and achieve a good compromise between security and scalability. However, to cope with more risky scenarios (e.g., characterized by resource-constrained IoT devices that can be easily compromised), the proposed infrastructure supports the data owner to directly submit in the blockchain his/her privacy preferences. This would exclude any tampering by compromised IoT devices.

Another security issue is due to possible DoS attack, where IoT devices try to send an excessive number of privacy preference tuples to continuously trigger the privacy enforcement. We consider that the gateway is equipped with standard countermeasures against DoS (queuing systems, detection, and monitoring system, IoT devices' ban, log and notification, etc.), so it can detect and mitigate by avoiding sending the tuples to the blockchain. In the case that also the gateway participates in the attack, the blockchain administrators can monitor the number of transactions submitted, set rate limiter, identify, and isolate malicious IoT devices or gateway.

Besides participating in DoS attacks, a malicious gateway could modify, duplicate, or omit the privacy preference tuples. We recall that IoT devices sign these, thus any tamper by the gateway is easily identifiable by the blockchain due to signature invalidation. The privacy preference tuples duplications can be identified thanks to the sequence number that cannot be altered, since it's included in the signature. Also, the privacy preference tuples omission can be detected by proposed smart contracts (see, Line 6, Pseudocode 1), which know the expected tuple sequence number and raise an alert in case of wrong sequence number. Finally, the correctness of the privacy enforcement process relies on the correctness of the proposed smart contracts. This has been proven by Theorem 1 available in Appendix B.2.

***Data release layer*** A possible security issue in this layer is represented by data tuples sent to unauthorized entity (e.g., consumers, peers). This could happen due to (1) untrusted gateway that sends the collected data to unauthorized entity or (2) an attacker who directly eavesdrops the communication. We have to recall that data

[6] Available at https://github.com/sivachokkapu/revive-cc
[7] Available at https://github.com/securego/gosec

sensed by IoT devices are encrypted with their private keys before their release (cfr. Section 5.3). Thus, even if data are shared with unauthorized entities by an untrusted gateway or eavesdropped by attacker, these are not accessible. Furthermore, the risk of eavesdropping is limited by the TLS protocol adopted by Hyperledger Fabric. A data leakage could also occur at the IoT device, as a malicious IoT device could send the sensed data to unauthorized entity. To cope with this possibility, we leverage on MUD (cfr. Appendix B.1), by which we can limit the IoT device communication only to the gateway and make the IoT device t not directly reachable remotely. Further security problems are given by the omission and modification of data tuples by the gateway. To cope with this threat, we adopt the same countermeasures seen in the previous section, e.g., exploiting auditing, signature, and sequence numbers. Finally, the logical correctness of the data release smart contract has been proven by Theorem 2, available in Appendix B.2.

## 5.5 Evaluation

In this section, we present the evaluation of our solution with a realistic load. At this purpose, we run a set of experiments aiming at measuring: (1) the data throughput, that is, the amount of data that the blockchain can manage in a time unit; (2) the space overhead implied by the additional information (metadata) that our solution requires to insert in the original IoT device streams; and (3) the time spent in privacy preference enforcement and data release. In running the experiments, we considered two main dimensions that impact performance. The first is the number of owner's IoT devices registered in the blockchain, that is, the number of data streams to be processed. Indeed, each distinct inbound data stream requires a distinct aggregation process in order to create $t_c$. Moreover, since the compliance checks have to be performed against each registered privacy policy, the second dimension is the number of registered consumers. In our state of the art analysis (cfr. Section 3.2) we did not find any work directly comparable to our proposal, as they differ in type of infrastructure, scenario, blockchain, and adopted privacy model.

### 5.5.1 Test environment

We simulated a smart home scenario exploiting both Raspberry Pi 3[8] devices and virtual machines[9]. The adopted smart home scenario consists of four different entities: (1) a gateway collecting data owner streams; (2) blockchain peers supporting the dialogue with consumers (i.e., policy registration, stream subscription); (3) an IoT device manufacturer, and (4) a third-party entity. This latter represents those entities that are not directly involved in the data generation/release but whose peers participate in the blockchain consensus. Moreover, we assume that each entity joins the blockchain with two peers and a certificate authority[10]. For data owners, consumers, and IoT manufacturers, we have implemented a peer on a Raspberry Pi 3 device, whereas the other peers and the certificate authority run virtualized on the server. Instead, peers and certificate authorities of the third party entities run only on the server.

In the experiments, we varied the number of data owners and consumers by changing the load on the corresponding blockchain peers. For the implementation,

[8] Model B+: Cortex-A53 (ARMv8) @ 1.4GHz, 1GB LPDDR2.

[9] Running on server Intel Core i7-6700 @ 3.4Ghz, 16GB DDR4

[10] In Hyperledger Fabric, each entity must have its own certificate authority for the generation of the cryptographic information (PKI keys, certificates, etc.).

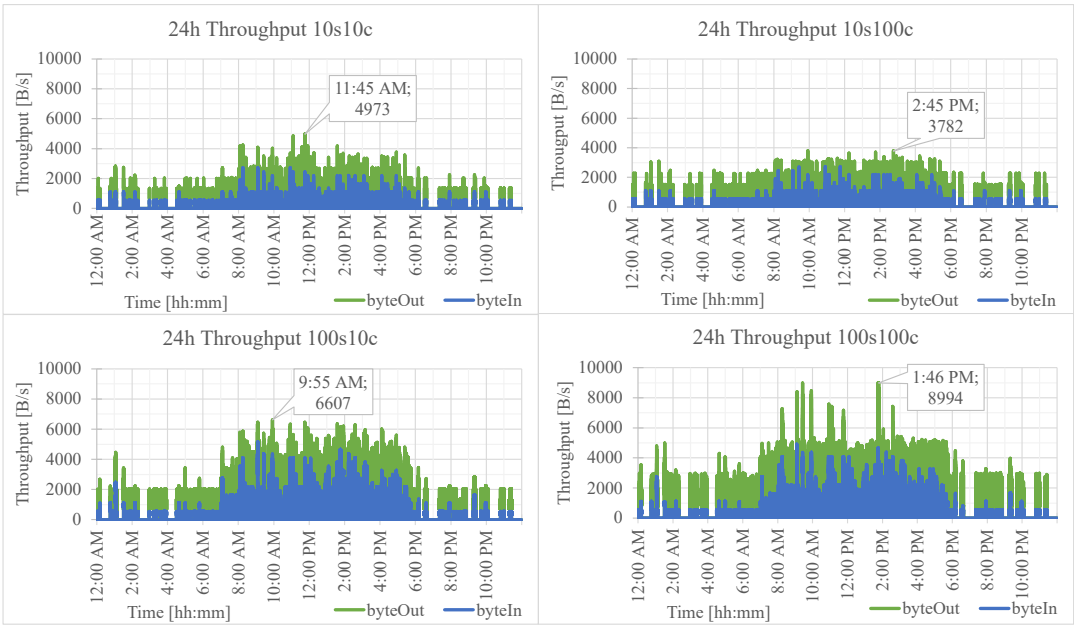



FIGURE 5.3: 24h Throughput

we used the latest stable Hyperledger Fabric release, that is, 1.4 version. We used the Raft consensus with 5 orderers, running on the server. To implement a blockchain peer on Raspberry Pi 3, we installed Ubuntu 18.04.2 4.15.0-1033-raspi2 AArch64 OS and compiled Hyperledger Fabric for arm64 platform. For the virtual peers, we used Ubuntu server 18.04 with peers virtualized on a switched Gigabit LAN. The chaincode was developed in Go[11], whereas we used the Hyperledger Fabric SDK for Node.js[12] to interface peers with clients.

The experiments were conducted by exploiting the CASAS project dataset[13], which is widely used by the scientific community. In particular, we used the "Two-resident apartment" dataset consisting of sensors' data collected from a house lived by two people. The dataset contains data gathered from 108 different devices for several days. For simplicity, we consider only 24 hours of a unique data stream. This interval is representative as the other days' pattern is similar due to daily activities' cyclical nature. To implement a more complex scenario, we duplicated this stream to simulate the flow generated by different houses. Each of them has a different data owner, with a different privacy preference. We assumed a minimum of 10 streams. At the beginning, for each stream, we assumed only one consumer with a single privacy preference. Then, we increased these numbers up to 100 subscribed consumers and 100 privacy preferences associated with each stream.

### 5.5.2 Performance results

In this section, we present the test results for throughput, space, and time overhead. We assess the performance by testing the scenario with different settings, by varying the number of data owner streams and consumers from 10 to 100 (i.e., 10, 50, 100). We changed privacy preferences and policies every 10 minutes to trigger the new compliance check computation, stressing the system.

[11] Available at https://golang.org/
[12] Available at https://hyperledger.github.io/fabric-sdk-node/
[13] Available at http://casas.wsu.edu/datasets/

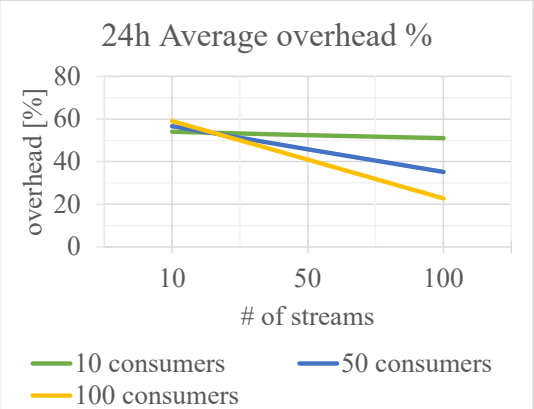



FIGURE 5.4: 24h average overhead

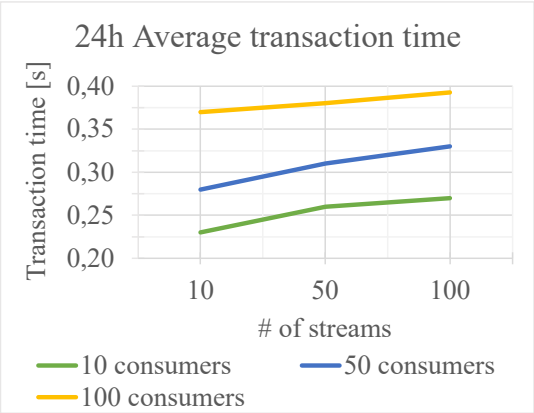



FIGURE 5.5: 24h average transaction time

***Throughput*** Figure 5.3 shows the 24h throughput for four settings, where $s$ and $c$ indicate the number of streams and consumers, respectively (e.g., 10s10c stands for 10 streams and 10 consumers setting). We omitted the 50 streams and 50 consumers configuration for lack of space, but they are taken into account in the following graphs. The *byteIn* input throughput is the number of bytes in a second, that the system through the IoT blockchain nodes can accept. The *byteOut* output throughput is the number of bytes in a second confirmed on the blockchain and that have reached the consumer. *byteOut* and *byteIn* have timestamps associated with them, the difference between these two timestamps represents the time interval during which the system performs all needed operations: selection, aggregation, privacy enforcement, data release, and blockchain consensus on each data. This time is the platform transaction time. Focusing on the outgoing stream (*byteOut*), we notice that the trend is jagged for a few consumers; instead, it becomes flat and constant for a higher number of consumers. This is because the system is not fully loaded initially and thus follows the input trend and has high overhead. When the incoming stream saturates the blockchain's speed of accepting transactions, the gateway's queuing and selection mechanism temporarily store the packets, reducing the outgoing overhead. Although there are higher peaks with few consumers, the average throughput trend is higher in the second case because the sending is continuous. The maximum throughput, on configuration *100s100c*, is around 9000 B/s. Certainly, this throughput is not comparable with the classic IoT systems that achieve higher performance but it is a first milestone in the use of the blockchain in this sector. In fact, being able to successfully manage 100 smart environments, creating about 10,000 data flows (assuming that each IoT device had its own consumer and therefore with a ratio of 1:1) is a good result that paves the way for subsequent experiments.

***Overhead*** In this experiment, we evaluated the impact that the metadata required by our solution implies. We measure overhead, after tuple grouping, as the ratio of metadata ($t_c$) to the overall value of metadata and data ($t_c$, **d**), in percentage. Figure 5.4 shows the 24h average overhead for nine settings. In general, from our experiments we have seen that a data tuple $t_d$ has a dimension of 272 bytes, of which 136 bytes of data ($d$) and 136 bytes of privacy metadata ($idS$, $sn$, $hash(d)$). Looking at the 100s100c throughput configuration with the peak value of 9000 B/s and knowing that we were able to output a **d** data vector of 8432 bytes (62 $d$ data values of 136 bytes), the overhead is about 568 bytes, that is 6.3% of the total size. In this case, we

reach the minimum overhead of the output stream. The overhead has the most significant impact when there are few transactions at the entrance and, therefore, little possibility of aggregating them. In fact, we achieved the minimum average overhead of 21% with 100 streams and 100 consumers. The overhead chart highlights the efficiency of the system on the aggregation of tuples and privacy compliance checks. The negative trend confirms the scalability of the platform as streams and consumers increase.

***Transaction time*** Figure 5.5 shows the 24 hours average transaction time for different settings. Our framework is able to reach around 6 transactions per second (0.17 s/tx), with an average of 3 transactions per second (0.33 s/tx), as shown in Figure 5.4. The average transaction time is almost constant as the number of streams changes because the data owner gateway aggregates the stream and lightens the blockchain. We note a time increase when the number of consumers increases because the compliance checks on privacy preferences and policies increase. Despite the increase in incoming streams, the graph shows that we are able to maintain an almost constant processing time and that it mainly depends on the number of consumers. This means that the system is able to scale well with respect to the number of input streams, thanks to the aggregation techniques used. Based on the obtained results, we estimate that our proposal can handle about 10 buildings[14] concurrently with 10 dwellings each and latency under 400 milliseconds. This confirms the feasibility of our approach, also considering the large margins for improvement in the use of more performing hardware, software, and networks [39].

[14]Calculation based on the Italian average number of dwellings per buildings https://entranze.enerdata.net/average-number-of-dwellings-per-building.html

# Chapter 6

# Scientific

Computer science and technological advancement are revolutionizing the world of scientific research by facilitating communication among scientists and contributing to the experimental phase. Information technologies support data-driven and information-driven science in multidisciplinary fields, such as biological, medical, physical, chemical, etc. Many scientific organizations make their services available to researchers, so to create distributed systems capable of processing and supplying large amounts of data allowing coordinated sharing among the parties involved, the so-called grids. Scientists can pool these distributed resources to achieve their scientific goal, generating a real scientific workflow (SWF). These resources include high performance computing services and large databases, used in pipelines. The sharing of information and results takes place according to the execution of a workflow. This means that the data must be accessible only to organizations that meet the release conditions to perform their task. In general, each piece of information that passes through the workflow has its own terms and conditions of use, such as the service itself that uses them. More importantly, when dealing with scientific workflows, many datasets may contain highly sensitive information (e.g., genomic data, health status, disease, and causes of death). Some organizations may be reluctant to both release and use this information without mechanisms to ensure controlled and secure sharing.

In this context, trust among the parties involved in the workflow is essential. Usually, a third party, in which all organizations place their trust, controls and manages both the execution of the workflow and the exchange of data, referred as workflow engine (WE). However, it is not always easy to find an agreement among the organization to choose the third party that will have the control, and many scientific works highlight problems of trust in centralized workflow, such as data provenance and the repeatability of experiments [36, 26].

For these reasons, we move on distributed solutions, in which the workflow is managed by the blockchain, introducing numerous advantages, such as verifiability, authenticity, immutability and non-repudiation of information. The blockchain is able to set up a reliable, immutable and replicated infrastructure between the nodes of the network, thanks to the consensus algorithm used. Each party involved in the collaboration is able to monitor and audit the workflow performed on the blockchain. From this comes that blockchain allows the provenance sharing in a collaborative and loose-relation community, giving the certainty of the experiments performed. Another aspect that particularizes scientific workflows, instead of others (e.g. business), is the large amount of data that organizations and services deal with. The successful use of the blockchain in the management of workflows, however, clashes with the current limitations of the technology itself. The blockchain is not able to manage large amounts of data on-chain, nor to process huge amount of controls for the release of data.

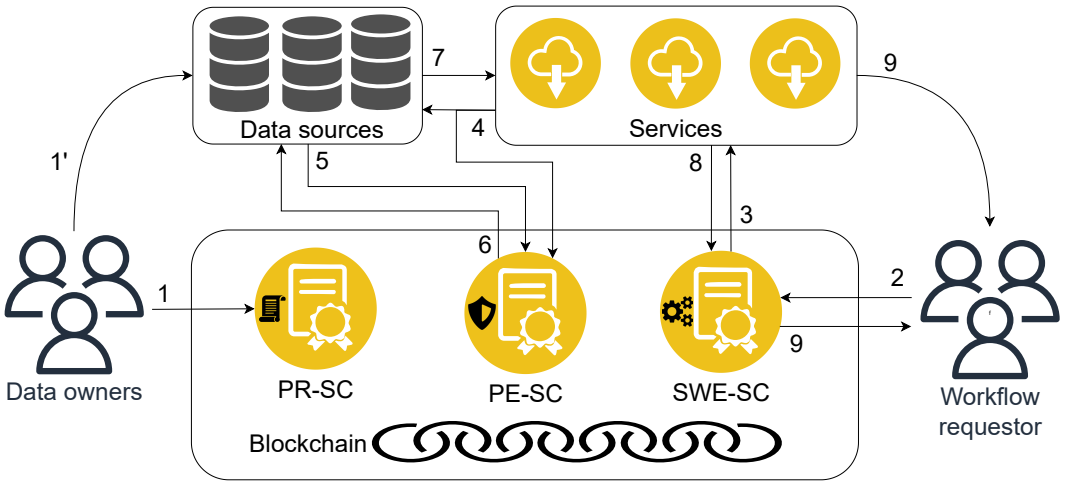



FIGURE 6.1: Overview

The approach presented in this work aims to overcome the limitations of the blockchain, optimizing the management of controls and offering a blockchain-based system for capillary privacy verification. Providing an environment that is reliable, immutable, and capable of carrying out checks efficiently guaranteeing the minimum overhead of resources with sustainable and compatible control times on the large amount of data and numerous computations.

The chapter is organized as follows. Section 6.1 provides the general architecture of the proposed solution, while the index data structure is discussed in Section 6.2. Section 6.4 gives a view of the workflow, detailed with smart contracts in section 6.5. The experiments are discussed in Section 6.6.

## 6.1 Architecture

As depicted in Figure 6.1, our reference architecture implies different actors. We have the workflow requestor (WR), which is the entity (natural or legal, such as a person or a company) invoking a SWF. A requestor could be a researcher providing an input microarray data in order to obtain new candidate target genes. Workflow tasks are executed via services delivered by different service providers (SP). Services might need to process dataset managed by external data sources (DS), which handle data collected by different individuals (data owner DO).

Our proposal uses blockchain technology to govern the workflow deployment, with the aim of ensuring that the data owners' privacy preferences are considered in every task execution. In our scenario, we assume that data owners specify a set of privacy preferences according to the privacy model in the Section 2.2.1. In particular, we assume that when a data owner uploads his/her data to a data source (see, Fig. 6.1, step 1'), he/she also registers the corresponding privacy preference in the blockchain. This is done through a dedicated smart contract called PR-SC (see, Fig. 6.1, step 1).

Similarly to other blockchain-based workflow execution platforms (cfr. [24, 71]), we assume the presence of an off-chain entity, called *Deployer*, in charge of encoding the workflow into a smart contract, called Scientific Workflow Engine Smart Contract (SWE-SC).[1] The contract is then deployed on the blockchain to manage

[1]The design of a smart contract for the workflow management has been extensively covered by previous work, such a [24] and [71]. Therefore, we do not focus on the *Deployer* component.

tasks/services invoking, following the defined workflow. Thus, when the requestor wants to execute a SWF, it invokes the corresponding SWE-SC (see, Fig. 6.1, step 2).

During the workflow execution, when a service is invoked (see, Fig. 6.1, step 3), the provider generates a request to data source only for the data needed for service execution. The request contains also the service privacy policy (Sec. 2.2.1) (see, Fig. 6.1, step 4). Once the data source has retrieved all data tuples needed by the service, it asks the blockchain to verify the privacy compliance (see, Fig. 6.1, step 5).

This request triggers the privacy enforcement that is executed via the *privacy enforcement smart contract* (PE-SC). This smart contract verifies the privacy preferences associated with requested data tuples against the service's privacy policy. Once done, PE-SC communicates to the data source the selection of tuples that can be released to the service, that is, those whose associated privacy preferences are satisfied by service's policy (see, Fig. 6.1, step 6). These tuples are then released to the requiring service (see, Fig. 6.1, step 7).[2] Once the service terminated its task, it returns the results to the SWE-SC to proceed with the next steps (see, Fig. 6.1, step 8). According to workflow specified in the SWE-SC, steps 3-8 in Figure 6.1 are repeated until the end of the scientific workflow. Lastly, the services or the smart contract can release any output data to the requestor (see, Fig. 6.1, step 9).

Having the privacy enforcement during the workflow execution might be not feasible in the context of scientific workflow. Indeed, scientific workflows are characterized by huge amount of data tuples to be processed, each of which requires a privacy compliance check. This implies an execution time linear to the data size, which is not feasible considering blockchain scalability problems (consensus protocol, communication latency, etc. [86, 79]).

At the task level, a way to optimize this process is to collect all tuples with the same privacy preference, such to perform only one privacy enforcement against the provider's privacy policy. Even if this represents a relevant improvement, it might not be enough to solve the scalability problem. Indeed, the proposed platform must be able to process multiple workflows in parallel, each consisting of several tasks. There is the need of adopt further solutions to decrease the number of compliance checks.

For this purpose, we propose an alternative solution that exploits the relationships between two privacy preferences. To better introduce this concept, let us consider two different privacy preferences $pp_1$ and $pp_2$, whose conditions are equal except for the retention times set to 12 months for $pp_1$ and 6 months for $pp_2$, respectively. Here, we can easily see that if the provider's policy satisfies the most restrictive privacy preference (i.e., the one with 6 months retention), it also satisfies the other one (i.e., with 12 months retention). By extending this concept to all preference's components, we introduce the *inclusion property* between two privacy preferences. If a privacy preference $pp_1$ is included in privacy preference $pp_2$ then if a policy $p$ satisfies $pp_1$ it also satisfies $pp_2$. This property can be used to determine if the compliance of a policy $p$ with a new privacy preference $pp_{new}$ is implied by the compliance already verified with another privacy preference included by $pp_{new}$. This brings the benefits of reducing the number of compliance checks to be performed. Inclusion property can be defined also between two privacy policies. This further reduces the number of checks to be performed during the workflow execution. Moreover, to efficiently exploit the inclusion property during the workflow

[2] In general, this release involves the exchange of large amounts of data, which has to be managed off-chain. This is out of scope of this paper, but we encourage to read [71] which proposes a secure off-chain data exchange in workflow scenarios.

execution, we introduce a index structure to organize the registered privacy preferences and policies according their inclusion properties. In this paper, we propose a multidimensional index data structure tailored for blockchain and smart contracts. This minimizes the required space and natively supports enforcement operations.

## 6.2 Inclusion property

In this section, we formally introduce the inclusion property. Conceptually, if constraints defined in a privacy preference $pp$ are *logically contained* in those of another privacy preference $pp'$, then a privacy policy $p$ that satisfies $pp$ it also satisfies $pp'$.

For the sake of simplicity, we limit the inclusion property to consider only $ip$, $rt$ and $tpu$ privacy preferences' components. We will show, how $\alpha$ and *consumer* components are considered directly during the enforcement.

Also, to facilitate the notation, we use the set $\overrightarrow{ip} = \overrightarrow{Aip} - \overrightarrow{Exc}$ (Sec. 2.2.1) instead of $ip$, having thus $pp$ defined as $= \langle \overrightarrow{ip}, rt, tpu \rangle$. The same considerations apply to the privacy policy $p = \langle \overrightarrow{up}, dataRet, dataRel \rangle$, where $\overrightarrow{up} = \overrightarrow{Aup} - \overrightarrow{Exc}$ (Sec. 2.2.1).

The inclusion property between two privacy preferences is defined as follows.

**Definition 5** (Preferences inclusion property)**.** Let $pp_1 = \langle \overrightarrow{ip_1}, rt_1, tpu_1 \rangle$ and $pp_2 = \langle \overrightarrow{ip_2}, rt_2, tpu_2 \rangle$ be two privacy preferences. We say that $pp_1$ is included in $pp_2$, denoted as $pp_1 \subseteq pp_2$, if $tpu_1 = tpu_2 \wedge rt_1 \leq rt_2 \wedge \overrightarrow{ip_1} \subseteq \overrightarrow{ip_2}$.

As a consequence, given a privacy policy $p$ and two privacy preferences $pp_1$ and $pp_2$ such that $pp_1 \subseteq pp_2$, if $p$ satisfies $pp_2$, it also satisfies $pp_2$. Let $PE()$ be the function performing the privacy performing check, we can say that if $pp_1 \subseteq pp_2$, $PE(pp_1, p)$ implies $PE(pp_2, p)$.

**Example 5.** Let us consider $pp_1 = \langle \langle 9 \rangle, 60, true \rangle$, $pp_2 = \langle \langle 8, 9, 10 \rangle, 90, true \rangle$, and $p = \langle \langle 9 \rangle, 30, true \rangle$. Here, $pp_1 \subseteq pp_2$ since: $\langle 9 \rangle$ is included in $\langle 8, 9, 10 \rangle$; the $tpu$ is the same for both the preferences, and 60 days is less than 90 days. Since $PE(pp_1, p) = true$, then it holds also $PE(pp_2, p) = true$.

The inclusion property can also be extended to privacy policies, as the following.

**Definition 6** (Policies inclusion property)**.** Let $p_1 = \langle \overrightarrow{up_1}, dataRet_1, dataRel_1 \rangle$ and $p_2 = \langle \overrightarrow{up_2}, dataRet_2, dataRel_2 \rangle$ be two privacy policies. We say that $p_1$ is included in $p_2$, denoted as $p_1 \subseteq p_2$, if $dataRel_1 = dataRel_2 \wedge dataRet_1 \leq dataRet_2 \wedge \overrightarrow{up_1} \subseteq \overrightarrow{up_2}$.

Similarly to preferences inclusion property, the privacy enforcement can take advantages also of this new property. Let consider a privacy preference $pp$ and two privacy policies $p_1$ and $p_2$, such that $p_1 \subseteq p_2$. Then, if $p_2$ satisfies $pp$, then also $p_1$ satisfies $pp$. Thus, if $p_1 \subseteq p_2$, $PE(pp, p_2)$ implies $PE(pp, p_1)$.

During the privacy enforcement, we can combine the two inclusion properties so to further reduce the number of compliance checks.

**Example 6.** Let us consider again $pp_1$ and $pp_2$ of Example 5, where $pp_1 \subseteq pp_2$. Moreover, let us consider $p_1 = \langle \langle 9 \rangle, 15, true \rangle$, $p_2 = \langle \langle 9 \rangle, 30, true \rangle$, where $p_1 \subseteq p_2$. We have that $p_2$ satisfies $pp_1$, since $p_1 \subseteq p_2$, then we can say that also $p_1$ satisfies $pp_1$. Then, since $pp_1 \subseteq pp_2$, $pp_2$ is also valid for $p_1$ and $p_2$. In summary, by verifying only that $PE(pp_1, p_2) = true$, we also verify all other combinations.

To take advantage of inclusion properties, we introduce two indexes to order the privacy preferences and policies, respectively. In the following section, we discuss the index for preferences, which is more demanding given the number of preferences possible linear to the data size.

## 6.3 Privacy Preferences Index

A first naive preferences index could be built by simply considering the preferences inclusion property (Def. 5). This could be done by creating an hierarchy on registered privacy preferences: each node corresponds to a preference, connected only with children nodes representing included privacy preferences.

Even if it would bring some benefits, we can improve it by considering that preference's components are in logical conjunction (*and*). This means that we need just a not-satisfied check to claim that the preference is not satisfied. This reduces the number of checks, avoiding unnecessary controls in case of non-compliance.

**Example 7.** Let us consider $pp_1 = \langle\langle 9\rangle, 140, true\rangle$, $pp_2 = \langle\langle 8,9,10\rangle, 90, false\rangle$ and $pp_3 = \langle\langle 7,8,9,10,11\rangle, 60, true\rangle$, where the inclusion property among them is not valid. Assuming a privacy policy $p = \langle\langle 8,9,10\rangle, 60, true\rangle$, we have to carry out three enforcement to verify compliance. Instead, by treating the privacy preference components separately, we see an inclusion relationship on $ip$: $\overrightarrow{ip_1} \subseteq \overrightarrow{ip_2} \subseteq \overrightarrow{ip_3}$. Using only the component $\overrightarrow{up} = \{8,9,10\}$ of $p$, we can discard $\overrightarrow{ip_1}$, aka $pp_1$, and examine the remaining $pp$ that have $\overrightarrow{up} \subseteq \overrightarrow{ip}$ following the inclusion relationship. In this way, we have avoided checks on other components when not needed.

Thus, rather than exploiting an hierarchy based on the inclusion property definition, we propose an index structure where we exploit separately the inclusion relationship of each components (i.e., $ip$, $tpu$, $rt$). In the following, we present how the index is generated by considering each single component.

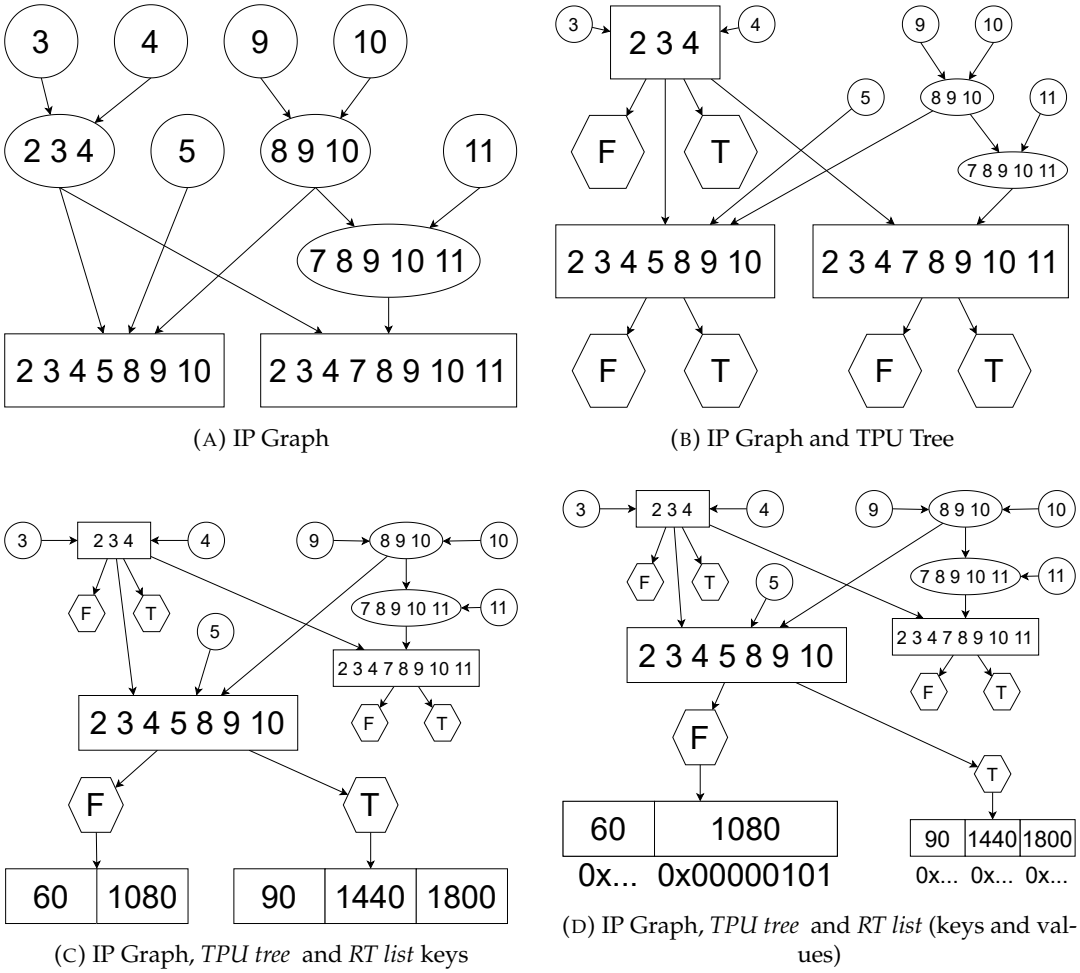
### 6.3.1 IP Graph

In this section, we focus on the privacy preference's $ip$ component.[3] We model the inclusion relationship among $\overrightarrow{ip}$ via a directed acyclic graph (DAG), called *IP graph*. Given a set of privacy preferences, this graph is built such that any vertex represents the $\overrightarrow{ip}$ of an existing privacy preference and its edges indicate the inclusion relationships, where the edge exits from the vertex whose $\overrightarrow{ip}$ is included in the $\overrightarrow{ip}$ of entering vertex. Multiple $pp$ sharing the same $\overrightarrow{ip}$ are identified by the same vertex.

In particular, given a new $pp$, this is inserted in the graph only if its $\overrightarrow{ip}$ is not already present in the graph. If this is case, we insert a node for $\overrightarrow{ip}$. According the *IP graph* definition, we need to add also those nodes representing the sets of purposes that could be included by $\overrightarrow{ip}$, if not already present. To compute these sets we exploit the purpose tree recursively as the following example clarifies.

**Example 8.** Let us consider $\overrightarrow{ip_1} = \langle 2,3,4,5,8,9,10\rangle$. According to the purpose tree $PT$ of Figure 2.2 (Section 2.2.1), from $\overrightarrow{ip_1}$ we can derive $2^{\downarrow} = \langle 2,3,4\rangle$, and recursively $3^{\downarrow} = \langle 3\rangle$ and $4^{\downarrow} = \langle 4\rangle$. All these vertexes are inserted in the graph (see Figure 6.2a), but we avoid connecting the latter directly to the starting $\overrightarrow{ip_1}$ since it has been already derived from $2^{\downarrow}$. From $\overrightarrow{ip_1}$, we also get $5^{\downarrow} = \langle 5\rangle$ and $8^{\downarrow} = \langle 8,9,10\rangle$. Again, recursively from $\langle 8,9,10\rangle$ we get $9^{\downarrow} = \langle 9\rangle$ and $10^{\downarrow} = \langle 10\rangle$. Let us suppose the insertion of a second $\overrightarrow{ip_2} = \langle 2,3,4,7,8,9,10,11\rangle$, in this case we connect the existing nodes (e.g., $\langle 2,3,4\rangle$ and $\langle 8,9,10\rangle$) and insert only the new ones $7^{\downarrow} = \langle 7,8,9,10,11\rangle$ and $11^{\downarrow} = \langle 11\rangle$.

[3] We recall that, for sake of simplicity, we refer to $ip$ as its implied intended purpose set $\overrightarrow{ip}$.




(A) IP Graph

(B) IP Graph and TPU Tree

(C) IP Graph, *TPU tree* and *RT list* keys

(D) IP Graph, *TPU tree* and *RT list* (keys and values)

FIGURE 6.2: Example of index data structure

As the example shows, this index involves the creation of $ip$ nodes for which there is no privacy preference among those registered (e.g. nodes xxxx in Figure 6.2a), but this simplifies and speeds up the enforcement. The Example 8 is shown in Figure 6.2a, where the numbers are the identifiers associated with the purposes, the nodes for which there is a privacy preference, i.e. *real nodes*, are marked as rectangles, while the others, i.e. *link nodes*, are represented by an oval/circle.

### 6.3.2 TPU tree

In the *IP Graph*, we have that all $pp$ sharing the same $\overrightarrow{ip}$, i.e. $\langle \overrightarrow{ip}, *, * \rangle$[4], are associated with the same node. By considering the $tpu$ component, we index all the $pp$ that also have the same $tpu$, i.e. $\langle \overrightarrow{ip}, *, tpu \rangle$. The $tpu$ component can only have two values, $true$ or $false$. Therefore, we can represent it through a 1-level tree, rooted on a *real node* $\overrightarrow{ip}$ and with $false$ and $true$ as leaves. We call this structure as *TPU tree*.

**Example 9.** Let us consider the Figure 6.2b which extends up the *IP Graph* in Figure 6.2a. Let us assume that two new $pp$ have been considered: $pp_1 = \langle \langle 2,3,4 \rangle, 270, false \rangle$, and $pp_2 = \langle \langle 2,3,4 \rangle, 180, true \rangle$. These have the same ip as $\langle 2,3,4 \rangle$, but two different $tpu$ values. As represented in Figure 6.2b, $\langle 2,3,4 \rangle$ has become a *real node*, and it can

[4]We use the wildcard character "$*$" to indicate any value of the corresponding component.

be used to index $pp_f = \langle\langle 2,3,4\rangle, *, false\rangle$ or $pp_t = \langle\langle 2,3,4\rangle, *, true\rangle$ in the *TPU tree*, represented with hexagons.

### 6.3.3 RT list

The $rt$ component represents the retention time expressed in days. In the enforcement phase, it must easy to select a subset of $rt$ greater than or equal to a given value. At this aim, we exploit a list, for storing an ascending ordered sequence of $rt$. We have two *RT lists* for each *ip node*, one stored in the "*F*" leaf for $tpu = false$, and one stored in "*T*" for $tpu = true$.

**Example 10.** Let us consider the Figure 6.2c which extends the *IP Graph* in Figure 6.2b. Let us assume that two new $pp$ have been considered: $pp_1 = \langle\langle 2,3,4,5,8,9,10\rangle, 60, false\rangle$ and $pp_2 = \langle\langle 2,3,4,5,8,9,10\rangle, 1080, false\rangle$. Both $pp$ have common $\langle\langle 2,3,4,5,8,9,10\rangle, *, false\rangle$, so we can use nodes $ip = \langle 2,3,4,5,8,9,10\rangle$ and $tpu = false$ to store them. The connected *RT list* (represented by sequences of $rt$ values) has to hold $rt = \{60, 1080\}$. The list for $tpu = true$, instead, hosts three new $pp$ with $rt = \{90, 1440, 1800\}$.

### 6.3.4 Task index

In order to support the privacy enforcement of multiple tasks simultaneously, the proposed index also associates to each registered $pp$ (i.e., to its components) the list of tasks where this preference is used. This list contains only those tasks that require at least a data tuple to which $pp$ applies. At this aim, at each task is assigned a *indexTask*, set as a counter starting from 1. Thus, for each $pp$ is assigned a *taskIDs*, that is, a bit-string representing all registered tasks. In particular, following the least significant bit, the j-th bit corresponds to task with $indexTask = j$ and it is set to 1 if task with $indexTask = j$ is associated with $pp$. Each *taskIDs* bit-string is associated with an item of the *RT list*, that is the $rt$ value.

**Example 11.** Let us consider the $pp = \langle\langle 2,3,4,5,8,9,10\rangle, false, 1080\rangle$ in Figure 6.2d. Moreover, let us assume that $pp$ applies to data tuples consumed in two tasks with *indexTask* "1" and "3". The corresponding *taskIDs* is $0x00000101$, where flags are 1 at positions "1" and "3". Then we save, or update, the *taskIDs* value for the $rt = 1080$ key in the *RT list* of $pp$.

## 6.4 Privacy enforcement workflow

In this section, we show how the introduced index is used to perform a privacy enforcement. In particular, we first present the enforcement for a single task $t$. Then, we discuss how the enforcement is extended to support the enforcement of multiple tasks.

### 6.4.1 Single task privacy enforcement

Let us consider a task $t$, with $indexTask = 1$, in the workflow $w$, assuming this has to be executed by a service provider $SP_t$. In order to proceed with task execution, $SP$ first requests to a data source $DS$ the access to a set of tuples and submits its privacy policy to the blockchain. As illustrated in Section 6.1, before releasing the data, $DS$ triggers the privacy enforcement. At this purpose, the blockchain generates the *privacy preference index* as described in the Section 6.2. In particular, it retrieves

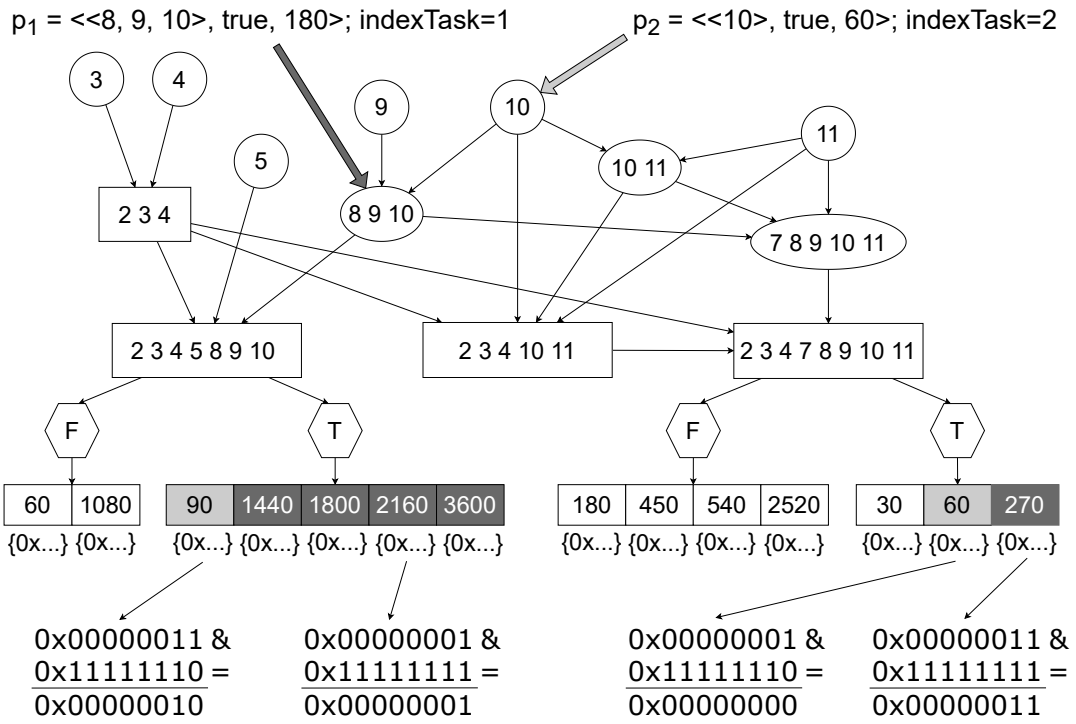



FIGURE 6.3: Privacy enforcement example

all privacy preferences applied to data tuples required by $t$. All these preferences have to be checked against the unique privacy policy $p$ of provider $SP_t$. We can see $p$ as an entry point in the *IP graph*. Indeed, we can easily identify the *IP graph* node where $pp.\overrightarrow{ip} = p.\overrightarrow{up}$.

Let us assume that *IP graph* contains an entry node $n$. If $n$ is a link node, then we continue the exploration of the other connected nodes. If $n$ is a real node, we consider its *TPU tree* and *RT list*. In particular, we follow the *TPU tree* branch based on the $p.dataRel$. Then, we select in the retrieved *RT list* all $rt$ greater than or equal to $p.dataRet$. For each of them, we check that $indexTask$ is present in $taskIDs$. If so, $p$ satisfies all privacy preferences associated with required data tuples. Thus, these tuples can be released to the $SP_t$. We repeat this check until we explore all the nodes branch off from the entry point.

**Example 12.** Let us consider the example shown in Figure 6.2d and $p = \langle\langle 8,9,10,\rangle, false, 720\rangle$. The node $\langle 8,9,10,\rangle$ is the entry point for $p$, from which all the nodes branch off. Starting from entry point $\langle 8,9,10\rangle$, we arrive at the first real node $\langle 2,3,4,5,8,9,10\rangle$, with $tpu = false$. The list returns $rt = \{60, 1080\}$. Considering $dataRet = 720$, we can select only $rt = 1080$ because greater than $dataRet$. It has an $idTask = 0x0000101$, indicating that this $pp$ is present also in the $taskIndex = 1$. From that we can release all data tuple associated with $pp = \langle\langle 2,3,4,5,8,9,10\rangle, false, 1080\rangle$. After that we continue to analyze the other real nodes.

If in the whole graph the entry point node does not exist, then we insert a new *link node* with $pp.\overrightarrow{ip} = p.\overrightarrow{up}$ value, and we proceed normally as described previously.

### 6.4.2 Multiple tasks privacy enforcement

The enforcement described in the previous section can be optimized to simultaneously support multiple tasks' execution. We limit the generation of a new *privacy*

*preference index* at each task execution and exploit the policy inclusion property (cfr. Def. 6) to reduce the compliance checks. In particular, we assume that the platform has a list of tasks $\mathcal{T}$ to be performed simultaneously and each task $t$ has its own privacy policy $p$. We are able to reduce the number of compliance checks, because given a privacy preference $pp$ that has to be verified against all policies in the privacy policy set $\mathcal{P}$, if the more restrictive policy satisfies $pp$, then all the more permissive ones satisfy it as well. More precisely, we store the privacy policies in an queue $q$ as $\{p_1, \ldots, p_n\}$, if for each pair of privacy policies $p_j$ and $p_{j+1}$ it holds that $p_{j+1} \subseteq p_j$. When a new task arrives, we check if its $p$ can be inserted in $q$. If so, we update the privacy preference index by including the privacy preferences associated with data tuples considered by the new task. On the blockchain we maintain a set of queues $\mathcal{Q}$, so that each queue $q \in \mathcal{Q}$ there is only one *privacy preference index*, and if a privacy policy $p$ does not satisfy the inclusion property in that existing queue or in others, a new queue is created for it, with a new related privacy preference index. Furthermore, for each queue, we number the contained privacy policies as $indexTask = 1$ to $p_1$, $indexTask = 2$ to $p_2$, $indexTask = 3$ to $p_3$, ..., and $indexTask = n$ to $p_n$.

**Example 13.** Let us consider to have $p_1 = \langle\langle 8, 9, 10\rangle, true, 180\rangle$ and $p_2 = \langle\langle 10\rangle, true, 60\rangle$ as shown in Figure 6.3. In that case $p_2 \subseteq p_1$, so we have in queue $q$: $indexTask = 1$ for $p_1$, and $indexTask = 2$ for $p_2$.

To proceed with privacy enforcement, first of all we select the queue of our interest from the queue set $\mathcal{Q}$. Once it has been chosen, we know which privacy preference index to use for the privacy enforcement of multiple tasks. We start by considering the most restrictive policy in $q$, i.e., $p_1$. As for the single task enforcement, we consider the node in *IP graph* with value equals to $p_1.\overrightarrow{up}$. If the node exists and it is a real node, we follow the single task enforcement. That is, we consider its *TPU tree* branch based on the $p_1.dataRel$ value. Then, we select in the retrieved *RT list* all $rt$ greater than or equal to $p_1.dataRet$. For each of them, we compare tasks indexes of $q$ with indexes in the corresponding $taskIDs$ element.

This check is done with an *And* bitwise operation, between the $taskIDs$ and a mask of onesThe mask of ones is made up of $n$ bits to 1, where $n$ is the number of tasks $t$ related to queue $q$. The $indexTask$ indicates to which task the $pp$ belongs, instead the mask is used to filter the validity of $dataRet$ on $rt$. We denote that the most restrictive privacy policy has $dataRet$ greater than all the others in the queue. So, if $rt$ satisfying $dataRet$, then $p$ for $indexTask$ greater than or equal to the one considered is also satisfied.

After that, we continue the exploration with the other descendant nodes, as well as in the case of link nodes. As in the single task case, when the entry point does not exist, a link node with value $p_1.\overrightarrow{up}$ is inserted.

The $indexTask = 1$ analysis ends when all real descending nodes have been examined. Enforcement continues by examining $indexTask = 2$ and repeating the full process from entry point $p_2.\overrightarrow{up}$. Furthermore, we shift the mask to the left of one position, so that in position $indexTask = 1$ there is 0. This means that $taskIDs$ is evaluated for values of $indexTask \geq 2$.

This time, the set of $pp$ to be analyzed is much less than the previous one and includes only those $pp$ that have remained outside the enforcement with $indexTask = 1$. This because at each enforcement with a certain $indexTask$ we also check the following ones.

The enforcement process continues by analyzing the other $p$ in $q$, also shifting the mask by one position for each $p$. When $q$ is empty or there are no more $pp$ to examine in the index privacy preference the enforcement process is finished.

**Example 14.** Let us consider again the example shown in Figure 6.3. When we analyze $p_1$, we arrive at the first real node $\langle 2,3,4,5,8,9,10\rangle$, with $tpu = true$. The list returns $rt = \{1440, 1800, 2160, 3600\}$, highlighted in dark grey. All these values are analyzed, but for example let us consider only one, $rt = 2160$. From that, we get an $idTask = 0x0000001$, indicating that this $pp$ is present only in the $taskIndex = 1$. Being the first privacy enforcement with $taskIndex = 1$, the mask is $0x11111111$, so this $pp$ is only released for $taskIndex = 1$. When we arrive at real node $\langle 2,3,4,7,8,9,10,11\rangle$, with $tpu = true$, the list returns $rt = \{270\}$ and $taskIDs = 0x00000011$. This means that the same privacy preference is present in $taskIndex$ 1 and 2. Applying the mask, this $pp$ can be released for both tasks. By examining all the additional $rt$ and nodes we complete the exploration started at $p_1$. Proceeding with $p_2$, the entry point is the node $\langle 10\rangle$. The first real node examined before returns only $rt = \{90\}$, the others have been excluded because they have already been examined previously. This $pp$ belongs to both tasks and therefore $taskIDs = 0x00000011$. This time the mask has a left shift of 1 position, indicating that $dataRet$ of $indexTask = 1$ is no longer valid in the control of $indexTask = 2$. In fact, the result is negative for every task, so this $pp$ is not released to anyone. From the mask $0x11111110$, the $pp$ is only released for $taskIndex = 2$. Moving on second real node, it returns $rt = \{60\}$ with $taskIDs = 0x00000001$ because it is present only for $taskIndex = 1$. The $rt = 270$ is excluded, because it has been checked previously. After that we continue with the other nodes.

The privacy enforcement described in the above example consideres only two tasks, but it can be generalized to $n$ tasks. The $taskIDs$ is also represented at 8 bits, but can be extended to $n$ bits.

Basically, although the graph is explored several times and there may be overlapping paths, the *RT list* becomes smaller and smaller with each iteration. When no value is present in the list, the node of the *TPU tree* is no longer examined. When both *RT lists* (true and false) are empty, it follows that the node of the IP graph is skipped.

In summary, this index data structure allows us to (I) aggregate privacy preference and privacy policy, reducing the number of checks per task, (II) not examine entire branches excluded from the entry point, eliminating unnecessary checks, and (III) speed up the control by examining a structure that dynamically simplifies and reduces itself with each iteration.

## 6.5 Blockchain workflow execution

In this section, we analyzing in detail the enforcement by presenting the smart contract PE-SC pseudocode (see Pseudocode 5) and its internal functions (see Pseudocode 6).

### 6.5.1 Index creation

As a first step (1, Fig.6.1), the data owner creates and sends his/her privacy preference $pp$ to the blockchain, using a privacy preference tuple defined below.

**Definition 7** (Privacy preference tuple)**.** A privacy preference tuple is a tuple $t_{pp} = \langle idTpp, idDO, idS, pp\rangle$, where $idTpp$ is the privacy preference tuple identifier, $idDO$ is the data owner identifier, $idS$ is the service identifier (aka *consumer* of privacy model, Sec. 2.2.1) for which privacy preference $pp$ applies.

**Pseudocode 4:** Privacy registration smart contract (PR-SC)

```
Let:
tpp be the privacy preference tuple;
idTpp be the privacy preference tuple identifier;
σ be the tuple selection ;
Function submitPrivacyPreference (tpp)
    putBcTpp(tpp.idTpp, tpp);
end
```

**Pseudocode 5:** Privacy enforcement smart contract (PE-SC)

```
Let:
tpp be the privacy preference tuple;
idTpp be the privacy preference tuple identifier;
σ be the tuple selection ;
Function dataRequest (tr)
    qList = getBcQList(tr.idS);
    idQ = insertPInQ(tr.p,qList);
    indexIP = getBcIndexIP(idQ);
    if not indexIP[tr.p.up] then
        insertIP(tr.p.up,indexIP);
        putBcIndexIP(idQ,indexIP);
    end
    putBcIdQ(tr.idT,idQ);
    putBcQList(tr.idS,qList);
    putBcTr(tr.idTr, tr);
end
Function dataReply (trp)
    idQ = getBcIdQ(trp.idT);
    indexTask = getBcIndexTask(trp.idT);
    indexIP = getBcIndexIP(idQ);
    indexTPURT = getBcIndexTPURT(idQ);
    forall idTpp in trp.idTppTd do
        tpp = getBcPp(idTpp);
        if tpp.idS == trp.idS then
            insertIP(tpp.ip,indexIP);
            insertTPURT(tpp.ip,tpp.tpu,tpp.rt,indexTask, indexTPURT);
        end
    end
    putBcIndexIP(idQ,indexIP);
    putBcIndexTPURT(idQ,indexTPURT);
    putBcTrp(trp.idTrp,trp);
end
Function privacyEnforcementFast (idS)
    qList = getBcQList(idS);
    idQ = selectQ(qList);
    indexIP = getBcIndexIP(idQ);
    indexTPURT = getBcIndexTPURT(idQ);
    enforcedPP = {};
    forall indexTask, p in qList(idQ) do
        explorePE(indexTask,p.up,p.dataRel,p.dataRet, indexIP,indexTPURT,enforcedPP);
    end
    idTpe = new id;
    tpe=(idTpe,idS,idQ,enforcedPP);
    putBcPE(idTpe,tpe);
end
```

**Pseudocode 6:** Internal function of privacy enforcement smart contract (PE-SC)

```
Let:
tpp be the privacy preference tuple;
ip be the ip of tpp;
tpu be the tpu of tpp;
rt be the rt of tpp;
Function insertIP (⃗ip,indexIP)
    ip̄=derivIP(⃗ip);
    forall ⃗ip′ in ip̄ do
        if not indexIP[⃗ip′] then
            insertIP(⃗ip′,index);
        end
        visited={};
        visited[⃗ip]=true;
        insertIPDesc(⃗ip′,⃗ip,indexIP,visited);
        indexIP[⃗ip′][⃗ip]=true;
    end
end
Function insertIPDesc (⃗ip′,⃗ip,indexIP,visited)
    forall ⃗ip″ in indexIP[⃗ip′] do
        if not visisted[⃗ip″] then
            visited[⃗ip″]=true;
            if ⃗ip″ ⊆ ⃗ip then
                indexIP[⃗ip″][⃗ip]=true;
            else if ⃗ip ⊆ ⃗ip″ then
                indexIP[⃗ip][⃗ip″]=true;
            else
                insertIPDesc(⃗ip″,⃗ip,indexIP,visited);
            end
        end
    end
end
Function insertTPURT (⃗ip,tpu,rt,indexTask,indexTPURT)
    taskIDs = indexTPURT[ip][tpu].get(rt);
    taskIDs = taskIDs | (1<< indexTask);
    indexTPURT[ip][tpu].set(rt,taskIDs);
end
Function explorePE (indexTask,⃗ip,tpu,rt,indexIP, indexTPURT,enforcedPP̄)
    forall ⃗ip″ in indexIP[⃗ip] do
        explorePE(indexTask,⃗ip″,tpu,rt,indexIP, indexTPURT,enforcedPP̄);
    end
    if ∃ skipList=indexTPURT[ip][tpu] then
        forall (rtKey,taskIDs) = skipList.gt(rt) do
            lblPP=genLabelPP(ip,tpu,rtKey);
            enforcedPP̄[lblPP]= taskIDs & (ones ⟨⟨ indexTask);
        end
    end
end
```

PR-SC receives $t_{pp}$ as parameter of function *submitPrivacyPreference()* (Line 5, Pseudocode 4) and saves it on-chain with the identifier $idTpp$ chosen by the data owner: $(idTpp, t_{pp})$[5]. In parallel, the data owner sends his/her data to the data sources (1', Fig.6.1), using a data tuple defined as the following.

**Definition 8** (Data tuple)**.** A data tuple is a tuple $t_d = \langle idTd, idDO, idTpp, d \rangle$, where $idTd$ is the data tuple identifier, $idDO$ is the data owner identifier, $idTpp$ is the privacy preference identifier, and $d$ are the data of data owner on which $idTpp$ applies.

We emphasize that privacy preference $idTpp$ can be applied to several data tuples $t_d$ by the same users, and that it can contain thousands of data from a single data owner.

The requestor starts a new workflow via the smart contract *SWE-SC*, to which he/she sends the workflow tuple (2, Fig.6.1).

**Definition 9** (Workflow tuple)**.** The workflow tuple is a tuple $t_w = \langle idW, idWR, input \rangle$, where $idW$ is the workflow identifier, $idWR$ is the requestor identifier, $input$ are the input data or parameters to pass to the workflow.

For each task of the workflow, the SWE-SC contacts the target service by sending the instructions necessary to perform the work (3, Fig.6.1). As anticipated, the blockchain-based workflow engine (SWE-SC) has already been treated in other studies (see [22, 71]), so we do not redefine the data structures, as well as the response of the service at conclusion of the task that allows the workflow progress.

In step 4 (Fig.6.1), the target service needs to access data stored in the data source to perform its task, so it sends a request tuple to the PE-SC smart contract.

**Definition 10** (Request tuple)**.** A request tuple is a tuple $t_r = \langle idTr, idS, idW, idT, idDS, p, r \rangle$, where $idTr$ is the request identifier, $idS$ is the service identifier, $idW$ is the workflow identifier, $idT$ is the task identifier, $idDS$ is the data source identifier, $p$ is the privacy policy, and $r$ is a request addressed to data source $idDS$.

When the function $dataRequest()$ (Line 5, Pseudocode 5) receives the tuple $t_r$, it examines $p$ and $idS$. If there is no queue $q$ for the service, it creates a new one and stores the request. If a queue already exists for that service, the function considers whether the new privacy policy has an inclusion relationship with those already present. If no match is found then a new queue is generated (Line 7).

After that, we check if $p.up$ is present in $indexIP$ as an entry point (Line 9). If there is no node that has an equal purpose value, then we create a link node with a value $p.up$. This allows us to insert in the graph only valid IP nodes that we really need, without exploring all the possible combinations, saving space and time. To do this, we invoke the function $insertIP()$ (Line 10). The $insertIP()$ function is used to populate the IP graph, called $indexIP$. It wants as parameters, $ip$ and $indexIP$ (we discuss this function in detail later). In the end, we save on blockchain the updated $indexIP$, $qList$, and $t_r$ with their keys $idQ$, $idS$, and $idTr$ respectively.

The same request is sent directly to the data source concerned, to process it and to know which tuples are involved in the enforcement process.

When the data source has examined the service request $r$, it produces a list of data tuples as a response. Before sending the response to the service, the data source asks the blockchain which of these data tuples can be released. To facilitate the blockchain

[5]We use the key-value notation $(k, v)$ to indicate a value $v$ stored on blockchain with key $k$, which can also be a composite key, i.e. made up of subkeys, e.g. $k = k_1.k_2.k_3$.

and not show sensitive information, it uses the $idTd$, appropriately collected according to the $idTpp$ in force. Each $idTpp$ can be applied to multiple $idTd$ and to optimize its size, a privacy preference key-value map $\overline{idTppTd}$ is used. For each key $idTpp$ exists a value that is a vector of $idTd$ to whom $idTpp$ appplies:

$$\begin{aligned}\overline{idTppTd} = ((&idTpp_1, [idTd_a, idTd_b, ..., idTd_l]),\\ &(idTpp_2, [idTd_c, idTd_d, ..., idTd_m]),\\ &\dots,\\ &(idTpp_q, [idTd_e, idTd_f, ..., idTd_n]))\end{aligned}$$

Then, $\overline{idTppTd}$ is inserted in a reply tuple $t_{rp}$ and sent to PE-SC (5, Fig. 6.1). We point out that the relationship between $idTpp$ and $idTd$ represents the $\alpha$ component of the privacy model, which is used in the enforcement phase and as future audit.

**Definition 11** (Reply tuple)**.** A reply tuple is tuple $t_{rp} = \langle idTrp, idTr, idS, idW, idT, idDS, \overline{idTppTd}\rangle$, where $idTrp$ is the reply identifier, $idTr$ is the request identifier, $idS$ is the service identifier, $idW$ is the workflow identifier, $idT$ is the task identifier, $idDS$ is the data source identifier, and $\overline{idTppTd}$ is the privacy preference map to be checked.

The function $dataReply()$ (Line 17), evaluating $idT$, receives from the blockchain the queue identifier $idQ$ and the $indexTask$. The queue identifier $idQ$ specifies in which queue the task $idT$ has been inserted and therefore which is the $indexIP$ and $indexTPURT$ to consider. We have one index data structure per queue, so multiple tasks share the same index. Now, for each $idTpp$ from $\overline{idTppTd}$, we look for the corresponding $tpp$. Before proceeding, we check if $idS$ (aka *consumer*) of the privacy preference matches that of the request (Line 24). This pre-enforcement check represents the match of the *consumer* component of the privacy model, in order to avoid loading privacy preferences in the index that certainly do not pass enforcement.

Now, as a first indexing step, we take care of $indexIP$, using the function $insertIP()$ (Line 25), where $tpp.ip$ and $indexIP$ are passed as parameters. The creation of the graph takes place starting from the child node and recursively generating the parent nodes. First of all, we call $derivIP()$ (Line 7, Pseudocode 6), which implements the subtree derivation procedure described in the Section 6.3.1. It returns a vector $\overrightarrow{ip}$, consisting of $\overrightarrow{ip}'$, aka *link nodes*, that have a direct inclusion relationship with input $\overrightarrow{ip}$. We point out that in order to explore all possible subtrees, we must recursively invoke the function, as described below.

For each $\overrightarrow{ip}'$ in $\overrightarrow{ip}$ we check if it does not exist in $indexIP$, then we recursively invoke $insertIP()$ (Line 10).

Continuing the execution, we create an empty map $visited$ to report the visited nodes and we indicate $\overrightarrow{ip}'$ as visited with the value $true$ (Line 13, Pseudocode 6). Now, we check if there is any descending node of $\overrightarrow{ip}'$ that can include or can be included to $ip$, using the $insertIPDesc()$ function. It is a recursive function that takes $\overrightarrow{ip}'$, $\overrightarrow{ip}$, $indexIP$, and $visited$ as input parameters. When a descendant $\overrightarrow{ip}''$ encounters an inclusion relation, an arc is added between $\overrightarrow{ip}''$ and $\overrightarrow{ip}$ (Lines 23 and 25, Pseudocode 6), otherwise the exploration continues recursively (Line 27, Pseudocode 6). Returning to $insertIP()$, the last step is add an arc from $\overrightarrow{ip}'$ to $\overrightarrow{ip}$ (Line 15, Pseudocode 6).

Continuing the execution of $dataReply()$, the next call is to $indexTPURT$ (Line 26, Pseudocode 5). The $insertTPURT()$ function (Line 32, Pseudocode 6) takes $\overrightarrow{ip}$, $tpu$,

$rt$, $indexTask$, and $indexTPURT$ as parameters. The index $indexTPURT$ is also defined as a two-dimensional map, on the first dimension we find $\overrightarrow{ip}$ and on the second one $tpu$. The map points to a list that has $taskIDs$ as values. We use a $get()$ function (Line 33, Pseudocode 6) to retrieve the value of the $rt$ key. Knowing $indexTask$, which indicates the position of the bit to be set to 1, we make a shift to the left of the flag. This value is put in $Or$ with the one already present (Line 34, Pseudocode 6) and stored on the blockchain with $set()$ function (Line 35, Pseudocode 6).

Finally, $trp$ and the updated $indexIP$ and $indexTPURT$ are saved on-chain (Line 29-31, Pseudocode 5).

### 6.5.2 Privacy enforcement

The enforcement is handled by $privacyEnforcement()$ and its execution is stimulated by a trigger (Line 33, Pseudocode 5). The trigger could be set on a time basis, e.g. every 500 milliseconds, or on the number of waiting tasks on queue, e.g. every 10 *task* collected, or on-demand. However, it must be sized so that the waiting time for the data source is negligible. The choice is left to the system administrator according to the desired deployment. When the function is called, passing $idS$, it collects the queue list $qList$ for the given $idS$ (Line 34, Pseudocode 5). From $qList$, we selects the $idQ$ to check (Line 35, Pseudocode 5). As in the case of the trigger, the queue management policy is left to the administrator. After that, we fetch $indexIP$ and $indexTPURT$ from the blockchain, and initialize an empty map $\overline{enforcedPP}$ (Line 38, Pseudocode 5) to store the $pp$ that pass the check and can be released to the requesting services.

For each $p$ listed in $qList(idQ)$ we call a recursive function $explorePE()$ (Line 40, Pseudocode 5) to select the $pp$ that meet the requirements of $p$ from the index data structure. Every single queue in $qList$ is sorted in such a way that all $p$ satisfy the inclusion property. The function $explorePE()$ (Line 37, Pseudocode 6) takes as parameters $indexTask$, $\overrightarrow{ip}$ ($\overrightarrow{up}$ in the first call), $tpu$, $rt$, $indexIP$, $indexTPURT$, and $\overline{enforcedPP}$. The $indexIP$ crossing begins by selecting $\overrightarrow{ip}$ as starting node. For each descendant $\overrightarrow{ip}''$ of $\overrightarrow{ip}$, recursively $explorePE()$ is called (Line 39, Pseudocode 6). Browsing the *IP graph*, we could find two types of nodes, the link ones and the real ones. In the case of link nodes we do nothing, we let the recursion find all the real nodes. We realize we have found a real node because $indexTPURT$, in position ($ip$, $tpu$) contains a list (Line 41, Pseudocode 6). At this point we need to examine which $rt$ we can select. The list includes the $gt()$ function which returns all values greater than or equal to a certain $rt$. Therefore, for each value obtained, we generate a label $lblPP$ (Line 43, Pseudocode 6) which represents the $pp$ in absolute terms (e.g. a string "*ip-tpu-rt*"). The map $\overline{enforcedPP}$ uses the label $lblPP$ to store the filtered bit field (Line 44, Pseudocode 6). The filtering operation is obtained by putting in *And* $taskIDs$ with the bit mask *ones* (bit sequence with value 1) shifted of $indexTask$ positions.

For simplicity, the visit flag is not shown in the pseudocode, indeed each visited $rt$, inside the list, is marked with the visited flag and is no longer visited, improving the performance. Similarly when each $rt$ has been used, the entire map entry ($ip$, $tpu$) of $indexTPURT$ is marked as visited and no longer accessible. We repeat the process using the next $p$ in the $qList$ (Line 39, Pseudocode 5). Finally, the smart contract stores, on blockchain, a privacy enforcement tuple $t_{pe}$ (Line 44, Pseudocode 5).

**Definition 12** (Privacy enforcement tuple)**.** A privacy enforcement tuple is $t_{pe} = \langle idTpe, idS, idQ, \overline{enforcedPP} \rangle$, where $idTpe$ is the privacy enforcement tuple identifier, $idS$ is the service identifier, $idQ$ is the queue identifier and $\overline{enforcedPP}$ is the key-value map of privacy enforcement.

The data source, receiving the $t_{pe}$, knows that it can release to the services (7, Fig. 6.1) only the data covered by the privacy preferences present in the map $\overline{enforcedPP}$.

## 6.6 Evaluation

We deploy our platform on dedicated Ubuntu server 18.04.5 LTS, with AMD Ryzen Threadripper 1950X 16-Core processor, 32GiB system memory, and 256GB NVMe disk storage. We use Hyperledger Fabric v2.2 LTS with Docker container to run peers. We define a set of experiments to measure the efficiency of the proposed solutions based on different metrics. In this section, we first introduce the metrics used, then the dataset, and finally the results obtained.

### 6.6.1 Metrics

We have considered three main metrics, discussed in the following.

#### Complexity

We define the complexity of privacy preferences based on the operations required to store it in the index. By examining the components of a privacy preference, we realize that the greater the number (intended as quantity) of purposes in $ip$, a greater number of nodes are created in the IP graph. For $tpu$, choosing $true$ or $false$ is irrelevant to the computation. A large $rt$ means longer list traversal times. Thus, we can define three complexity classes:

- Low: $1 \leq purp[\#] \leq 6 \wedge 1 \leq rt[d] \leq 1200$;
- Mid: $7 \leq purp[\#] \leq 12 \wedge 1201 \leq rt[d] \leq 2400$;
- High: $13 \leq purp[\#] \leq 18 \wedge 2401 \leq rt[d] \leq 3600$.

The complexity of the privacy policy is determined by the operations necessary to carry out privacy enforcement through the proposed algorithm. Assuming an index with privacy preference uniformly distributed between low, mid and high complexity, we note that: the lower the number (intended as quantity) of purposes in $up$, the more nodes analyzed in the IP graph; the lower $dataRet$ the higher only the values to be selected in the list; for $dataRel$ the choice is irrelevant. Also in this case we define three complexity classes:

- Low: $13 \leq purp[\#] \leq 18 \wedge 2401 \leq dataRet[d] \leq 3600$;
- Mid: $7 \leq purp[\#] \leq 12 \wedge 1201 \leq dataRet[d] \leq 2400$;
- High: $1 \leq purp[\#] \leq 6 \wedge 1 \leq dataRet[d] \leq 1200$.

#### Coverage

The data source, when formulating the reply to a data request, sends a number of data tuples with the indication of the privacy preference associated for each tuple. A privacy preference can cover multiple data tuples, because the same privacy preference can be associated with different data tuples. Coverage measures the relationship between the amount of privacy preference and the total number of data tuples provided by the data source, in percent. We assume that the coverage is uniformly distributed, for example a coverage of 5% means that a privacy preference covers an average of 20 data tuples out of 100.

#### Selectivity

Selectivity is the ratio between the privacy preferences that pass the compliance check and the total privacy preferences. For example, a selectivity of 75% indicates that 75/100 privacy enforcement are successful, i.e. the privacy policy is compliant with 75/100 privacy preference, while the remaining 25/100 is not.

### 6.6.2 Dataset

For our tests we used a synthesized dataset, since none of the existing ones fits our purpose and our implementation. We could not use datasets such as OPP-115 Corpus [94] or PolicyQA [2] because they are a collection of website privacy policies and therefore incomplete on the privacy preference part.

The number of tasks is an essential parameter to be evaluated. A task is equivalent to a certain number of privacy enforcement. This quantity is defined by the coverage on the number of data tuples of the task. For example, if a task has 100000 data tuples with coverage = 1%, it means that there are 1000 privacy preferences. Our system supports multi-task scenarios, that is the privacy enforcement of different tasks within the single execution of the smart contract. In this case, we set the same coverage for each task, so as to have results consistent with the same amount of privacy enforcement.

The other two parameters we vary are the complexity of the privacy policy and the selectivity. The privacy preferences that populate the datasets have complexity uniformly distributed between *low*, *mid* and *high*.

To evaluate the platform under every possible situation, datasets were defined under the following conditions:

- Privacy policy complexity: Low, Mid, High.
- Coverage: 1%, 5%, 10%.
- Selectivity: 50%, 75%, 100%.
- Task: 1-10.

In total, 270 datasets were generated, each repeated 10 times, for a total of 2700 tests.

By sizing the data reply to 100000 data tuples, as the coverage varies we get 1000 (1%), 5000 (5%), 10000 (10%) privacy preference per task. In the case of coverage at 10% and 10 tasks, we reached 100000 privacy enforcement in a single execution of the smart contract.

The same tests were repeated with the naive implementation to make the comparison, reported in Pseudocode 7. In the *dataReplyNaive*() function we do not create the index but we simply add the privacy preference to a list *listPP* (see,

**Pseudocode 7:** Privacy enforcement smart contract Naive (PE-SC)

```
1  Let:
2  tpp be the privacy preference tuple;
3  idTpp be the privacy preference tuple identifier;
4  σ be the tuple selection ;
5  Function dataReplyNaive (trp)
6      idQ = getBcIdQ(trp.idT);
7      indexTask = getBcIndexTask(trp.idT);
8      new listPP;
9      forall idTpp in trp.idTppTd do
10         tpp = getBcPp(idTpp);
11         listPP.add(tpp);
12     end
13     putBcListPP(idQ-indexTask,listPP);
14     putBcTrp(trp.idTrp,trp);
15 end
16 Function privacyEnforcementNaive (idS)
17     qList = getBcQList(idS);
18     idQ = selectQ(qList);
19     enforcedPP = {};
20     forall indexTask, p in qList(idQ) do
21         listPP = getBcListPP(idQ-indexTask);
22         forall pp in listPP do
23             Let ipFlag be a boolean variable, initialized as False;
24             Let rtFlag be a boolean variable, initialized as False;
25             Let tpuFlag be a boolean variable, initialized as False;
26             if (p.up ∈ pp.ip→) then
27                 ipFlag=True;
28             end
29             if (p.dataRet ≤ pp.rt) then
30                 rtFlag=True;
31             end
32             if (p.dataRel = pp.tpu) then
33                 tpuFlag=True;
34             end
35             if (ipFlag=True & rtFlag=True & tpuFlag=True) then
36                 lblPP=genLabelPP(pp.ip,pp.tpu,pp.rt);
37                 enforcedPP[indexTask][lblPP]= True;
38             end
39         end
40     end
41     idTpe = new id;
42     tpe=(idTpe,idS,idQ,enforcedPP);
43     putBcPE(idTpe,tpe);
44 end
```

Line 11). Then we keep $listPP$ on the blockchain with a composite key, made up of $idQ$ and $indexTask$ (see, Line 13), because we can not aggregate privacy preferences of different tasks. In this way we have as many lists as there are tasks. The $privacyEnforcementNaive()$ function uses $listPP$ instead of indexes in privacy enforcement. At each $taskIndex$, we receive from the blockchain $listPP$ which corresponds to the composite key $idQ$-$indexTask$ (see, Line 21). For each $pp$ in $listPP$, we perform the compliance check against the service's $p$. The check is successful if the $p.up$ are contained in $pp.\overrightarrow{ip}$, if $p.dataRet$ is less than or equal to $pp.rt$ and the same data release flag is present. When $pp$ and $p$ are compliant, $pp$ is added to the multidimensional list of privacy preferences enforced $enforcedPP$ (see, Line 37. We

use *indexTask* on the first dimension to differentiate the membership of the task, the *lblPP* label on the second for easy search and the *True* value as a sign of compliance.

### 6.6.3 Results

All combination of tests were carried out, producing a large amount of results. To present them, we start by setting the coverage at 5% and the complexity at Mid. This is an average case, reflecting a real system load, because it does not allow the system to effectively group privacy preferences nor to achieve maximum grouping.

#### Privacy enforcement execution time

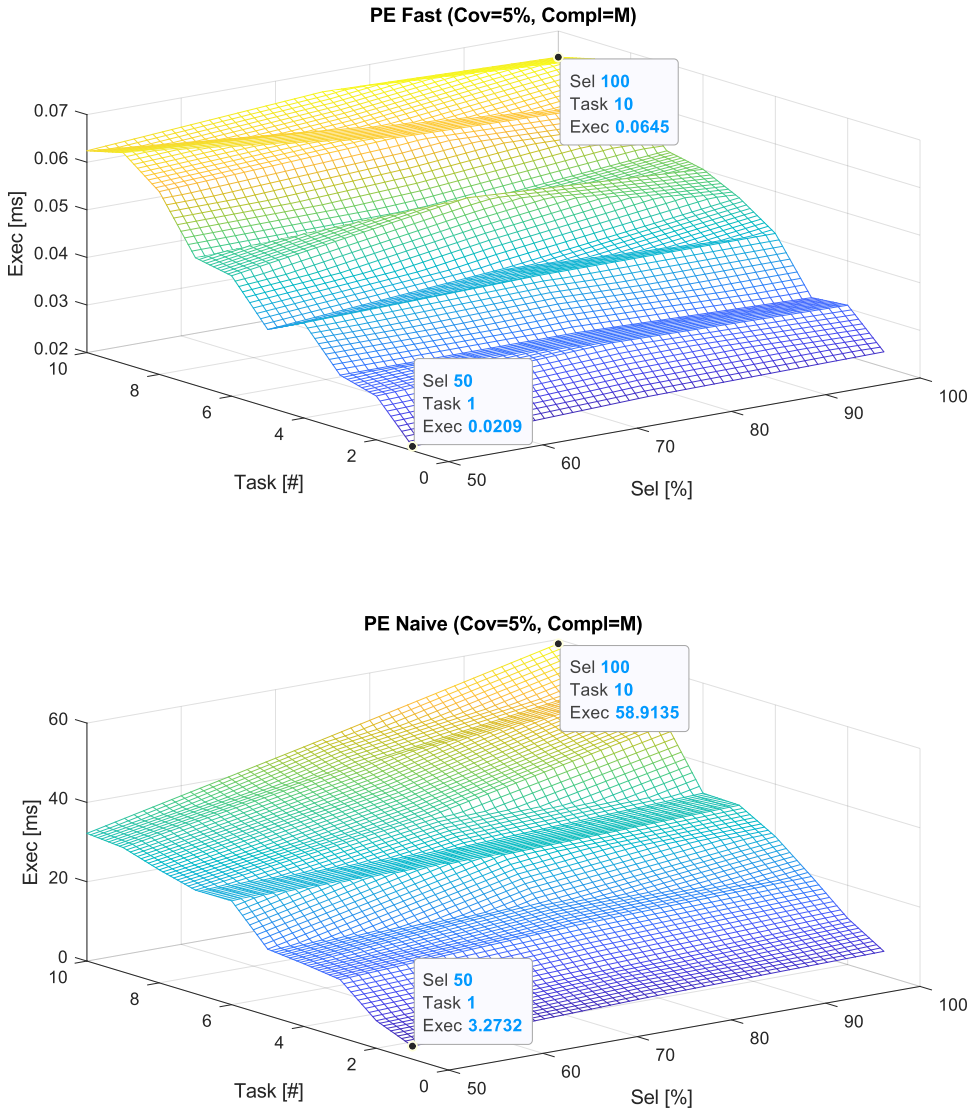



FIGURE 6.4: Privacy enforcement execution time in fast and naive approach.

The privacy enforcement execution time is the net time that the $privacyEnforcementFast()$ function (Pseudocode 5) takes to process all the privacy enforcement of all the tasks in the queue $idQ$. This measure does not take into account the times needed by the blockchain to reach the distributed consensus. Figure 6.4 shows the trend, setting complexity and coverage, and varying the number of tasks and selectivity. At maximum load, in the fast implementation, we get the same result as in the naive case using 0.0645 ms, instead of 58.9135 ms. We see that the trend is increasing as the number of tasks and selectivity increase.

### Transaction throughput

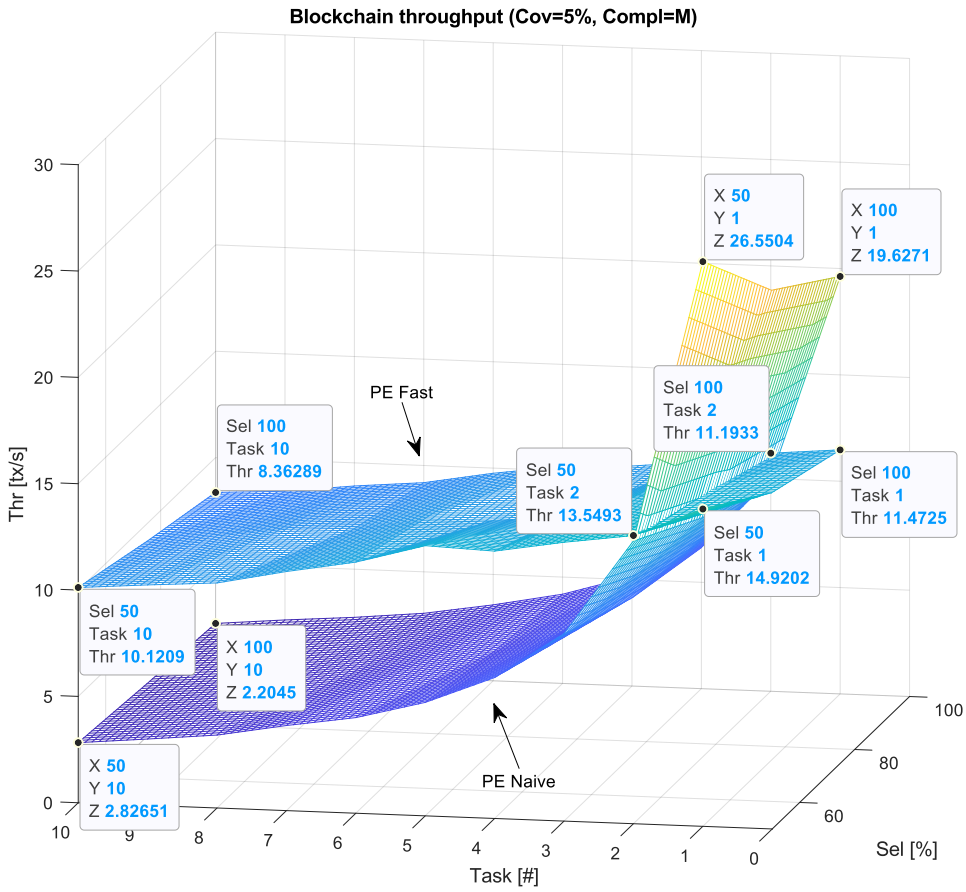



FIGURE 6.5: Blockchain throughput measured in transaction per second.

The transaction throughput is the number of blockchain transactions performed in one second. Each transaction includes the call time of the smart contract, the execution of the function $privacyEnforcementFast()$ function (Pseudocode 5), the consent process, the writing of the transaction on the blockchain and the communication times between peers. The deployment of the blockchain was carried out in single host, so the communication times correspond to those of the Docker network within the same machine. The figure 6.5 shows the throughput trend for the fast and naive cases. The naive algorithm proves to be more efficient on a single task. Its trend decreases quickly when dealing with multiple tasks together, reaching 19-26 tx/s. The

fast algorithm, designed to perform privacy enforcement of aggregated tasks, maintains a linear trend describing as the number of tasks and selectivity increase, until 8-10 tx/s. It shows a higher number of transactions per second, compared to the naive approach, for values greater than 2 tasks. In both cases, the trend is decreasing to increasing selectivity, because the number of privacy preferences that pass control increase, up to 100%.

### Privacy preference throughput

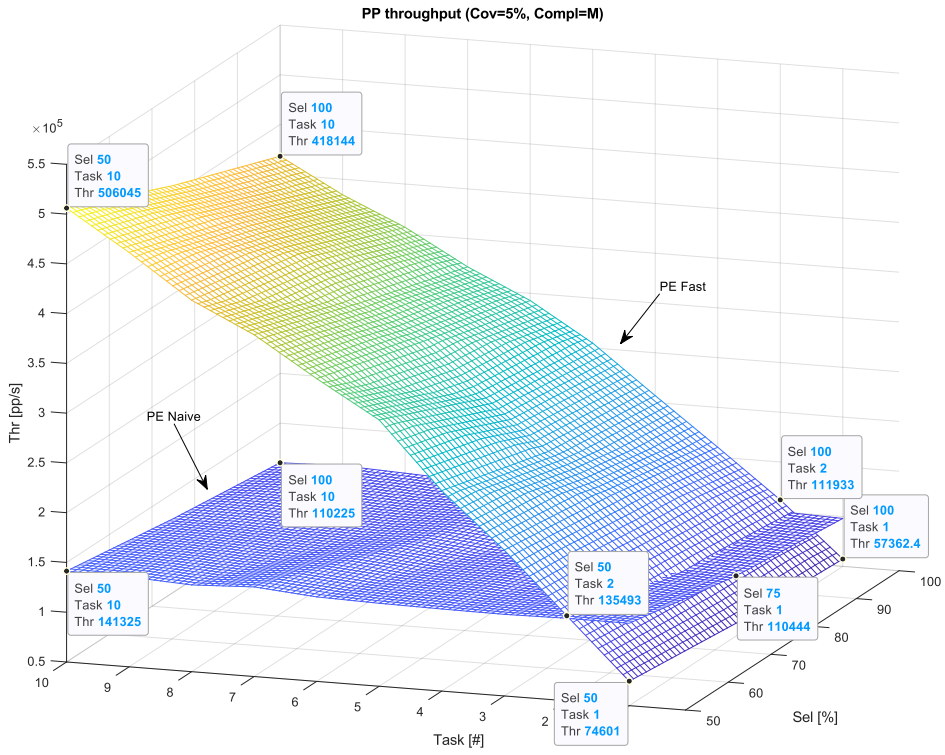



FIGURE 6.6: Elaborated privacy preference throughput.

The privacy preference throughput shows the number of privacy preferences processed per transaction in one second. It relates the transaction throughput to the coverage and the number of tasks. In Figure 6.6, for the fast case, we obtain higher throughput values then naive, starting from the 2 tasks. We reach peaks of 500000 privacy preference per second, thanks to the aggregation approach of this method that controls many privacy preference at the same time. In the naive case, the trend is around 110000-141000 privacy preferences per second, it shows the limitations of this implementation. The same decreasing trend with increasing selectivity is noted, due to the lowest number of privacy preference not compliant.

### Privacy enforcement overall execution time

To avoid showing all the graphs, for each combination of coverage and complexity, we summarize the privacy enforcement execution time in a single graph. Each point in Figure 6.7 represents the average time for all tasks (from 1 to 10) and for selectivity (50%, 75%, 100%) with the same coverage and complexity. Basically, the value of

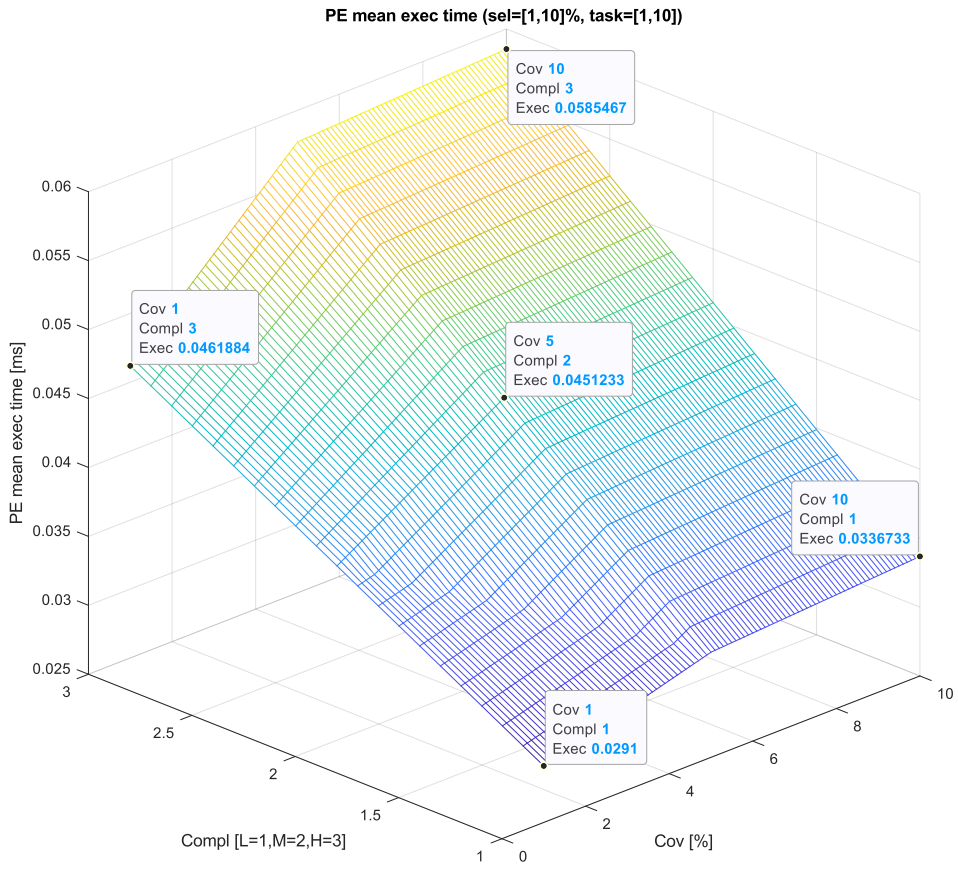



FIGURE 6.7: Privacy enforcement mean execution time for coverage from 1% to 10% and complexity from *L* to *H*.

the central point with coverage at 5% and Mid complexity (M = 2), represents the average of all times of the graph Figure 6.4. The value of the average times is to be considered in relative value with respect to the times represented in the graph. It does not give us absolute information on the execution time, but shows us how the trend of the system evolves with increasing complexity and coverage. In Figure 6.4, we had seen how the trend was increasing as the number of tasks and selectivity increased. In Figure 6.7, we see how the system load also increases with increasing complexity and coverage.

### 6.6.4 Indexing costs (insert, update, space overhead)

Indexing privacy preferences has a cost that we can analyze in terms of time and space overhead in inserting and reading operations. We do not consider the cost of deleting or modifying an element in the index, because it is developed with the append-only principle in order to adapt to the blockchain.

Figure 6.8 shows the percentage time overhead as a ratio between indexing and listing. The listing is the operation of reading the privacy preferences from the blockchain that must be done in both the naive and the fast cases. In the naive

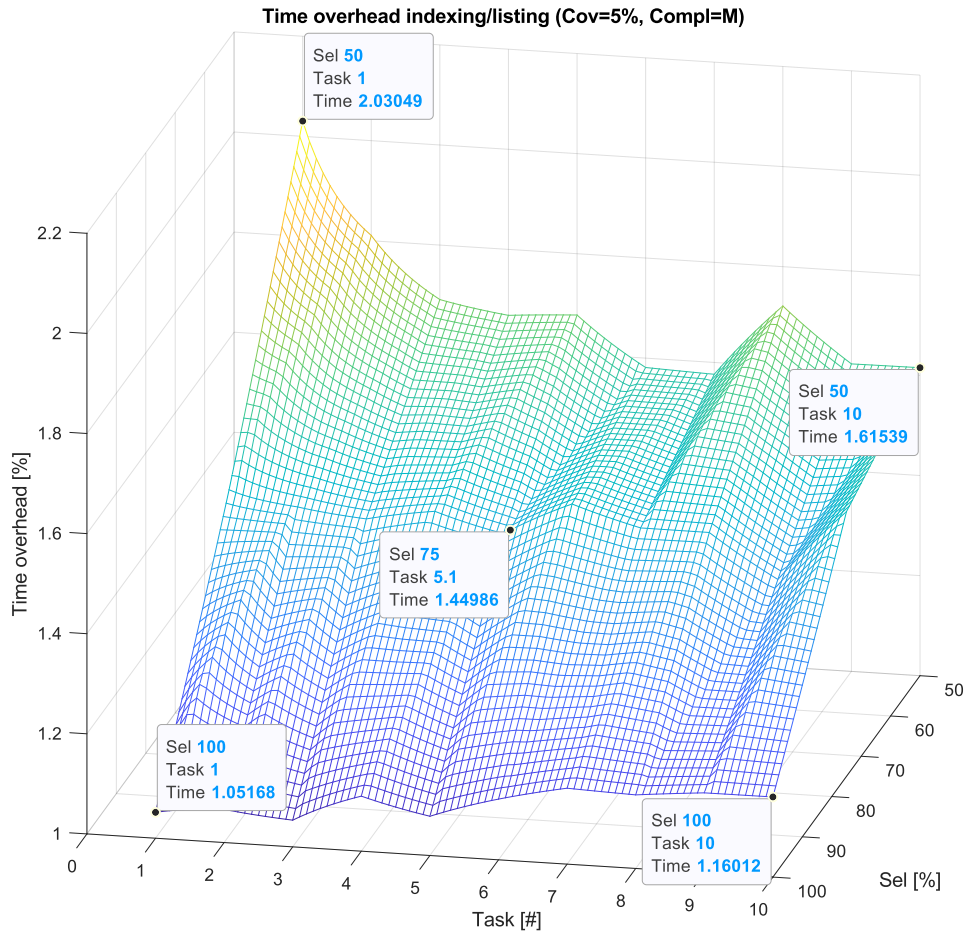
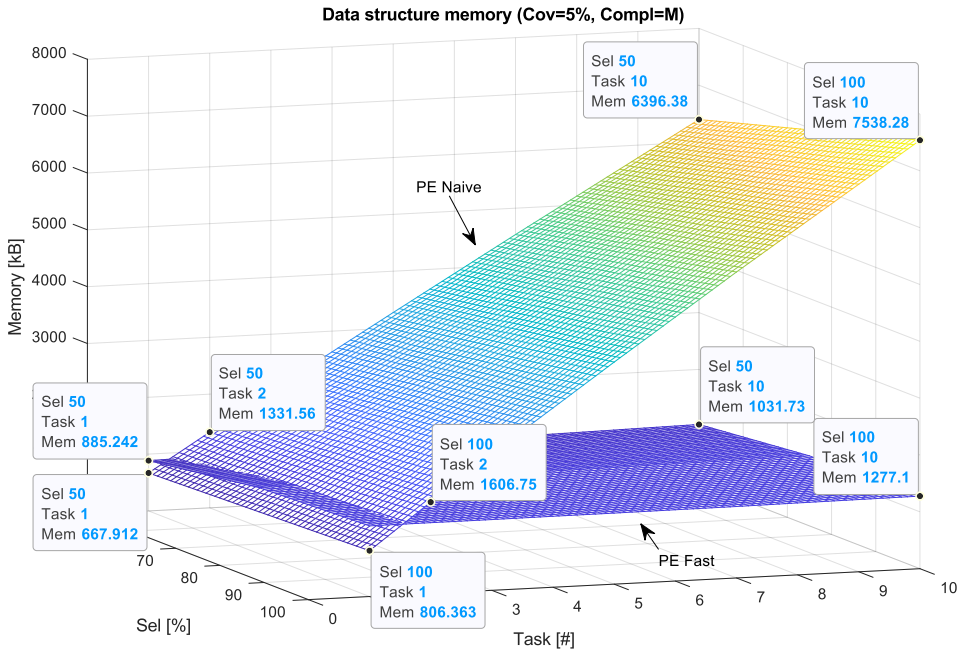



FIGURE 6.8: Insert time overhead as ratio between indexing (PE fast) and listing (PE naive).

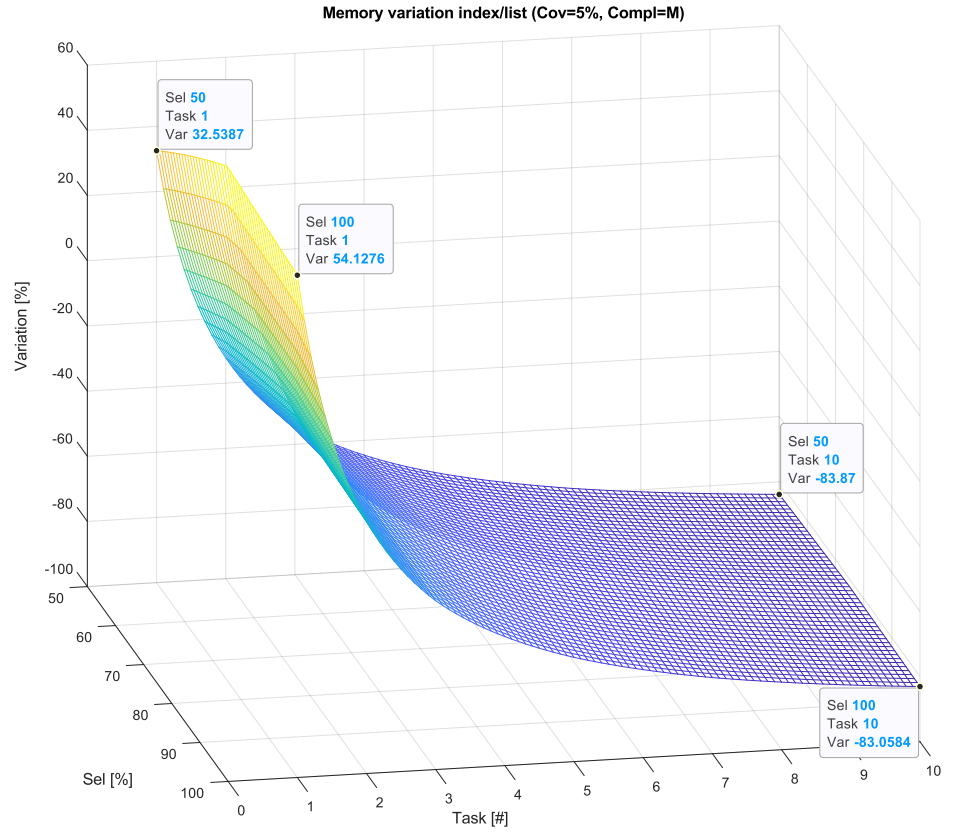
case, the list is then used directly to perform privacy enforcement by element. Instead, in the fast case the list is indexed. Starting from the privacy preference list, read from the blockchain, the time overhead indicates how much more time is spent for indexing. We can see, in the medium case (coverage at 5% and Mid complexity), how the maximum time overhead, of about 2%, for the creation of the index is obtained with the number of tasks at 1 and the selectivity at 50%. Here, we get a high overhead value because the index creation for 1 task is averaged over a smaller number of privacy preferences. By increasing the number of tasks, therefore the number of privacy preferences, the average indexing time is reduced because there are overlapping privacy preferences. When we insert a privacy preference that has elements in common with a previously indexed one, for the new one we consider the elements that differ (purposes, retention time, third party usage). Similarly, the low selectivity indicates that many privacy preferences do not pass the compliance check. Not passing the control means that they differ a lot from the privacy policy, so it means treating privacy preferences that have few elements in common with each other, with a low probability of aggregation. Considering that the worst case, in the average scenario, it is still a very good time compared to the listing. Reading from the blockchain is one of the most time-consuming and bottleneck operations.

In any case, the time consumed for indexing, even if done in moments prior to privacy enforcement, is significantly rewarded by the excellent performance of the fast algorithm. In fact, using indexing allows us to operate with a much smaller data structure in terms of memory and manageable. We are able to overcome the bottleneck that is highlighted in the number of privacy preferences processed per second (see, Figure 6.6) and significantly scale.

The Figures 6.9 analyze the cost of storing the index against the list. In the figure 6.9a, for coverage at 5% and Mid complexity, the cost of the list in terms of space is linear as the number of tasks, and therefore of tuples, increases. In this case the variation on the selectivity does not affect. The trend of the index, on the other hand, has a less pronounced slope than the list, due to the ability to aggregate on similar privacy preferences. We note that for 1 task, the index has higher values than list, but we remember that the goal of the platform is the multi-task privacy enforcement. In the figure 6.9b, the cost of memory is shown in percentage values. The percentage variation, of the index on the list, gives us positive values only for tasks at 1, then it decreases rapidly from the 2 tasks, to reach almost stationary levels from the 6 tasks. For these task values, memory savings is about 80%.




(A) Data structure memory cost.




(B) Memory variation cost as ration between index and list.

FIGURE 6.9: Memory cost of index (PE fast) and list (PE naive).

# Chapter 7

# Conclusion

Blockchain technology is now recognized for having the potential to revolutionize the economy and business models. In addition to the financial sector, it can be used in a wide range of applications, including collaborative workflow and privacy enforcement, thanks to its transparency, accountability, consensus-based verification, smart contract execution and so on. Hence, blockchain can help overcome many challenges in these sectors.

In this thesis, we focused on solving some of them, we ensured a *correct and safe execution of collaborative processes with off-chain resource sharing*, we ensured that the *privacy preferences of data owners were considered by consumers*, and we ensured that *this platform adapts to different contexts, such as IoT or scientific workflows*.

We have significantly based our platform on blockchain technology, in particular for the trust feature, extending its applicability and favoring its introduction into new contexts.

In the following, first we give a short summary of the work that has been presented in this thesis, and finally we discuss the future work.

## 7.1 Summary

We present a summary of the work that has been done in this thesis chapter by chapter.

- In chapter 4, we presented a blockchain-based framework to ensure a trusted workflow execution and access control policy enforcement. As collaboration among heterogeneous organizations may involve the exchange of sensitive data, we proposed to use a smart contract to implement the workflow coordination and the access control mechanism. Also, we rely on the blockchain validation of smart contract execution to ensure the correctness of the enforcement. Moreover, we proposed to enforce the access to these resources through the implementation of access control policies that are deployed on the blockchain as smart contracts. Experimental results highlight the implied costs and the feasibility of the proposed approach.

- In chapter 5, we presented a blockchain-based framework that allows the data owner to express their wishes about data processing (privacy preferences) in the IoT context. We designed the platform using smart contracts that execute privacy enforcement in a distributed way, eliminating the need for a centralized entity. The validation of the blockchain network of the execution of the smart contract, in fact, guarantees the correctness of the processes. The release of data, from the data owner to the consumer, takes place off-chain but always under the supervision of the blockchain, enabling the audit function.

The experimental results highlight the implicit costs and the feasibility of the proposed approach.

- In chapter 6, we presented a privacy-preserving blockchain-based scientific workflow platform. In this chapter, we jointed the two platforms of previous chapters, enabling the workflow execution with the privacy enforcement feature. We have provided a method to perform privacy enforcement in an efficient and scalable way, adapting it to the blockchain context. To do this we have acted on three main guidelines, the definition of a method to aggregate privacy preferences and policies, the construction of an index for rapid access to data in an environment with high read/write latencies, and the reduction of the computational cost for the execution of privacy enforcement, that is the reduction in the overall number of checks and the speeding up of the individual checks. The results obtained are much better than the naive implementation, and show the feasibility of our approach.

## 7.2 Future work

We plan to extend the thesis work along several dimensions. First, we plan to run a more extensive set of experiments, for different scenarios. Then, we plan to consider other security properties (e.g., data provenance). We also plan to deploy our solution on other blockchains in addition to Hyperledger Fabric. Another relevant extension is the support of a dynamic workflow deployment. As an example, a workflow where task executors are determined at run time, based on given criteria (e.g., QoS criteria). On the IoT context, we plan to extend this work along many directions. We want to enable the blockchain to perform privacy-preserving complex event processing, so as to move the processing of the tuples from the consumer to the blockchain. In general, we will also expand the privacy model in order to manage extended privacy preferences on derived data. We also plan to define optimization strategies for an off-chain self-designed data release layer. Then, we plan to run a more extensive set of experiments, (e.g., standard benchmarks).

# Appendix A

# Blockchain platforms

*In this chapter, we will discus in details about Ethereum and Hyperledger. We will take into account history, components, the protocols and finally, their relationship with Smart Contracts.*

## A.1 Ethereum

Ethereum [95] is a blockchain-type computer technology, which means that it is a decentralized platform, which allows you to run programs called "smart contracts". That is to say that it carries out in a completely digital form and in a safe, reliable, fast, automated and economic way the completion and management of the most various types of contract and many other operations. Ethereum contracts "pay" for the use of its computational power through a unit of account, called Ether, which acts as a cryptocurrency but also serves to finance the platform. Therefore, Ethereum is not to be considered therefore only a simple network for the exchange of monetary value (such as bitcoin), but a network that allows the circulation of contracts.

### A.1.1 Hystory

In November 2012, a small group of people had gathered at the Pauper's Pub in Toronto, Canada. The meeting was organized by the well-known businessman Anthony Di Iorio, because he was attracted to the world of cryptocurrencies and the fact that he was worried about how things were going in the financial system. So he sold his properties owned in Toronto, buying some bitcoins with the hope of doing excellent business, he also wanted to involve other people in his project. Di Iorio thus organized an ad hoc event on the Meetup.com site to talk about bitcoin.

Only a few people, unknown among them, signed up. Among them was a young computer science student enrolled in the first year at the University of Waterloo, located in southwestern Ontario. His name is Vitalik Buterin.

Di Iorio claims not to have talked much with him and that there was no real conversation between them. They then got to know each other better at subsequent meetings. Vitalik Buterin in 2011, had created Bitcoin Magazine with a Romanian friend and was already well known in the world of cryptography. A few months after that meeting in Toronto, Buterin left university and devoted himself to traveling around the world writing for the full-time magazine. The following year since that first meeting, Di Iorio was still organizing meetings, although in larger spaces as the stalls had gradually increased.

Vitalik Buterin had created something sensational: he had made it possible, by discovering what had already been made by bitcoin, without the mediation of credit institutions or the control of governments and central banks, by sending money to different countries and in different currencies. And so he thought, for example, of

the fact that an Iowa farmer could monetize an insurance contract, which would allow him to obtain a certain amount in the event that, in a given season, the rain had not fallen.

Or, a rental car could connect with the driver's smartphone and start the ignition via an application. Or again, people could earn money by renting their hard drives for a decentralized cloud storage service like Dropbox.

To make these contracts come true, Buterin creates a new system based on a blockchain called Ethereum, simple to understand and facilitated by a coding language capable of theoretically solving any computational problem.

Ethereum meant that the trust of a financial company, person or government could not be relied upon to keep certain amounts of money and data safe.

### A.1.2 Composition

Ethereum is made up of several components. In the following sections present the details.

**EVM.** The Ethereum Virtual Machine, often abbreviated through the acronym EVM, is the computer center that allows the execution of complex codes (Smart Contracts) above the Ethereum platform. It is therefore a virtual machine, i.e. a software capable of emulating a physical machine in all respects, through a virtualization process in which physical resources are assigned (e.g., CPU, RAM, hard disk, etc) to applications that run above the virtual machine (including its operating system). We therefore understand how it plays a role similar to what JVM plays for Java, that is, a safe environment, isolated and protected from the rest of the running processes or files on the host computer.

A crash or malfunction of the EVM does not cause side effects in the file system of the host computer and in the processes of the operating system. The EVM also act as guarantor for the network nodes, which "offer" their own physical infrastructure for the storage and processing of potentially harmful smart contracts. The open and permissionless nature of the Ethereum blockchain (like that of Bitcoin), in fact, allows anyone to be part of it and to deploy potentially malicious smart contracts. [18]

**Accounts.** Accounts are one of the main elements that make up the Ethereum blockchain. The status is created or updated as a result of the interaction between the accounts. Operations performed between accounts, state transitions performed. State transition is used using the so-called "Ethereum state transition function", which can be defined as follows:

1. Verify the validity of the transaction by checking the syntax, the validity of the signature and that the nonce corresponds to that of the sender's account. Otherwise, an error occurs.

2. Calculates the transaction fee and determines the control of the sender from the signature. In addition, the sender's account balance is checked and subtracted accordingly and the nonce is increased. An error is returned if the account balance is insufficient.

3. Provides enough ether (gas price) to cover the cost of the transaction. This is charged per byte incrementally, based on the size of the contract.

4. Verifies the actual transfer of the value from the sender's account to the recipient's account. The account is created automatically if the target account specified in the transaction does not yet exist. Also, if the target account is a contract, the contract code is executed. This also depends on the amount of gas available. If sufficient gas is available, the contract code will be executed in full, otherwise, it will be executed to the point where the gas ends.

5. In the event of a transaction failure due to the account balance or insufficient gas, all status changes are canceled with the exception of the commission payment, which are added to the miners' account.

6. Finally, the rest (if any) of the commission is respected by the sender. At this point, the function returns the resulting state.

In Ethereum can be defined two types of account:

- Accounts owned externally (EOA -Externally Owned Account), controlled by a public-private key pair and therefore referred to a person / entity.
- Contract Account (CA -Contract Account), controlled by the code present in the contract, also have an associated code.

Account status is made up of four fields:

- **Nonce.** If the account is external this number represents the number of transactions sent by the account address. If the account is a contract, the nonce value is equal to the number of contracts created by the account.
- **Balance.** This value represents the number of Wei which is the smallest unit of the currency (Ether) in Ethereum held by the address.
- **StorageRoot.** This field represents the root node of a Merkle Patricia tree. This tree encodes the hash of the content stored in this account and is empty by default.
- **CodeHash.** This is an immutable field, containing the hash of the smart contract code associated with the account. In the case of normal accounts, this field contains the 256-bit Keccak hash (cryptographic function) of an empty string. This code is invoked through a message call.

**Blocks.** As previously discussed, blocks are the main elements of a blockchain. Ethereum blocks are made up of various components (see Figure A.1), which are described as follows:

- The block header.
- The list of transactions.
- The list of Ommers or Uncles headers.

The transaction list is simply a list of all transactions included in the block, which also includes the list of Uncles headers. The most important and complex part is the block header.

- **Block header.** Block headers represent the most critical and detailed components of an Ethereum block.

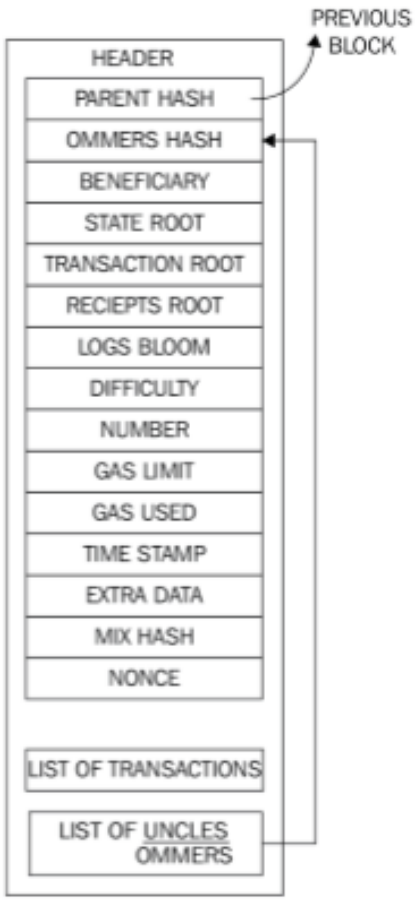



FIGURE A.1: Block structure.

- **Blook header.** Indicates the Keccak 256-bit hash of the parent block header (previous).
- **Ommers hash.** Indicates the Keccak 256-bit hash from the list of Ommers (Uncles) blocks included in the block.
- **Beneficiary address.** Indicates the 160-bit address of the recipient who will receive the mining reward once the block has been successfully extracted.
- **State Root.** Contains the Keccak 256-bit hash of the root node of the trie state. It is calculated after all transactions have been processed and finalized.
- **Root transactions.** Indicates the Keccak 256-bit hash of the root node of the trie transaction. The trie transaction represents the list of transactions included in the block.
- **Receipts root.** Indicates the hash of the receipt array for a given block.
- **Logs bloom.** Bloom filters are compact data structures for probabilistic representation, in order to support membership queries.
- **Difficulty.** Indicates the difficulty level of the current block.
- **Number.** Indicates the total number of all previous blocks (the genesis block is the zero block).
- **Gas limit.** Indicates the value that represents the limit set on the gas consumption per block.
- **Gas used.** Indicates the total gas consumed by the transactions included in the block.
- **Timestamp.** The timestamp or timestamp prevents the operation, once performed, from being altered or canceled, consists of a specific sequence of characters that uniquely, indelibly and unchangeably identify a date and / or time to set and ascertain the actual event of a certain event.
- **Extra data.** It is used to store arbitrary block data.
- **Mixhash.** Contains a 256-bit hash which, when combined with the nonce, is used to demonstrate that adequate computational effort has been spent to create a particular block.
- **Nonce.** Indicates a 64-bit hash (a number) which is used to demonstrate, in combination with the mixhash field, that adequate computational effort has been spent to create this block.

**Transactions.** A transaction is a cryptographically signed data package that stores a message, which tells the acrshort EVM to transfer ether, create a new contract, activate an existing one, or perform some calculations. Contract addresses can be recipients of transactions, just like users with external accounts. Transactions can be divided into two types based on the output they produce:

- Message call transactions, which simply generate a message call that is used to pass messages from one account to another.

- Contract creation transactions, which lead to the creation of a new contract. This means that when this transaction is successful, it creates an account with the associated code.

Both of these transactions are composed of a number of common fields:

- **Nonce:** Represents a number that is incremented by one each time a transaction is sent by the sender. It is used as a unique identifier for the transaction. A nonce value can only be used once.
- **gasPrice:** Represents the number of Wei required to perform a transaction.
- **gasLimit:** Contains the value that represents the maximum amount of gas that can be consumed to perform the transaction.
- **To:** Represents the address of the recipient of the transaction.
- **Value:** Represents the total number of Wei to be transferred to the recipient.
- **Signature:** The signature is made up of three fields, namely v, r and s. These values represent the digital signature (R, S) and some information that can be used to retrieve the public key (V). V is a single byte value that represents the size and sign of the point of the elliptical curve and can be 27 or 28. R is derived from a point calculated on the curve, while S is calculated by multiplying R with the private key and adding it to the hash of the message to be signed and finally divided by the random number chosen to calculate R.
- **Init:** Represents a byte array that specifies the EVM code to use in the account initialization process. The code in this field only runs once, when the account is first created and immediately destroyed.
- **Data:** If the transaction is a message call, the data field is used instead of init, which represents the input data.

**Messages.** A message is a data packet sent between two accounts. This data packet contains the data and a value indicating the amount of ether. It can be sent via a smart contract (autonomous object) or by an external actor (external property account) in the form of a transaction digitally signed by the sender. Contracts can send messages to other contracts. Messages exist only in the execution environment and are never stored. They are similar to transactions; however, the main difference is that they are produced by contracts, while transactions are produced by external entities (external property accounts) to the Ethereum environment. A message is composed of:

- Sender.
- Recipient.
- Quantity of Wei to be transferred.
- Optional data field.
- Maximum quantity of gas that can be consumed.

**Call.** A call is a local request to a contract function and is performed locally on the node. It does not consume gas as it is a read-only operation, moreover they do not involve any change of state.

**Ether.** Ether or ether is a necessary element for the operation of the Ethereum distributed application platform. It is a form of payment made by customers of the platform, to the machines that perform the requested operations. Ether represents an incentive to ensure that developers write applications of a certain quality, as expensive code costs more, and that the network remains healthy (people are compensated for the resources provided). Metaphorically speaking it can be considered as gasoline for all applications in Ethereum.

**Gas.** Gas represents a value that must be "paid" for every operation performed on the Ethereum blockchain. This is a mechanism that ensures that infinite loops cannot cause the entire blockchain to stop. A transaction fee is charged as a certain amount of Ether and is withdrawn from the balance of the account of who created the transaction. There must also be a fee to be included in the transactions intended for miners. If this fee is too low, the transaction cannot be withdrawn; the higher the fee, the more likely the transactions are collected by miners to include them in the block. Conversely, if the transaction that has an appropriate fee is included in the block by the miners but has too many complex operations to perform, an out-of-gas exception is thrown because the cost of gas is not enough. In this case, the transaction will fail but will still be made part of the block and the person who created the transaction will not receive a refund. The transaction cost can be estimated using the following formula:

$$Totalcost = gasUsed * gasPrice \tag{A.1}$$

GasUsed is the total gas that is supposed to be used by the transaction during execution and gasPrice is specified by the creator of the transaction as an incentive for the miners to include the transaction in the next block.

If you run out of gas before execution is complete, any operations performed by the transaction up to that point are restored. If the transaction is successful, the rest of the gas is reimbursed to the sender of the transaction. This concept should not be confused with the "mining" tax, which represents a different concept.

### A.1.3 Ethereum Consensus mechanisms

Ethereum's consensus mechanism is based on the GHOST protocol originally proposed by Zohar and Sompolinsky in December 2013. The motivation behind the creation of GHOST is that blockchains with fast confirmation times suffer, at present, from reduced security due to a high latency rate, because the blocks take a certain time to propagate through the network. If miner A processes a block, and subsequently miner B processes another block before the block of miner A is propagated to B, the block of miner B will be wasted and will not contribute to network security.

As described by Sompolinsky and Zohar, GHOST solves the first problem of loss of network security by including latent blocks in the calculation of which "chain" is the "longest". So, not only the previous block, or those still older than a block, but also those latent and descendants from an older block (in Ethereum called Uncle) are added in the calculation of which block has the largest proof of work that supports it.

Ethereum uses a simpler version of this protocol and is defined as follows:

- A block must specify a parent and must specify 0 or more uncle.
- An uncle included in block B must have the following properties:

  - Must be a direct child of B's generation antagonist of B, where $2 \leq k \leq 7$.
  - Cannot be an ancestor of B.
  - An uncle must be a valid block header, but it does not necessarily have to be a previously verified or even valid block.
  - An uncle must be different from all the uncles included in the previous blocks and all the other uncles included in the same block.

- For each uncle U in block B, the miner of B receives an additional 3.125 % added to the prize of his coinbase and the miner of U receives 93.75 % as the standard prize of the coinbase.

This limited version of GHOST, with uncles that can be included for up to 7 generations, has been used for two reasons. The first, an unlimited GHOST could include too many complications in the calculation of which uncle, for a certain block, are valid. The second, an unlimited GHOST eliminates the incentive for a miner to perform operations on a "chain" of an attacker, but, on the contrary, to carry them out on the main "chain".

### A.1.4 Mining

Mining is the process by which new currency is added to the blockchain. This is an incentive for miners to validate and verify transaction blocks. Theoretically, a miner performs the following functions:

- Listen to the transactions transmitted on the Ethereum network and determine the transactions to be processed.
- Determines obsolete blocks called Uncle or Ommers and includes them in the block.
- Update your account balance with the reward successfully obtained from block mining.
- Finally, a valid state is calculated, which defines the result of all transitions.

Miners are investors who devote time, computer resources and energy to blocking. When the mining process is successful, the miners will present their solutions to the currency issuer. After the verification of the work, i.e. proof of work system, the issuer of the currency will offer rewards in the form of a part of the transactions that helped to verify.

**Ethash.** The proof-of-work algorithm in Ethereum is called by the name of Ethash. Originally, it was proposed as a Dagger-Hashimoto algorithm or a memory-hard algorithm resistant to brute-force, but over time the Proof of Work algorithm has evolved a lot, transforming itself into what is now known, in fact , like Ethash. It represents a new function created to solve the problem of centralizing the extraction in Bitcoin. This algorithm mixes two standard cryptographic functions, namely the SHA -3 and the Keccak. This results in a function that is resistant to ASIC but at the same time quick to verify and execute.

The resistance to ASIC is guaranteed by the memory hard nature of the algorithm, given the use of a pseudo random data set initialized based on the length of the blockchain, therefore variable. This data set is called DAG (direct acyclic graph) and is regenerated every 30000 blocks, about 5 days. This period is called an era.

Currently the DAG has a size of about 2.3 GB, which is why it is not possible to mine Ethereum with video cards equipped with less than 2.5, 3 GB of video memory, as the data set is necessary for the how the hash function works.

**Casper.** Casper is the Proof of Stake protocol that Ethereum has chosen to follow. Casper's creator is identified as Vlad Zamfir, although in reality this project was carried out by a team he was part of. Casper has implemented a process by which malicious elements can be punished.

Operation:

- Those who validate bring a portion of their Ethers into play.
- After that, the blocks will begin to validate. It means that when they discover a block that they think can be added to the chain, they validate it by placing a bet on it.
- If the block is added, the "validators" will receive a reward proportional to their bet.
- However, if a "validator" acts maliciously, he will be immediately punished.

As you can see, casper is designed to work in a trustless environment. The advantages that this protocol can bring are:

1. Less use of electricity: the Ethereum network consumes 1 million $ in electricity, thanks to its PoW consensus mechanism. Through Casper, miners don't have to use powerful computers for mining, thus reducing electricity consumption.
2. Scalability improvement: the Ethereum network takes 15 seconds to generate a block, but the PoS system will reduce it to a few seconds. This will improve its scalability by making the network run even faster.
3. Improved security: the Ethereum network is currently not secure from the attack at 51
4. It is not necessary to generate so many new coins: in the PoS system, miners will no longer be motivated to validate blocks and add them to the chain. In the PoW protocol, the incentives for the miner are given in the form of ETH coins after the successful mining, they will be completely eliminated after the implementation of the PoS mechanism.

### A.1.5 Smart Contract

A Smart contract [18], also known as cryptocontract, is a program that directly controls the transfer of currencies or digital goods between the parties involved, taking certain conditions into consideration. A smart contract not only defines the rules and penalties related to an agreement in the same way a traditional contract does, but it can also automatically enforce those obligations.

Nick Szabo introduced this concept in the late 1990s, defining them as follows:

*"A smart contract is a computerized transaction protocol that executes the terms of a contract. The general objectives are to satisfy common contractual conditions (such as payment terms, liens, confidentiality, and even enforcement), minimize exceptions both malicious and*

*accidental, and minimize the need for trusted intermediaries. Related economic goals include lowering fraud loss, arbitrations and enforcement costs, and other transaction costs."*

That is, a computerized transaction protocol that executes the terms of a contract. Szabo suggested translating the contractual clauses (guarantees, bonds, etc.) into code and incorporating them into properties (hardware or software) that could be automatically respected, in order to minimize the need for trusted intermediaries between the parties carrying out transactions and occurrences of harmful or accidental exceptions [28]. Indeed, the advantages of smart contracts are more evident in business collaborations, in which they are typically used to impose a type of agreement, so that all participants can be certain of the result without the involvement of a third party. The rationale on which a smart contract is based is easily transcribed with the "if-this-then-that" sentence, if this happens then that happens.

This concept is easily understood through the following example:

- Alice and Bob have a boy named Charlie.
- Charlie is a shrewd child and his grades at school are poor.
- Alice and Bob must leave for a vacation in a place where they will only be accessible via email.
- So to encourage Charlie to do his best at school while they are away, they decide to make a deal with him.
- They tell Charlie that if he manages to take a 10 in a matter then he will get 5 $.
- Charlie accepts this agreement, but a problem arises. Charlie is impatient and wants a guarantee that if he succeeds he gets a 10 in a subject, then he will receive payment BEFORE his parents return home from their trip.
- But remember, Charlie is smart, and Alice and Bob can't trust him just by getting his vote via email. In fact, Charlie could easily lie about his vote, this only to get the 5 $ reward.
- So Alice and Bob write a smart contract and "block" the 5 $. This money will be released to the address of Charlie's Ethereum public wallet, only if he gets a 10. Alice and Bob entrust Charlie's teacher with the task of entering the grade (data) in the smart contract.
- Alice and Bob also give the teacher a cryptographically secure private key so that she is the only one able to write data in the smart contract.
- In this case, the teacher is defined as trusted oracle.
- Alice and Bob can rest assured knowing that Charlie will only be paid if his teacher inserts a 10 in the smart contract.
- And Charlie is happy because he knows he will get his reward if he works hard to get a 10, and will be paid before his parents return from their holidays.

A smart contract has the following four properties:

- Ability to run automatically
- Applicable

- Semantically valid
- Safe and unstoppable

Bitcoin was the first to implement smart contracts as a model for transferring value from one person to another, with its network of nodes that validates transactions only under certain conditions. However, since contracts can be coded on any blockchain, Ethereum turns out to be the most exploited platform for this purpose since it offers unlimited processing capacity. Ethereum, unlike Bitcoin which is limited to currencies, replaces the rather restrictive language of bitcoin with a language that allows developers to write their own Smart Contract programs.

**Smart contract structure.** The two main languages used to write Ethereum smart contracts are Serpent and Solidity.

Serpent is a high-level programming language designed to be very similar to Python, even if it has not been used much since September 2017. Vitalik himself considers an obsolete technology with a very low degree of safety compared to current standards. Solidity, also a high-level object-oriented language, and based mainly on Javascript, has become the most recommended language for writing smart contracts as it offers greater security and the possibility of making contracts interact with each other. A contract, in solidity, can consist of state variables, functions, struct, function modifiers, events, enumerative types and many others, the main ones will be shown below.

**State variables.** State variables are variables whose values are stored permanently within the contract.

```
contract SimpleStorage {

    uint storedData; // State variable
    // ...
}
```

**Functions.** Functions are the units of executable code, within a contract.

```
contract SimpleFunction {

    function bid() public { // Function
        // ...
    }
}
```

**Struct.** Struct are user-defined types that can group multiple variables.

```
contract Ballot {

    struct Voter { // Struct
        uint weight;
        bool voted;
        address delegate;
        uint vote;
    }
}
```

**Function Modifiers** Function modifiers can be used to change the semantics of functions declaratively.

```
contract Purchase {

    address public seller;

    modifier onlySeller() { // Modifier
        require(
            msg.sender == seller,
            "Only seller can call this."
        );
        _;
    }

    // Modifier usage
    function abort() public view onlySeller {
        // ...
    }
}
```

**Events.** Events allow clients to view changes in the Blockchain efficiently.

```
contract SimpleAuction {

    // Event
    event HighestBidIncreased(address bidder, uint amount);

    function bid() public payable {
        // ...

        // Triggering event
        emit HighestBidIncreased(msg.sender, msg.value);
    }
}
```

**Enumerate type.** Enums can be used to create custom types with a finite set of constant values.

```
contract Purchase {

    enum State { Created, Locked, Inactive } // Enum
}
```

**Development environment.** There are several development environments that can be used for Ethereum. However, it was decided to choose REMIX as it offers a more complete development experience due to its characteristics. Remix is a browser-based compiler and IDE, that is, it helps to write contracts directly from the browser. This open source tool allows users to create Ethereum contracts with the Solidity language, in addition, it also supports testing and debugging of various transactions. More details will be defined in the Technologies chapter.

### A.1.6 Oracle

Smart contracts running on blockchain networks have significant potential to increase efficiency and reduce transaction costs in different sectors. As explained previously, smart contracts have many advantages, but also some limitations.

Oracle is an important component of the smart contract ecosystem. The limitation of smart contracts is that they cannot access external data that may be needed to carry out checks. Oracle can be used to provide this data to smart contracts. An Oracle is, therefore, an interface that provides data from an external source to smart contracts. Depending on the sector and requirements, Oracle can provide different types of data ranging from weather reports, real world news and business actions to data from IoT devices.

They are also able to digitally sign the data proving its source and authenticity. Although the Oracle is reliable, in some cases there may be the possibility of data manipulation, altering its integrity. Therefore, Oracle should not be able to modify the data. [11]

We can define different types of Oracle that are used based on the type of use:

- **Software Oracles.** They are intended to manage the information available online. An example would be temperature, freight prices, flight or train delays, etc. The data therefore comes from online sources, such as company websites. The Oracle software extracts the necessary information and then enters it in a smart contract.

- **Hardware Oracles.** Some smart contracts need information directly from the physical world, for example a car that crosses a barrier where motion sensors have to detect the vehicle and send the data to a smart contract. The biggest challenge for Hardware Oracles is the ability to report without sacrificing data security.

- **Consensus Based Oracles.** Forecast markets like Augur and Gnosis rely heavily on Oracle to confirm future results. Using a single source of information could be risky and unreliable. To avoid market manipulation, forecasting markets implement an evaluation system for Oracle. For added security, a combination of several "oracles" can be used, where for example 3 out of 5 could determine the outcome of an event.

Oracle are third party services that are not part of the blockchain consent mechanism. The main challenge with Oracle is that people need to trust these sources of information. Whether it's a website or a sensor, the source of information must be reliable.

## A.2 Hyperledger Fabric

The growth in popularity of blockchain technology, initially launched by Bitcoin and followed by Ethereum, has led to the birth of many other projects, some of which are mainly focused in the enterprise sector, in particular in the Business to Business context.

In a permissionless blockchain anyone can participate and each participant is anonymous. In this context, we cannot count on the reliability of the players involved and therefore to mitigate this lack and guarantee the integrity and security of the system, incentive mechanisms are used for the participants. Monetary currencies are often used or obtained through mining processes or taxes paid by the various nodes for the validation of transactions.

Vice versa, a permissioned blockchain operates in a context where the participants are known, well identified and often controlled through governance models, characterized by a certain degree of reliability.

Being able to count on the knowledge of the participants' identities, a permissioned blockchain can use consensus protocols that do not require high computational costs, given the absence of a mining process. Furthermore, in this context, the risk of a participant intentionally introducing a malicious code through a smart contract is significantly reduced.

In a Business to Business context, where different organizations must communicate and exchange sensitive information, it is essential to use a blockchain that can guarantee the following properties:

- Participants must be identified / identifiable
- The network must be permissioned
- High performance in transaction processing
- Low latency in confirming transactions
- The privacy and confidentiality of transactions must be guaranteed
- There must be information relating to commercial transactions

In 2015, thanks to the Linux Foundation the Hyperledger project starts [90, 91, 92, 50], and several platforms based on blockchain technology were born, including Fabric [8]. While many blockchain platforms have been modified to adapt to enterprise use, Fabric has been designed for this purpose from the beginning.

In the following sections, the features provided by the Hyperledger Fabric platform will be presented in detail, the various components that characterize it will be presented, including the mechanism of the channels, the certification authorities, the ordering nodes, the consent mechanism and the smart contracts, however, before going into detail, it is useful to present the main functionalities and architectural choices underlying the project, which make the platform particularly useful for the context and differentiate it from other solutions.

### A.2.1 Functionality and main features

Fabric is characterized by a modular and configurable architecture, which guarantees versatility and optimization to different use cases in the industrial and commercial sector (for example banking, finance, insurance, health care, human resources, supply chain, distribution and digital content delivery) [41].

Considering a high level vision, Fabric includes the following modular components:

- *Ordering Service*: has the function of establishing consensus on the order of transactions and subsequently transmits the blocks to the nodes
- *Membership Service Provider*: deals with associating network entities with cryptographic identities
- *Peer-to-peer Gossip Service*: send the blocks, outgoing from the ordering service, to the other nodes
- *Smart Contracts*, which can be written in standard programming languages
- *Ledger*, which can be configured to support multiple DBMSs
- *Validation policies*, which can be configured independently for each individual application

Fabric supports smart contracts written with generic programming languages, such as Java, Go and Node.js, enabling many companies, who already know languages, to develop smart contracts without having to learn specific domain languages from scratch. [6]

Many blockchain platforms that support smart contracts, follow a **order-execute** architecture, according to which the consensus protocol:

1. Validate and order transactions and propagate them to all network nodes
2. Each node executes transactions sequentially

A smart contract that runs in a blockchain with this architecture must be deterministic, otherwise the consensus would never be reached. To solve this problem, many platforms require that contracts be written through a specific domain programming language in order to eliminate any non-deterministic operation (e.g. Solidity for Ethereum). This solution hinders its adoption, as it requires developers to learn new languages, wich can lead to more programming errors. In addition, since all transactions are executed sequentially from all nodes, performance and scalability are limited.

To solve these problems, Fabric introduces a new architecture, called **execute-order-validate**, which separates the whole process into 3 distinct phases:

1. a transaction is executed and its correctness is verified
2. transactions are ordered through a consensus protocol
3. the transactions are validated with respect to the control policies and are then entered in the ledger (a register, which will be described in the next section)

This architecture differs from that described above, in that the transactions are executed before the final consensus in the order they are reached and not subsequently. Furthermore, given that in Fabric the validation policies specify which and/or how many nodes must vote for the correct execution of a smart contract, each transaction can be performed only by the set of nodes necessary to satisfy a certain policy. This allows parallel execution of transactions, which increases the performance and scalability of the entire system. The first phase eliminates any kind of non-determinism, since any inconsistent result can be detected and filtered before the sorting phase.

It is precisely this feature that allows the writing of smart contracts with standard languages.

Fabric is a private permissioned platform, therefore it provides access control and identity verification systems, as well as privacy and confidentiality systems, based on the mechanism of the channels, thanks to which only participants in a certain channel can see the information in it. Furthermore, it supports various consensus protocols, thus ensuring high adaptability to contexts. Transaction sorting, as seen above, is performed by a modular component (Ordering Service), which is separate from the nodes that execute the transactions and that maintain the ledger. Fabric currently offers two implementations of Crash Fault Tolerant sorting services. The first is based on the Raft protocol, while the second is called Kafka.

### A.2.2 Hyperledger Fabric Composition

In this section all the various components that make up the Hyperledger Fabric blockchain platform will be discussed in detail, starting from the concept of ledger [42].

**Ledger.** The concept of ledger has a key role in Fabric and its function is to save objects (e.g. business events). More precisely, the current value of the attributes of the objects and the history of the transactions that led to these values are saved in the ledger, which can be defined as a real register.

To explain this concept we introduce the following example:

*A bank account has an available balance, which indicates the amount of currency that the account owner can spend at that time. If you want to understand how you got that available balance, then you can look at all the credit and debit transactions that led to it. The one just described is a concrete example of a ledger, in fact it includes a state (the available balance) and the ordered set of transactions (credit and debit) that determined it.*

The ledger does not literally save objects, which are generally saved in external databases, but instead saves the facts concerning the objects. The facts that are saved in the ledger allow us to identify the location of the objects, as well as many other key information about them. While the facts that describe the current state of an object may change, the history of the facts about an object is immutable; facts can be added to history, but facts already present in history cannot be changed.

To represent the state and history, Fabric uses two components: the world state and the blockchain ( Figure A.2) . These two components together form what is precisely called ledger.

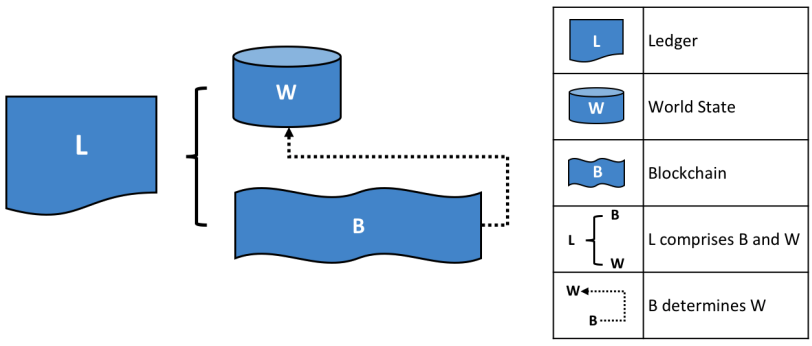



FIGURE A.2: World state W from blockchain B [6].

**World State.** As we saw earlier, the world state contains the current value of the attributes of a business object. This is useful as many applications often request the current value of an object and in this way they can obtain it simply by consulting the world state directly, without the need to cross the entire blockchain to calculate the final value of the object.

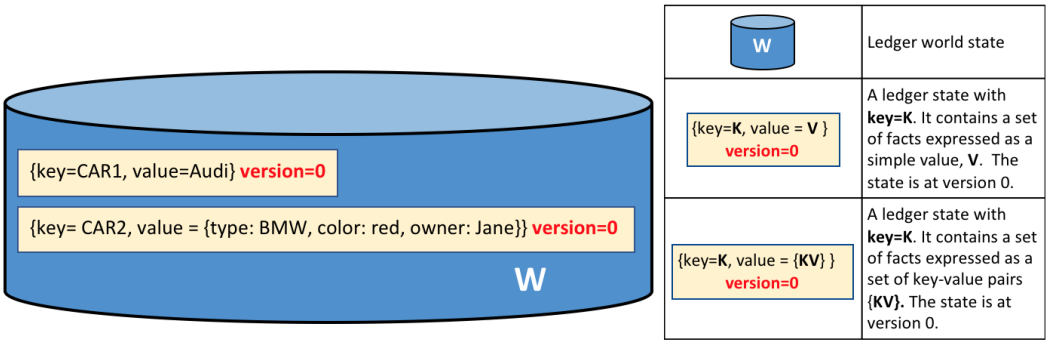



FIGURE A.3: World state with two state [6].

The example shown in figure A.3 shows two states of two machines, CAR1 and CAR2, each having a key and a value, which can be simple (CAR1) or compound (CAR2). An application can invoke a smart contract, which uses simple APIs for the ledger, allowing it to read, insert or delete states through specific functions (queries).

The world state is implemented through a database, as it is necessary to save information and have the functions available to be able to read and modify it.

Since modularity is a fundamental feature in Fabric, the database representing the world state can be implemented in different ways, in fact a relational, temporal, graph database can be used. Currently two implementation solutions are represented by LevelDB and CouchDB. While the first is preferable when the ledger states are simple key-value pairs the second is an appropriate choice when the states are structured as more complex JSON documents, as it supports queries and changes to more complex data types, often found in business transactions.

The version number, visible in the figure above and present in both states, is increased each time the state undergoes a change. The version is checked with every change, to be sure that the status is changing as expected and to check that there have been no concurrent changes.

Applications that generate transactions that can cause changes in the world state are isolated from the details of the consent mechanism. In fact, they simply invoke smart contracts and are informed when a transaction has been successfully inserted into the blockchain. The fundamental point is that only the transactions that are signed by the set of organizations will lead to a modification of the world state, vice versa it will remain unchanged.

**Blockchain.** As explained in chapter 2.1, where the general structure of the blockchain was described, each block contains a header, which includes the hash value of the transactions of the current block as well as the hash value of the block that precedes it in the chain.

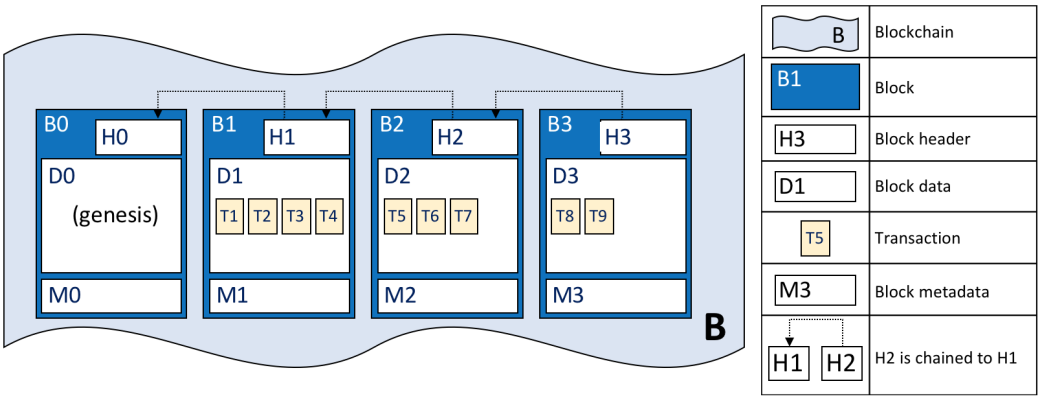



FIGURE A.4: Blockchain with four blocks [6].

The blockchain is implemented through a file, as there is no need to provide operations supported by a database, such as the set of queries available for the world state. The main operation in this case is the simple addition of transaction blocks at the end of the chain.

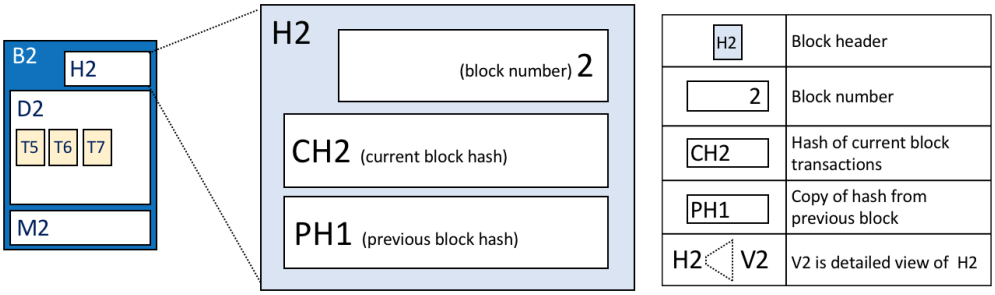



FIGURE A.5: Block structure [6].

**Blocks.** The structure of a block consist of 3 main parts:

- **Block Header** composed of:
  - **Block Number**: An integer that starts at 0 for the genesis block and is incremented by 1 each time a new block is added to the blockchain.
  - **Current Block Hash**: The hash value of all transactions contained in the current block.
  - **Previous Block Header Hash**: the hash value of the previous block header.
- **Block Data**: Contains the list of ordered transactions present in the block, it is generated by the ordering service when the block is created.

- **Block Metadata**: Contains the certificates and signatures of the creator of the block, which are used by the nodes of the network for its verification. Subsequently, a validation map for each transaction is added in this field; the hash value of the cumulative changes to the state is also added, in order to detect any forks. Unlike the fields above, the metadata is not inserted as input to the block's hash function.

**Transactions.** The figure A.6 shows the fundamental fields that make up a transaction in Fabric:

- **Header**: Contains some essential metadata about the transaction, for example the name of the chaincode and its version.
- **Signature**: Contains a cryptographic signature generated by the client application. This field is used to verify that the transaction details have not been changed, as it requires the application's private key to be generated.
- **Proposal**: Encodes the input parameters provided by an application to the smart contract, which generates the desired modification of the ledger.
- **Response**: Captures the values of the world state before and after the execution of the smart contract. It consists of the contract output and, if the transaction is successfully validated, will be applied to the ledger with the consequent modification of the world state.
- **Endorsements**: Contains a list of signatures from all organizations, with respect to the transaction response, required and sufficient to satisfy the validation policies.

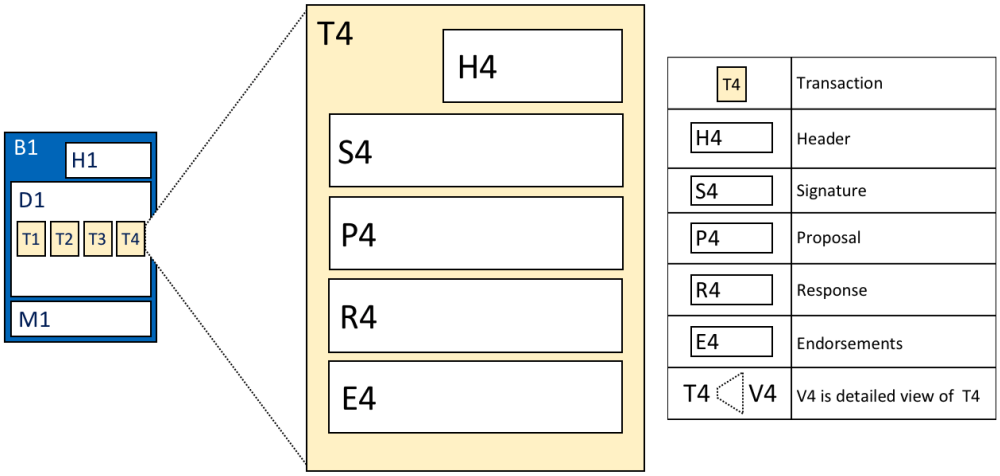



FIGURE A.6: Transaction structure in Fabric [6].

**Blockchain Network.** To present the architecture of Hyperledger Fabric, the example in the figure A.7 is used.

For blockchain network [43] in Fabric we consider a technical infrastructure that provides ledger and smart contract services to applications. Smart contracts are used to generate transactions, which are subsequently distributed to the various network

nodes, which have their own copy of the register, in which they save the transactions. Application users can be end users or network administrators. In most cases, a set of organizations join together to form a consortium and the various authorizations are based on a series of policies agreed by all consortium members during the network creation phase:

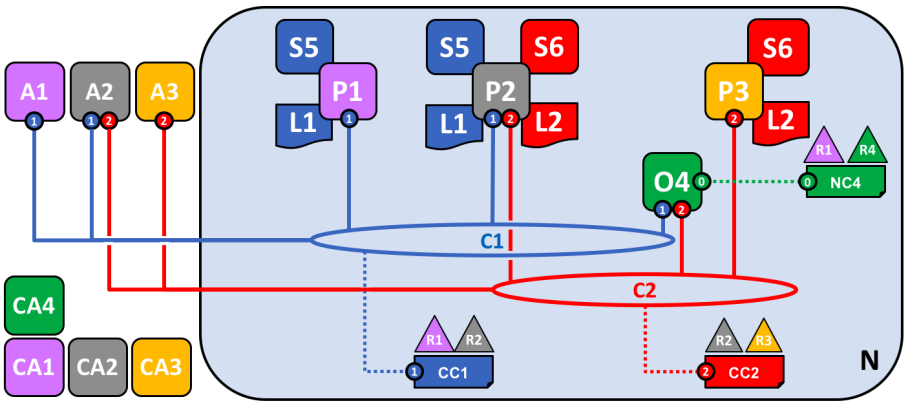



FIGURE A.7: Example network [6].

In the figure A.7 you can see the different components that will be described in the following sections, including:

- Organizations R1, R2, R3, R4
- The Certification Authorities CA1, CA2, CA3, CA4
- The Ordering Service O4 and the configuration document of the NC4 network
- The channels C1 and C2 and their configuration documents CC1 and CC2
- Applications A1, A2, A3 that have access to channel C1, C2 or both in the case of A2
- The 3 peer nodes P1, P2, P3
- The smart contracts S5 and S6
- The ledgers L1 and L2

In the figure A.7 there are 4 organizations R1, R2, R3, R4 that have decided to participate in the network. R4 is considered as the initiator of the network, whose rules (policies) are specified within an NC4 configuration document, under the control of R1 and R4. R1 and R2 require private communications between them within the network, as well as R2 and R3. This is made possible thanks to the mechanism of the channels, in fact the organization R1 has a client application A1 that can perform business transactions within the channel C1, making them effectively invisible outside the channel, while R2 does the same job through application A2, which however has access to both channels C1 and C2.

Organization R3, through application A3, has access to channel C2. In the network presented there is also a node P1, which keeps a local copy of the ledger L1 associated with channel C1 saved, and is also able to execute the S5 smart contract. The P2 node, having access to both communication channels, in addition to keeping a copy of L1, also saves a copy of L2 and is able to execute the smart contracts S5

and S6. P3 keeps a copy of L2 and has installed S6. The two channels C1 and C2 are managed in accordance with the policies specified in the respective configuration documents CC1 (controlled by R1 and R2) and CC2 (controlled by R2 and R3). In the scheme you can also notice the O4 node, called the Ordering Service, which can be considered as the network administration point. O4 interacts with the application channels C1 and C2, as it performs the functions of sorting transactions, generating blocks and distributing them. Each of the 4 organizations is based on a different certification authority: CA1, CA2, CA3, CA4, which have the function of providing certificates to administrators and network nodes.

**Certification authority.** In Fabric, the "Certificate Authorities" [44] have the function of providing digital identities to administrators and network nodes, in the form of cryptographically verified digital certificates. They can be considered as trusted, secure and known entities by all members who participate in the network.

The various nodes of the blockchain platform use certificates to identify themselves as belonging to a particular organization, for this reason, generally different organizations use different certification authorities.

Certificates are also used for signing transactions, this allows nodes, which keep the copy of the ledger saved, to verify that the transactions are valid and then add them to the blockchain.

Certificates, generally of the X.509 type, are digitally signed by the certification authority and bind an actor together with the respective public key, asymmetric encryption is then used to ensure the security and secrecy of the information. The idea is that if a participant trusts a certain certification authority, then they will also trust a node that is trusted by the same authority they trust.

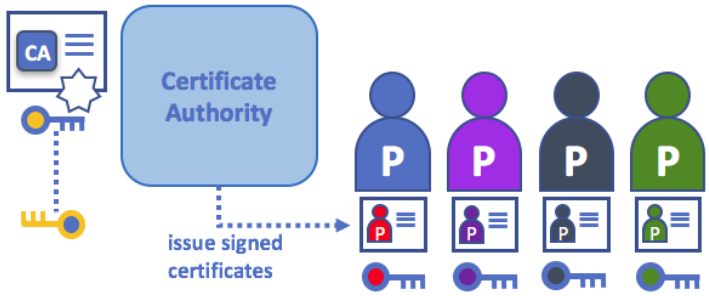



FIGURE A.8: Certificate Authority [6].

**Reliability chain.** Since the certification authorities must safely distribute large quantities of certificates to the various users, it makes sense to extend the process by inserting two different types of CA:

- **Root CAs**
- **Intermediate CAs**

A "chain of trust" is established between a Root CA and a set of Intermediate CAs where a certification authority is considered reliable if its certificates are signed directly by the Root CA or if they are signed by another Intermediate CA belonging to the chain which, if travelled in the reverse direction, leads to Root CA.

The certification authorities exploit another component of the network, called the Membership Service Provider (MSP), which has the function of mapping the various certificates provided by the CAs with respect to the various members of the organizations, as we will see in the next section.

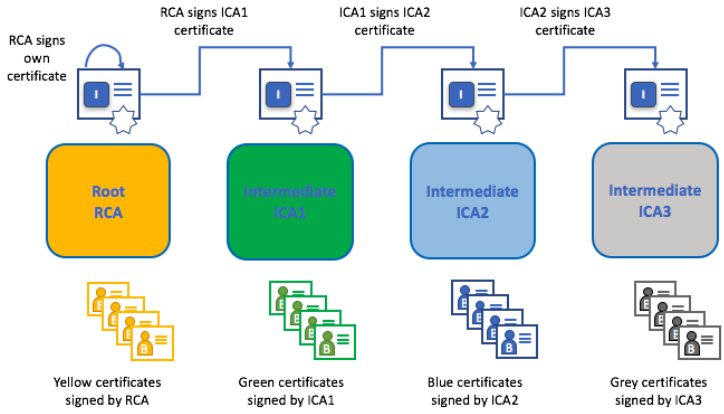



FIGURE A.9: Reliability chain [6].

**Membership Service Provider (MSP).** The main function of the MSP [45] is to identify which certification authorities are trusted to define members of a certain domain. In the case of an organization, listing the identities of those members or defining which certification authorities are authorized to provide valid identities for members.

The capabilities of an MSP go beyond simply affirming who a particular participant in the network or a member of a channel is, in fact it is able to identify the specific roles that an actor has within an organization and lays the foundations for definition of access privileges on the network or on a channel.

An organization is an organized group of more or less extensive members and is typically represented by a single MSP, however in some cases, large organizations may require the presence of different functional groups, which therefore carry out operations in different business contexts. In these cases, it makes sense to have more than one MSP, which configurations are:

- A single MSP defines the list of all members of an organization
- Several MSPs are defined to represent different organizational groups:
  1. National
  2. International
  3. Government

There are basically two types of Membership Service Provider:

- Local MSP: is located locally inside a node
- Channel MSP: found in a channel configuration file

**MSP structure.** You can think of the structure of an MSP as a list of folders:

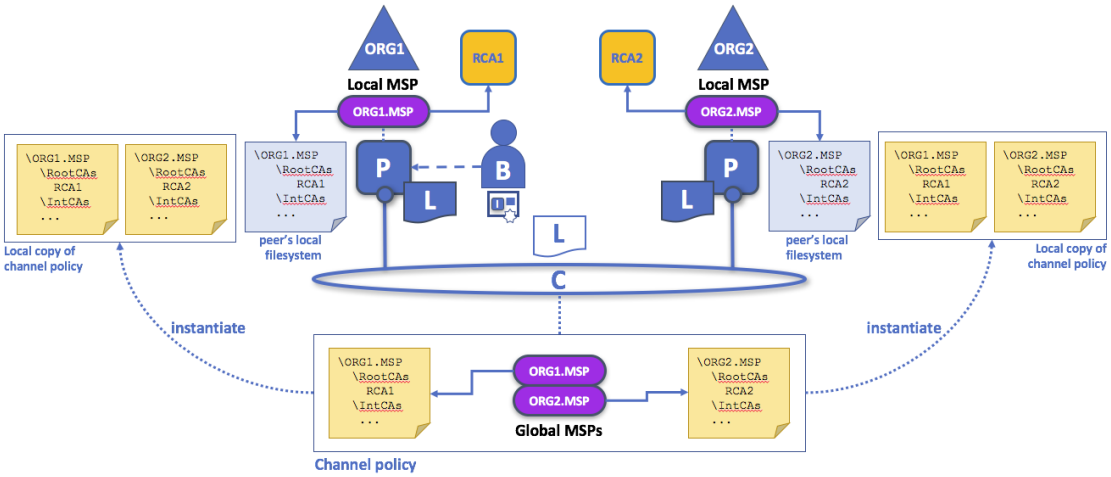



FIGURE A.10: Local MSP and Channel MSP [6].

An administrator B connects to a node with an identity provided by RCA1 and saved in a Local MSP. When B tries to install a smart contract on the node, it checks its Local MSP ORG1-MSP to verify that B is a member of ORG1. Subsequently B wants to insert the smart contract within channel C. In this case all the organizations participating in this channel must agree, therefore the node must check the various channel MSPs in order to proceed with the operation.

- **Root CAs**: contains a list of X.509 self-signed certificates of Root CAs considered reliable by the organization represented by this MSP. It is very important as it identifies the certification authorities from which all other certificates must be derived to be considered members of the corresponding organization.
- **Intermediate CAs**: contains a list of X.509 certificates of Intermediate CAs considered reliable by this organization. Each certificate must be signed by one of the Root CAs in the MSP or by an Intermediate CA which is directly or indirectly connected to one of the Root CAs.
- **Organizational Units**: an organization is often divided into several organizational units, each of which has different responsibilities. In this folder there is a list of all the units into which the current organization has been divided.
- **Administrators**: Contains the list of identities that have the role of administrators for this organization.
- **Revoked Certificates**: if the identity of a member has been revoked, all information relating to that identity can be found in this folder.
- **Node Identity**: contains cryptographic material which, in combination with the contents of the Keystore folder, allows the node to authenticate itself in the messages it sends to the other participants of a channel and network.
- **Keystore (for Private Key)**: Contains the node signing key, which is used to sign the information.
- **TLS Root CA**: contains a list of self-signed x.509 certificates of Root CAs considered reliable by the organization represented by this MSP for Transport Security Layer (TSL) communications, such as connecting a node with a Ordering Service to receive ledger updates.

- **TLS Intermediate CA**: contains a list of X.509 certificates of Intermediate CAs considered reliable by the organization represented by this MSP for TSL communications.

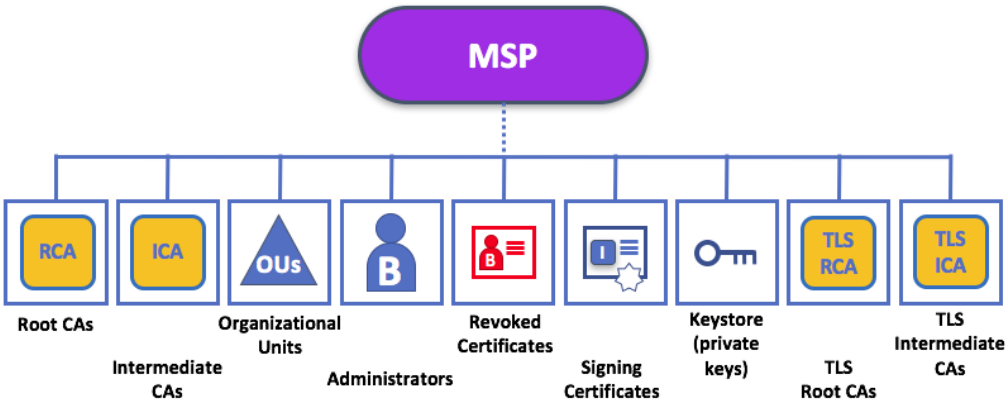



FIGURE A.11: How Local MSP are saved in a local filesystem [6].

**Channels.** A channel is a means of communication through which the members of a consortium, that is a set of organizations, are able to exchange information with each other. [46] As already mentioned in the previous sections, in a B2B context it is essential that the information exchanged by the various organizations remains private. The channels serve precisely to guarantee this, in fact they provide an efficient mechanism for the private exchange of information between the members of a consortium. They provide privacy from other channels and from the network and allow organizations to share with all public information and at the same time to keep information private to be disclosed only to a small group of participants.

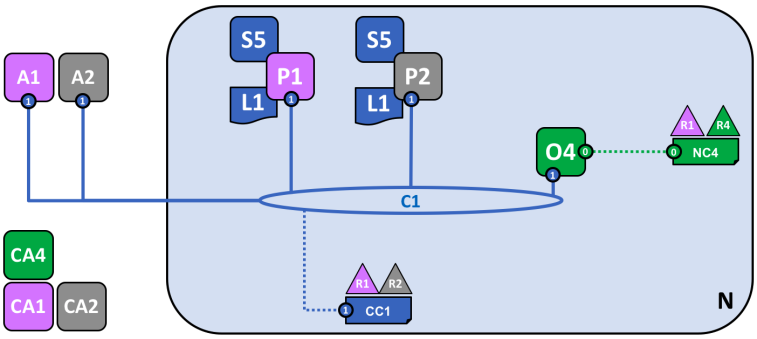



FIGURE A.12: Channel example [6].

Each channel is managed thanks to a document called channel configuration, which is generated and can only be modified by organizations that take part in the specific channel. This document is separate from the network configuration, therefore a change in the network configuration does not affect the channel and all data exchanged on one channel are completely isolated from the rest of the network, including the other channels.

In the figure A.12 you can see the CC1 channel configuration and the NC4 network configuration. Channel C1 is managed by organizations R1 and R2 and this channel has access to the two client applications A1, A2 and two peers P1 and P2,

which locally save the L1 ledger and the S5 smart contract. A client application (eg A1) associated with an organization (eg R1), can use S5 to access the ledger L1 through node P1. This shows how important channels are in Fabric for communication between the various components of the network.

A channel can have an unlimited number of organizations participating in it and an organization can participate in multiple channels simultaneously, as can be seen in the diagram in the figure A.7 presented at the beginning of this chapter, where the P2 node belonging to the organization R2 participates in both channel C1 and channel C2. When a node wishes to enter a channel, it sends the request to the ordering service, in this case O4 (we will see later how it works and the functions it makes available), which uses the configuration of the channel to determine whether the node in question has permissions on it, for example if the node can read or write information on the L1 ledger.

**Network and channel configuration.** The configurations referring to the various channels and the network contain all the policies agreed upon by the participating members, which provide a concrete and shared reference for controlling and verifying access to the various resources. In addition to this, the configurations contain facts regarding the composition of the network and the channels themselves, such as the name of the consortia and organizations that are part of it.

1. **Network Configuration**: When creating a network, a specific Ordering Service (eg O4) is in charge of its training and its behavior is governed by a network configuration created previously (eg NC4). This implies that although the Ordering Service (which generally consists of several nodes) is the entity that creates the consortia and the various communication channels, the intelligence of the network is contained in the configuration on which the Ordering Service is based, which therefore effectively controls access to the network. So as long as O4 correctly implements the rules defined in NC4, the network will behave as all organizations have foreseen and agreed.

2. **Channel Configuration**: The same principles seen for network configuration are applied here to peers. When the nodes interact with client applications (eg P1 and P2 with A1 and A2), they use the rules defined within the configuration of the channel in which they are communicating (eg CC1) to control access to the resources present in it . So the various behaviors are dictated by the policies previously defined in the configuration files.

Being in the context of a platform based on blockchain technology, although there is logically a single configuration, this is replicated and therefore kept consistent by each node that forms the network or that participates in a channel. This is done through the same system used for user transactions and in this case we talk about configuration transactions.

**Mini-Blockchain.** The nodes of the Ordering Service actually generate a blockchain, connected by a special channel, called the system channel, used to distribute network configuration transactions to all nodes. These transactions are used to cooperatively maintain a consistent copy of the network configuration and therefore to make it editable only under mutual agreement. The same thing is done by peer nodes, using the various application channels to distribute the channel configuration transactions.

This characteristic of having objects that are logically individual, but that are physically distributed among various components of the network, is a recurring model in Fabric, in fact we can see it applied also with respect to ledgers and smart contracts, which logically exist at the channel level, but which are physically replicated on the various nodes. This allows the platform to be decentralized and easy to handle at the same time.

**Nodes.** The nodes [47] are a fundamental element of the network as they host copies of the ledger, containing all the transactions and copies of the smart contracts that generate them. If an administrator or an application wants to access resources, it must necessarily interact with a node that contains them.

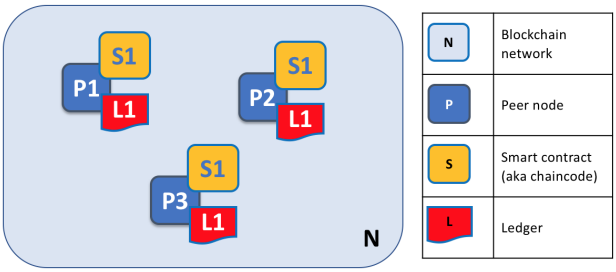



FIGURE A.13: Network consisting of 3 nodes, on which the S1 contract is installed [6].

Nodes can be created, started, stopped, reconfigured and deleted; These operations are performed through a set of APIs that allow administrators and applications to interact with the services provided by the nodes.

As shown in figure A.13, a node can host many ledgers and many smart contracts and there is no fixed relationship between the number of the ones and the others, in fact a ledger can have several smart contracts that allow you to access it. This is possible when a node has the permissions to communicate on more than one channel and therefore it saves all the ledgers representing the various channels, as well as the smart contracts.

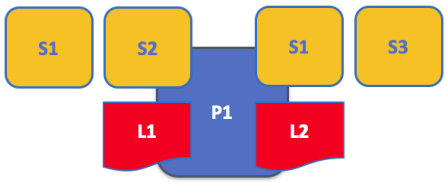



FIGURE A.14: Example node with two ledgers and three smart contracts [6].

In Hyperledger Fabric, nodes can take on different roles, even multiple depending on how the network is configured. Consequently, **4 main types of nodes** can be listed:

- **Committing Peer**: each node in a channel is considered a Committing Peer, as it receives the transaction blocks and validates them before they can be added to the local copy of the ledger.

- **Endorsing Peer**: each node with a smart contract installed is considered an Endorsing Peer if the contract is used by a client application to generate a signed transaction response. So in fact these nodes take part in the process of endorsement of transactions, which is fundamental for their generation and final validation.
- **Leader Peer**: when an organization owns several nodes in a channel, a node must take responsibility for distributing transactions from the sorting service to the other nodes. The Leader node can be chosen through a static or dynamic selection process.
- **Anchor Peer**: when a node needs to communicate with another node belonging to a different organization, it can use an Anchor Peer, defined in the channel configuration for that organization. It is an optional node, which simplifies and therefore makes communication in scenarios with many participating organizations more efficient.

A node can belong to all the types listed above, as they are not mutually exclusive, but simply add functionality.

**Gossip Protocol.** Another functionality of the nodes is to distribute the channel information and the various ledger updates to all the other nodes belonging to the channel. This is done through a protocol called Gossip, thanks to which the sending of messages takes place continuously and each node is constantly updated by many others on the state of the ledger. Every message sent through this protocol is signed, in order to guarantee security and therefore avoid false messages or messages sent to incorrect recipients. This information distribution protocol performs two primary functions in a fabric network:

1. manages the discovery of new nodes and the members of a channel, continuously identifying the new nodes and detecting those that have gone offline.
2. disseminates ledger data to a number of nodes on a channel. Each node with information that is not synchronized with the rest of the channel, identifies the missing blocks and adds them to its own copy of the ledger.

**Interaction between nodes and applications.** The A.15 figure shows the phases that make up the interaction between a node P1 and a client application A, which wishes to access the L1 ledger.

An application may want to interact with the ledger for two reasons:

1. read the information inside it through query interactions.
2. update the status of the ledger through ledger-update interactions

In the first case, the result of a query can be returned immediately to the application that requested it, since all the information useful for satisfying it can be found in the local copy of the ledger inside the node. In the second case, however, a single node cannot generate a change in the state of the ledger, but must wait for other nodes to agree with the update in question and this is done through the consent mechanism and with the help of the nodes of the ordering service, which we will see in detail in the following sections.

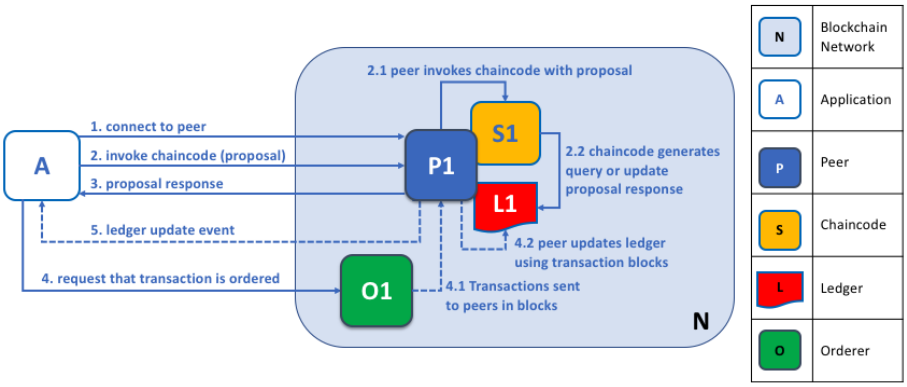



FIGURE A.15: Interaction between nodes and applications [6].

**Nodes and organizations.** The nodes that make up a network in Fabric are controlled by different organizations, in the example in the figure A.16 we find 4 different ones. It can be seen that the C channel connects 5 nodes together (P1, P3, P5, P7, P8), effectively communicating all four organizations.

The network is formed and managed by a set of organizations, which by collaborating, make resources available. This whole structure could not stand if organizations did not make available the various nodes that hold together and make the network work. This highlights the decentralized nature of the platform.

The various applications controlled by a certain organization can connect to nodes controlled by the same organization or by others, depending on the nature of the interaction with the ledger.

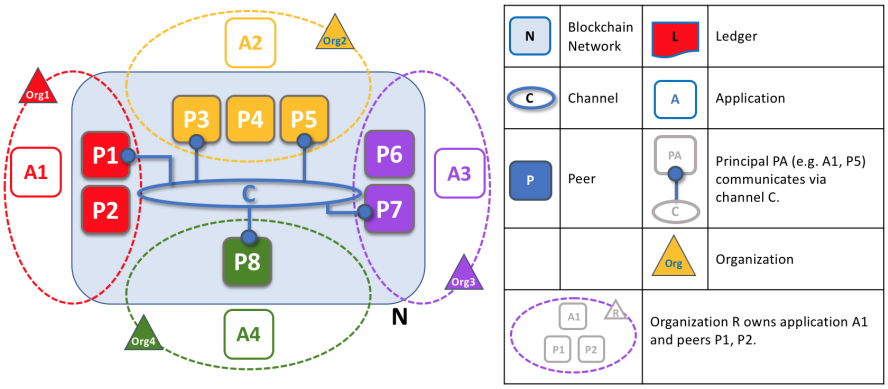



FIGURE A.16: Network formed by several organizations that communicate through channel C [6].

**Nodes and Identity.** As we have seen in the section dedicated to the Membership Service Provider, each node has a precise identity, which is assigned to it in the form of a digital certificate by a certain certification authority.

When a node connects to a channel, its digital certificate identifies its organization, thanks to an MSP. In the A.17 figure, the identities of P1 and P2 have been provided by the CA1 certification authority and channel C determines, based on a policy present in its configuration, that all the identities provided by CA1 are associated with the Org1 organization, using the membership service provider ORG1.MSP.

In the same way P3 and P4 are identified by ORG2.MSP as part of the Org2 organization.

Regardless of where a node is physically located (in the cloud, in a data center or on a local machine), it is the associated digital certificate that indicates its membership in a specific organization.

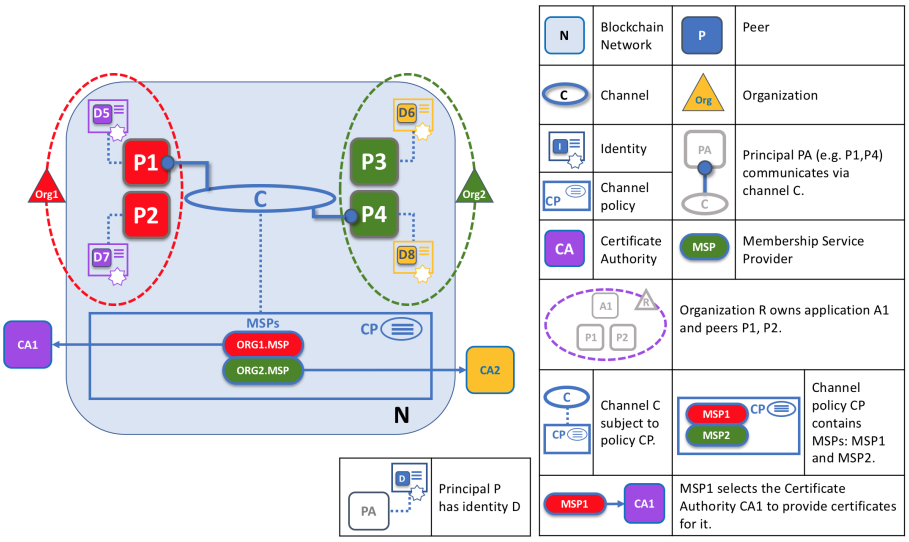



FIGURE A.17: Assigning identities to the nodes of the network[6].

**Ordering Service.** We have seen how the nodes that make up the blockchain network can locally maintain a copy of the ledger and smart contracts useful for generating and managing transactions. These nodes, like all network components, have an identity and are able to connect with channels and applications, allowing the exchange of information between companies, thus allowing the execution of business processes.

However, there is another type of node besides those listed above, called "Orderer" cite ordering, literally authorizing officer, very important and necessary for the proper functioning of the network. This node can be part of a larger set consisting of nodes of the same type, called the Ordering Service.

As mentioned above, when a request to update the ledger occurs, in order for this to be satisfied, a certain number of nodes must approve this change; when this occurs, the new transactions are entered in the ledger, effectively changing their status and all connected applications are informed by the nodes. The process just described takes the name of consent and the Ordering Service plays a fundamental role for its correct execution.

**Ordering Service tasks.** Here are the main task for the ordering service:

- Maintains the list of organizations that have permission to create channels, known as the consortium. This list is kept in the configuration of a particular system channel, to which only the ordering service has access and which can only be modified by an Admin.
- Manages access control to channels, limiting who can read and write on them and who can configure them. Remember that the permissions that the various members have on a channel are subject to the policies that the administrators generated when creating a consortium or channel.

- Sorts the transactions, generates the blocks and distributes them to the various nodes for verification. Let us now see the different stages that characterize this process.

**Ordering Service Phases.** Let us analyze all the phases that characterize the process of ordering transactions, creating blocks and inserting them into the various copies of the ledger distributed.

- **Phase 1: Proposal.** In the first phase, a client application sends a transaction proposal to a set of Endorsing peers, effectively requesting the approval of the various organizations with respect to the transaction. Each node belonging to the set will independently execute a smart contract (chaincode), using the transaction proposal response to generate a response to the same (transaction proposal response), which is then signed and sent to the application. At this time no changes have been made to the ledger yet. Once the application has received a sufficient number of responses signed by the endorsing peers, the first phase can be considered finished.

  The set of nodes chosen by an application to generate responses depends strictly on the endorsement policies defined in the smart contracts, which list the organizations that must necessarily sign and approve the information, before it can be considered definitive. This approval consists in inserting the digital signature of the organization within its reply and the subsequent signature of the entire information payload with its own private key. This allows the application and the other components of the network to verify by public key of that organization that the information has been verified and approved by you.

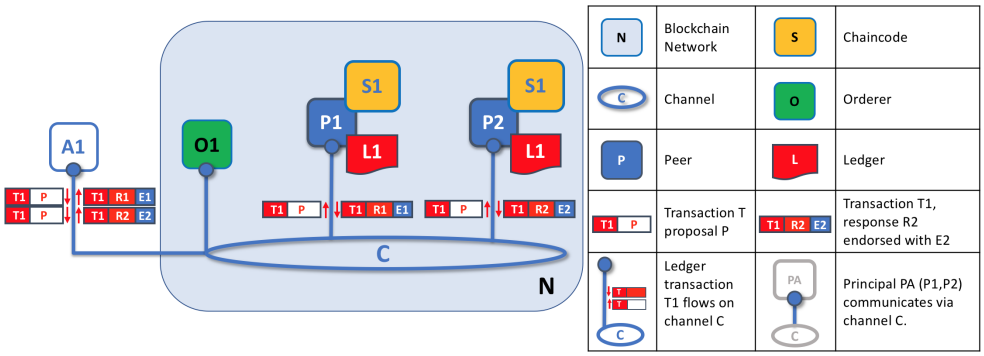



FIGURE A.18: Proposal phase [6].

Application A1 generates a proposal P for transaction T1 and sends it to nodes P1 and P2 on channel C. P1 executes S1 using P and generates the response R1, which it approves with E1. P2 does the same thing and application A1 thus receives two approvals of its transaction.

- **Phase 2: Sorting and packaging of transactions in blocks.** In this phase the nodes that make up the Ordering Service simultaneously receive different transactions from different client applications. These nodes work together to arrange groups of transactions in a well-defined sequence and package everything in blocks. The number of transactions in a block depends on some

channel configuration parameters relating to the desired size and the maximum duration spent for a block. The blocks are then saved in the ledger of each ordering node and distributed to all the nodes participating in the channel. If a node is offline or subsequently enters the channel, it will receive the blocks upon reconnection with an ordering node, or for "gossiping" with another node of the channel.

As you can see in the figure A.19, the order of the transactions present in a block is not necessarily the order of arrival to the ordering service, since there can be many nodes, belonging to the service, which receive transactions approximately in the same moment. The important thing is that the ordering service places the transactions in a fixed and rigorous order, on which the nodes will then base themselves for validation and for reaching a final consensus.

While in other blockchains the same transactions can be inserted in different blocks, which will then compete to be added to the chain (as happens in Bitcoin with the mining mechanism of the Proof of Work), In Fabric the blocks generated by the ordering service are **Finals**. Once a transaction has been written into a block, its position in the ledger is immutably ensured. This means that ledger separations, called "ledger forks", cannot occur: Validated transactions will never be restored or eliminated.

Another key point is that while all nodes, which are not part of the ordering service, perform smart contracts and process transactions, the ordering nodes do not. They mechanically block the transactions they receive, without giving any judgment on their content (except for channel configuration transactions).

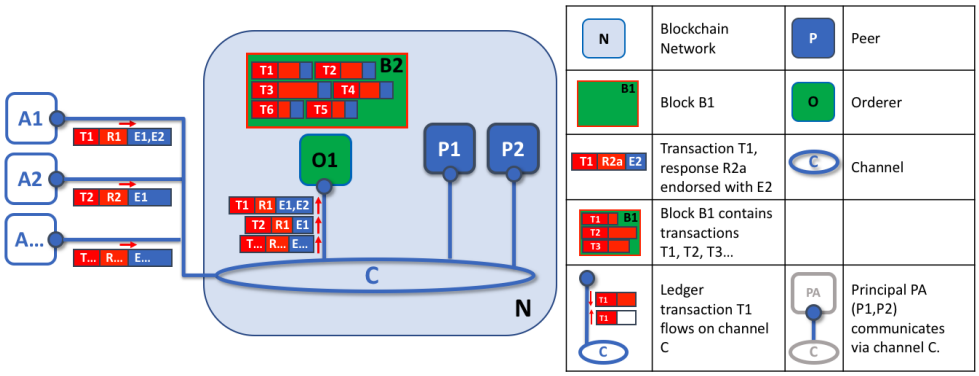



FIGURE A.19: Ordering phase [6].

Application A1 sends a T1 transaction, signed by E1 and E2 to the Orderer O1, while A2 sends T2 signed by E2 to O1. The Orderer bundles these two transactions with others from different applications, thus creating block B2.

- **Phase 3: Validation and Insertion.** The last phase consists in the distribution and consequent validation of the blocks from the ordering service to the various nodes, where they can be added to the ledger.

  At each node, all transactions within a block are checked to verify that they have been signed correctly by all the necessary organizations, and then added to the local copy of the ledger. The transactions considered incorrect are in any case kept within the block created previously (as immutable), but are marked as invalid and therefore do not change the state of the ledger.

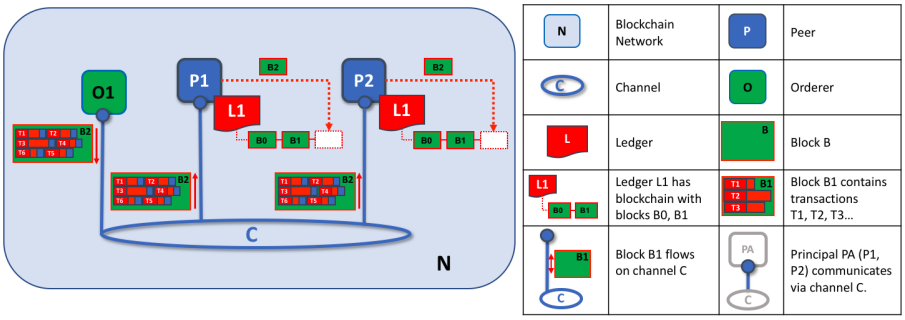



FIGURE A.20: Validation and insertion phase [6].

The orderer O1 distributes block B2 to nodes P1 and P2. The node P1 processes B2, checking the validity of the transactions present in it and adds it to the local copy of the ledger, containing the blocks B0 and B1 in succession. P2 node does the same.

The fixed order of transactions within the blocks allows the nodes to confirm that the ledger updates are applied consistently and identically on the entire blockchain network.

The verification process present in this phase is different from that of phase one, where the endorsing peers validated the transactions by signing them with their own certificates and then sending the signed response to the applications that requested it. In the event that an application has violated the rules by sending incorrect transactions to the ordering service nodes, the nodes are able to verify it and identify that violation in the last phase, through this final verification of all the signatures. As said, if something does not return in a transaction, the nodes mark it as invalid and this cannot in any way change the state of the ledger.

If a node verifies that a transaction is correct, then it will try to insert it into its local copy of the ledger, however it must first check that the current state of the ledger is compatible with what was the state at the time the transaction was generated. . This check could reveal an incompatibility for several reasons, for example another transaction could have changed the same group of information in the ledger, making the new change no longer valid and therefore no longer applicable to the current ledger. In this way the blockchain is kept consistent on all the nodes of a channel, since all of them follow the same validation rules.

Two important characteristics can be noted in this process:

- The only phase that requires the execution of smart contracts is the first and this allows you to make the contract logic (chaincodes) accessible only to endorsing peers, hiding it from all other nodes in the network. This property allows to improve confidentiality and scalability.
- Whenever a block is added to a ledger, the node that added it generates appropriate events:
  * **Block Event**: includes the entire contents of a block.
  * **Block Transaction Events**: include summary information, such as indicating which transactions in the block have been validated and which are not.

* **Chaincode Events**: Contain information that was produced by the execution of smart contracts. The various applications can be registered to all these types of events, in order to be notified when they occur.

### A.2.3 Consensus mechanism

The previous sections described the ordering service and the features that this component makes available on the network, including sorting the transactions in a fixed sequence and inserting them into the blocks. However, it has not yet been explained how the various orderers nodes, which together form the ordering service, manage to reach a consensus on the order of transactions, which in fact must be the same for everyone. In this section we will present the consensus mechanism implemented in Fabric, which serves this purpose and which is called Raft [67] (in the first versions of the project there was a simplified version of the algorithm, called Kafka [87], which has now been completely replaced by Raft).

**Raft.** The implementation of the Raft protocol in Fabric follows a "leader follower" model, in which a node is dynamically elected as leader among the ordering nodes belonging to a channel and its decisions are replicated by the various followers.

This protocol is called Crash Fault Tolerant (CFT), as it can sustain the loss of nodes, including the leader, provided that the majority, defined as quorum, is still active. Furthermore, Fabric's idea is to make the protocol also Byzantine Fault Tolerant (BFT) and we will see how some design choices lead precisely in this direction.

Before explaining the functioning of the consensus mechanism, some fundamental concepts will be defined in the Raft [48, 67] protocol:

- **Log Entry**: is the main unit of work in a Raft ordering service. The sequence of all entries is called "log". The log is considered consistent if the majority of members agree on the entries and their order.
- **Consenter Set**: are the ordering nodes that actively participate in the consensus mechanism for a given channel and receive the replicated logs for the same.
- **Finite-State Machine** ( textbf FSM): each ordering node in Raft has an FSM and all together are used to make sure that the sequence of logs present in the various nodes is deterministic (must be the same for all ).
- **Quorum**: indicates the minimum number of nodes will be allowed to allow transactions to be sorted. If a quorum of nodes cannot be reached for any reason, then the service will no longer be available for any read or write operation on the channel and no new logs will be accepted.
- **Leader**: a Consenter Set for a given channel elects a single node as leader, who is responsible for obtaining the new log entries, replicating them to the follower nodes and managing when an entry is considered "committed". A leader is not a special node, but simply a role that an orderer can take on certain times, depending on the circumstances.
- **Follower**: these nodes receive logs from the leader node and replicate them deterministically, making sure they remain in a consistent state. We will see later how this happens.

A node in Raft can be in separate **3 states**: Follower, Candidate, Leader and changes state according to the occurrence of certain situations, as explained below. The algorithm is characterized by 2 main phases: the election of the leader node and the log replication.
**Leader Election**: In this phase the various steps that lead a node to be elected as leader are listed.

1. Initially all nodes are in the Follower state and each node is assigned a random timer, called the election timeout (between 150 and 300 ms).
2. When a timer on a node reaches zero, that node goes into the Candidate state
3. The Candidate node starts a new voting phase, called the election term, voting itself and sending a message (Request Vote) to all the other nodes, asking to vote for it.
4. If a node that receives a Request Vote has not yet cast a vote, then it will vote for that candidate node, sending it a message and resetting its timer.
5. If a Candidate node receives the total majority of votes (what is defined as Quorum), then it goes into the Leader status (this ensures that only one leader exists at a time) and all the other nodes, seeing that that node has the majority of the votes will go into the follower status and therefore will consider that node as their leader.
6. The Leader, in order to maintain his role, must communicate continuously with all the following nodes and does so by sending him a message (Append Entries) with a regular rhythm (heartbeats). All followers nodes who receive the message will reply in turn. The process continues until a follower no longer receives the message and then goes into Candidate status, generating a new election process.

**Log Replication**: Once a leader has been elected, his job is to replicate all the changes that occur in the system to all the follower nodes.

1. As shown in the second phase of the block creation and insertion process, client applications send signed transactions to the ordering service, in particular to the leader node.
2. The leader inserts the information obtained in his log, without considering it still definitive (not committed).
3. Next, always the leader, send the information to all the follower nodes, using the next "heartbeat".
4. The nodes that receive the message in turn save the information in their logs and send a confirmation message to the leader.
5. When the leader receives a number of confirmations equal to or greater than the majority of the nodes present, then he can consider the information saved in the final (committed) log.
6. Finally the leader sends a response message to the client application to communicate the update.

In this way the nodes that are part of the ordering service are able to reach a general agreement on the order of the transactions to be inserted within the blocks, thus allowing the passage to the final phase of distribution of the same to the various nodes of the network belonging to a channel, which will verify them and insert them into the ledger.

**Notes:**

- Each channel is based on a separate instance of the Raft protocol, this allows each instance to elect a different leader and allows a better decentralization of the service in contexts where there are different clusters of nodes belonging to different organizations.
- While all ordering nodes must have access to the system channel, which is used by them to communicate and exchange information, as regards application channels, not all nodes must necessarily be part of it. In fact, the creators of the channels and the admins decide which available ordering nodes to add or remove from them.
- A transaction is automatically routed, from the authorizing node that receives it, to the node that is currently in the leading state for that channel. This means that nodes and applications don't need to know the identity of the leader.

### A.2.4 Smart contact on hyperledger: Chaincode

Fabric introduces a concept closely related to smart contracts, called Chaincode [49], which represents the way in which contracts are grouped and organized to be distributed to various nodes and channels for installation.

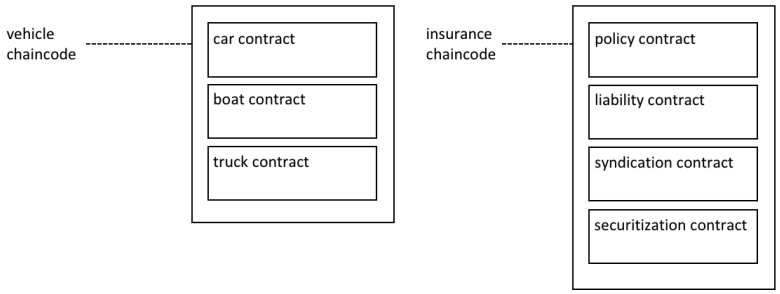



FIGURE A.21: Chaincode structure [6].

Different smart contracts can be defined within the same chaincode. When this is distributed, all smart contracts within it are made available to applications.

A chaincode can therefore be considered as the container of a set of smart contracts, linked by the same business context. In the figure above we see a chaincode containing contracts related to the insurance field and another concerning vehicles.

**Functions:** A smart contract is used to read and write access to the ledger, in particular it can communicate both with the world state and with the blockchain, through these functions:

- **get**: usually represents a query to obtain information regarding the current state of a business object.

- **put**: generate a new business object or modify one already present in the world state.
- **delete**: removes a business object from the current state of the ledger, without changing its history, saved in the immutable blockchain.

These interrogation and modification operations of the ledger are defined in different APIs (Application Program Interfaces), referring to several categories.

Within a smart contract there are the definitions of the various transactions, which once performed allow you to carry out the operations on the ledger mentioned above.

```
async createCar(ctx, carNumber, make, model, color, owner) {

    const car = {
        color,
        docType: 'car',
        make,
        model,
        owner,
    };

    await ctx.stub.putState(carNumber, Buffer.from(JSON.stringify(car)));
}
```

FIGURE A.22: Transaction for the creation of a car

Once called from an application, the function described in the figure A.22 will be sent to the various nodes of the system and the process described in the previous sections will begin. If everything proceeds according to the established rules and the transaction is approved and signed by all the necessary participants, then it will lead to a modification of the world state, in which the newly created business object will be inserted.

Fabric supports two runtime development environments **Java Virtual Machine** and **Node.js**. This allows developers to write smart contracts using any language executable on one of the two environments just described, such as **JavaScript**, **TypeScript**, **Java** and **Go**.

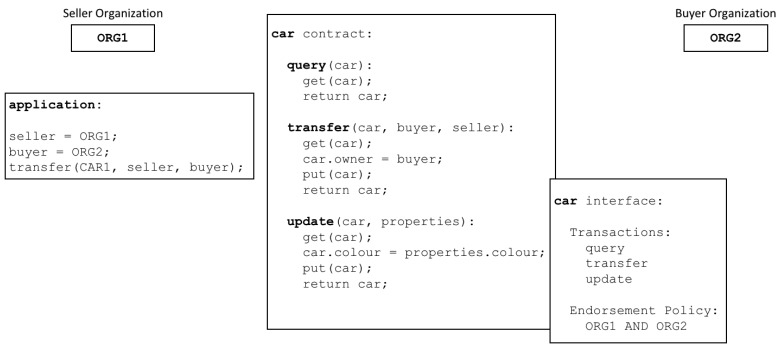



FIGURE A.23: Transfer of ownership of a car [6].

In this example an application wants to transfer ownership of a car from ORG1 to ORG2, which represent the seller and the buyer respectively. In order for the transaction to be approved, the policy says that both organizations must execute and sign the transaction

**Smart Contracts and policy.** For each chaincode a specific validation policy (endorsement policy) is defined, which applies to all the smart contracts present within it. This policy indicates which organizations within the blockchain network must sign the transactions generated by the execution of the contracts, in order for these transactions to be considered valid.
The presence of these policies makes Fabric suitable for modeling interactions as they occur in reality, where they must be validated by reliable organizations.

**Execution.** Each smart contract, after being installed on a node of the system, can be executed by the same.

During execution, the contract takes as input the parameters, contained in the transaction proposal, generated by an application and uses them, in combination with the logic of the program, to generate a response message (transaction proposal). response), containing both the status prior to the change, and the subsequent one, which precisely includes the changes to be made.

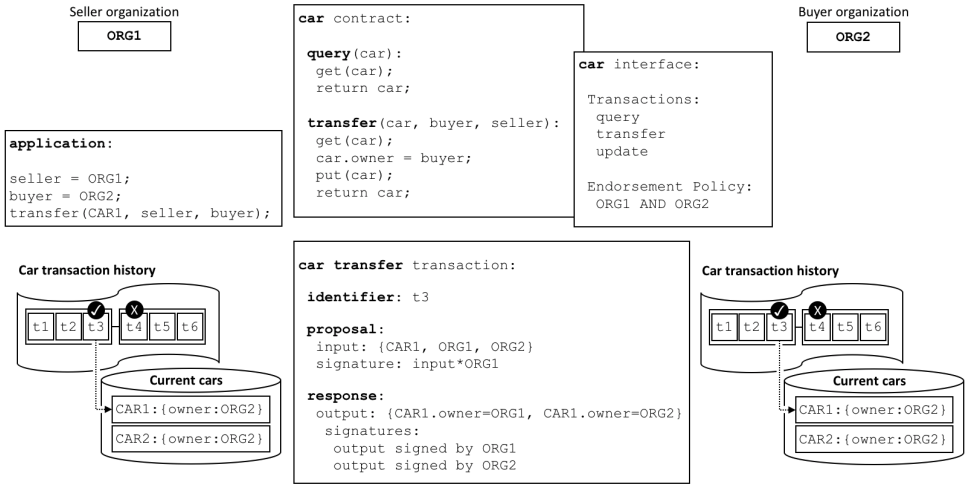



FIGURE A.24: Smart Contract execution [6].

Still considering the transfer of ownership of a car, it can be seen that there is an input and an output in the transaction, representing the transfer of ownership from ORG1 to ORG2. The output must be signed by both organizations.

**Chaincode Definition on a channel.** In order for nodes belonging to a channel to execute a smart contract, the chaincode that contains it must be defined on that channel.

The definition of a chaincode (chaincode definition) is a structure that contains the parameters that control how the chaincode operates on the considered channel. These parameters include the name, version, validation policy and must be approved by a sufficient number of organizations in the channel.

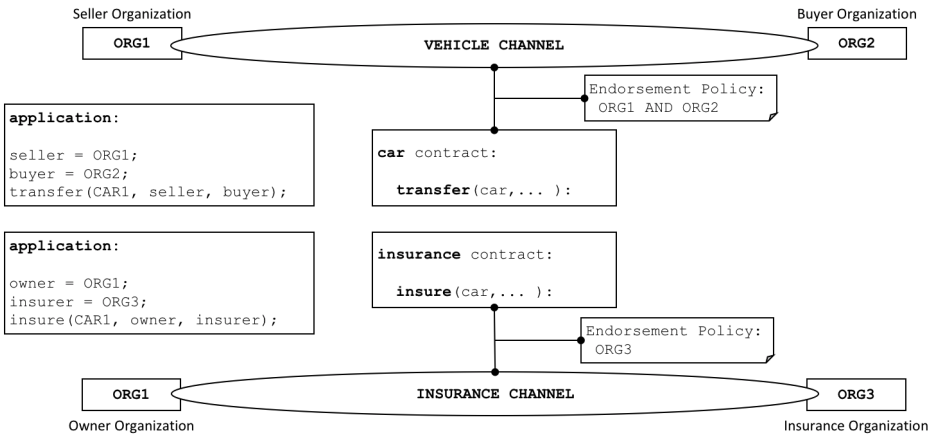



FIGURE A.25: Chaincode Definition on a channel [6].

In this example we define a car contract on the VEHICLE channel and an insurance contract on the INSURANCE channel. The first contract specifies a validation policy that requires ORG1 and ORG2 to sign transactions, while the second contract specifies only ORG3.

# Appendix B

# Blockchain-based Privacy Enforcement in the IoT domain

## B.1 Manufacturer Usage Description

The Manufacturer Usage Description (MUD) is an Internet Engineering Task Force (IETF) standard, which allows end devices to indicate to the network their communication needs. Let us consider, for instance, a smart light bulb. In general, this device does not require to interact with other smart home devices (such as heaters or coffee machines). The only connection needed is to the specific vendor service for remote control. As such, by properly configuring MUD, the light bulb owner can block any other unexpected communication. This allows to reduce the attack surface and block some malicious attempts to exploit the device for other purposes. The general MUD schema is as follows. The manufacturer defines, through a configuration file, the access policies and the type of network functionalities required by the device. The configuration can express that the light bulb has to communicate only on port 80 with https protocol to a specific server. When a new device joins a network, it communicates the URL where its MUD configuration file can be retrieved. Then, the network access device (NAD) retrieves the URL and sends it to the MUD manager. This component, usually placed in the local network, retrieves the MUD configuration file from the MUD file manager (aka, the smart light bulb vendor) and adapts the network setting such as to adhere to the MUD configuration (e.g., smart light bulb can connect only with its remote server). Although the standard currently covers only aspects related to network access control, the goal is to extend it to other fields, such as quality of service and data privacy. Indeed, the standard includes a MUD Extension[1] field designed at this purpose.

## B.2 Theorems and proofs

The security and completeness of the privacy enforcement mechanism is stated by Theorem 1, given below.

**Theorem 1** (Security and completeness PE)**.** Let us consider a data stream $S_d$, identified by $idS$, and a privacy preference $pp$ of data owner $DO$, which applies to $t_d \in S_d$ and contained in the privacy preference tuple $t_{pp}$. For a consumer $C$, identified by $idC$ and subscribed to $idS$ with a privacy policy $p$, the smart contract in Pseudocode 1 returns a chunk tuple $t_c$, containing $t_d$, where $tc.check.grant = true \Leftrightarrow p$ complies with $pp$.

[1]Reporting MUD behavior to vendors - Available at https://tools.ietf.org/html/draft-lear-opsawg-mud-reporter-00.

*Proof*. We first prove that $check.grant = true \Rightarrow p$ complies with $pp$. The privacy policy $p$ complies with privacy preference $pp$ when $p.up \in pp.\overrightarrow{ip} \wedge p.dataRet \leq pp.rt \wedge p.dataRel = pp.tpu$. Let us suppose that $p$ does not comply with $pp$. This means that one or more of the previous conditions are not met. Privacy enforcement is done by the smart contract in Pseudocode 1. Function $submitPrivacyPreference()$ (line 5) is invoked when a new privacy preference tuple $t_{pp}$ arrives to the blockchain. This function generates the privacy preference $pp$ which is stored on the blockchain with an identifier $idTpp$. Then, function $privacyComplianceChecker()$ (line 12) is invoked, passing as parameter the $idTpp$ of the previously created tuple. We assumed that the consumer $idC$ has subscribed to the data stream $idS$, therefore it is present inside the $consumerVector$ (line 16). Thus, when the for loop (line 18) is executed for consumer $idC$, its privacy policy tuple $t_p$, used for subscribing to $idS$, is returned. The privacy preference $pp$ and the privacy policy $p$, contained in $t_p$, are used as parameters for function $verifyAuth()$ (line 21). Since we assumed that $p$ does not comply with $pp$, the function returns $False$. Therefore, *privacyComplianceChecker()* (line 22) inserts a new entry in the **check** vector for consumer $idC$ with $grant = false$. This result is stored in the blockchain with identifier $idCheck$. When the Pseudocode 1 receives as input a tuple selection $\sigma$ containing $t_d$, function $dataTupleChunk()$ is invoked. Tuple selection $\sigma$ refers to an $idS$ (line 28) from which the function retrieves $idCheck$ from the blockchain (line 32). After that, a chunk tuple $t_c$ is created (line 34). The chunk tuple $t_c$ is a group of data tuples containing $t_d$, sharing the same privacy preference $pp$ and on which the compliance check $idCheck$ is applied. Therefore, for consumer $idC$, $check.grant = false$, which contradicts the hypothesis.

Let us now prove that, if $p$ complies with $pp \Rightarrow check.grant = true$. Let us suppose that $check.grant = false$. Then, function $privacyComplianceChecker()$ (line 12) is invoked with parameter $idTpp$, obtained, as in the previous case, from function $submitPrivacyPreference()$ (line 5). We assume that consumer $idC$ has subscribed to data stream $idS$. Therefore, he/she is present inside the $consumerVector$ (line 16). In the for loop (line 18), when consumer $idC$ is considered, its privacy policy tuple $t_p$ used for subscribing to $idS$ is retrieved. After that, the privacy preference $pp$ and the privacy policy $p$, contained in $t_p$, are used as parameters for function $verifyAuth()$ (line 21) that, by assumption, returns $grant = false$, which is added to the **check** data structure. Hence, **check** is saved on the blockchain with identifier $idCheck$. When the blockchain receives $t_d$, contained in a tuple selection $\sigma$, function $dataTupleChunk()$ is invoked. Here, $t_d$ is inserted into a chunk tuple $t_c$ (line 34) to which the $idCheck$ previously computed is associated. Therefore, for consumer $idC$, $check.grant = false$, that is, $p.up \notin pp.\overrightarrow{ip} \vee p.dataRet > pp.rt \vee p.dataRel \neq pp.tpu$. Therefore, $p$ is not compliant with $pp$, which contradicts the hypothesis.

Finally, the security and completeness of the data release process is stated by Theorem 2, given below.

**Theorem 2** (Security and Completeness DR)**.** Let $S_d$ be a data stream, identified by $idS$, and let $t_d \in S_d$ be a data tuple. Let $pp$ be a privacy preference of data owner $DO$ that applies to $t_d$. Let $t_c$ be the chunk tuple for $S_d$, identified by $idTc$, which contains $t_d$ and the privacy preference $pp$. A consumer $C$, identified by $idC$ and subscribed to $idS$ with a privacy policy $p$ that complies with $pp$, receives $t_d \Leftrightarrow t_c.check.grant = true$.

*Proof*. We first prove that if $C$ receives $t_d \Rightarrow check.grant = true$. Data release is done by the smart contract in Pseudocode 3. Let us suppose $check.grant = false$ for consumer $idC$. When function $dataRelease()$ (line 4) is invoked, it receives as input the chunk tuple identifier $idTc$ and the data vector **d**. We recall that $t_d$ is partly

contained into $t_c$ and partly in **d**. Firstly, the chunk tuple $t_c$ is retrieved from the blockchain (line 5), using *idTc*. Then, each entry of the check vector in $t_c$ is considered by a for loop (line 7). When, the entry related to consumer *idC* is checked, by assumption we find $check.grant = false$. This implieas that **d** is not released to consumer *idC*, so it cannot get $t_d$, and this contradicts the hypothesis.

Now, we prove that if $check.grant = true \Rightarrow C$ receives $t_d$. Let us suppose $C$ does not receive $t_d$. Let's consider again function $dataRelease()$ (line 4, Pseudocode 3), when invoked with parameters the chunk tuple identifier *idTc* and data vector **d**. This function retrieves from the chunk tuple identified by *idTc* the corrispondent check vector **check** (line 6). By assumption, consumer *idC* does not receive **d** This implies that the corresponding entry in **check** has $check.grant = false$ for *idC*, which contradicts the initial hypothesis.

The logics of previous theorems are valid only if executed by a smart contract which ensures the process cannot be altered. Thanks to the blockchain and consensus protocols, the execution of the smart contract is considered safe and trust. The smart contract is executed by different autonomous and independent parties, then the result is subject to a consensus process that guarantees its consistency between the different executions and immutably, being stored on the blockchain.

# Bibliography


[1] Karim Adam et al. "A privacy preference model for pervasive computing". In: *Proceedings of the First European Conference on Mobile Government*. 2005, pp. 10–12.

[2] Wasi Uddin Ahmad et al. *PolicyQA: A Reading Comprehension Dataset for Privacy Policies*. 2020. arXiv: 2010.02557 [cs.CL].

[3] Abduljaleel Al-Hasnawi, Steven M. Carr, and Ajay Gupta. "Fog-based local and remote policy enforcement for preserving data privacy in the Internet of Things". In: *Internet of Things* (2019).

[4] Mansoor Alblooshi, Khaled Salah, et al. "Blockchain-based ownership management for medical IoT (MIoT) devices". In: *Int. Conf. IIT*. 2018.

[5] Mohammad Amiri-Zarandi, Rozita A. Dara, and Evan Fraser. "A survey of machine learning-based solutions to protect privacy in the Internet of Things". In: *Computers & Security* (2020).

[6] Elli Androulaki, Yacov Manevich, et al. "Hyperledger fabric". In: *Proceedings of the Thirteenth EuroSys* (2018).

[7] Elli Androulaki et al. "Hyperledger fabric: a distributed operating system for permissioned blockchains". In: *Proceedings of the thirteenth EuroSys conference*. 2018, pp. 1–15.

[8] Elli Androulaki et al. "Hyperledger fabric: a distributed operating system for permissioned blockchains". In: *Proceedings of the Thirteenth EuroSys Conference*. ACM. 2018, p. 30.

[9] L. Bahri, B. Carminati, and E. Ferrari. "Privacy in web services transactions: a tale of more than a decade of work". In: *IEEE Transactions on Services Computing* PP.99 (2017), pp. 1–1. ISSN: 1939-1374. DOI: 10.1109/TSC.2017.2711019.

[10] Adam Barker and Jano Van Hemert. "Scientific workflow: a survey and research directions". In: *International Conference on Parallel Processing and Applied Mathematics*. Springer. 2007.

[11] Imran Bashir. "Mastering Blockchain-Distributed ledgers, decentralization and smart contracts explaine". In: Packt Publishing Ltd, 2017.

[12] Khalid Belhajjame et al. "Privacy-Preserving Data Analysis Workflows for eScience." In: *EDBT/ICDT Workshops*. Vol. 2322. 2019.

[13] Jorge Bernal Bernabe et al. "Privacy-preserving solutions for blockchain: Review and challenges". In: *IEEE Access* 7 (2019), pp. 164908–164940.

[14] Saliha Besik and Johann-Christoph Freytag. "Ontology-Based Privacy Compliance Checking for Clinical Workflows". In: Sept. 2019.

[15] Luca Bonomi, Yingxiang Huang, and Lucila Ohno-Machado. "Privacy challenges and research opportunities for genomic data sharing". In: *Nature genetics* 52.7 (2020), pp. 646–654.

[16] Sotirios Brotsis, Nicholas Kolokotronis, et al. "On the Security and Privacy of Hyperledger Fabric: Challenges and Open Issues". In: *2020 IEEE World Congress on Services (SERVICES)*. 2020.

[17] Ethan Buchman. "Tendermint: Byzantine fault tolerance in the age of blockchains". PhD thesis. 2016.

[18] Vitalik Buterin et al. "A next-generation smart contract and decentralized application platform". In: *white paper* (2014).

[19] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.

[20] *California Consumer Privacy Act (CCPA)*. URL: https://oag.ca.gov/privacy/ccpa.

[21] B. Carminati, C. Rondanini, and E. Ferrari. "Confidential Business Process Execution on Blockchain". In: *2018 IEEE International Conference on Web Services (ICWS)*. 2018, pp. 58–65. DOI: 10.1109/ICWS.2018.00015.

[22] Barbara Carminati, Elena Ferrari, and Christian Rondanini. "Blockchain as a Platform for Secure Inter-Organizational Business Processes". In: *2018 IEEE 4th International Conference on Collaboration and Internet Computing (CIC)*. 2018, pp. 122–129. DOI: 10.1109/CIC.2018.00027.

[23] Barbara Carminati, Elena Ferrari, and Christian Rondanini. "Blockchain as a platform for secure inter-organizational business processes". In: *2018 IEEE 4th International Conference on Collaboration and Internet Computing (CIC)*. IEEE. 2018, pp. 122–129.

[24] Barbara Carminati, Christian Rondanini, and Elena Ferrari. "Confidential business process execution on blockchain". In: *2018 ieee international conference on web services (icws)*. IEEE. 2018, pp. 58–65.

[25] Miguel Castro, Barbara Liskov, et al. "Practical byzantine fault tolerance". In: *OSDI*. Vol. 99. 1999. 1999, pp. 173–186.

[26] Wanghu Chen et al. "Blockchain based provenance sharing of scientific workflows". In: *2018 IEEE International Conference on Big Data (Big Data)*. IEEE. 2018, pp. 3814–3820.

[27] Olivia Choudhury et al. "A blockchain framework for managing and monitoring data in multi-site clinical trials". In: *arXiv preprint arXiv:1902.03975* (2019).

[28] Konstantinos Christidis and Michael Devetsikiotis. "Blockchains and Smart Contracts for the Internet of Things". In: *IEEE Xplore Digital Library* (2016).

[29] Raiane Coelho et al. "Blockflow: Trust in scientific provenance data". In: *Anais do XIII Brazilian e-Science Workshop*. SBC. 2019.

[30] Federico Daidone, Barbara Carminati, and Elena Ferrari. "Blockchain-based Privacy Enforcement in the IoT domain". In: *IEEE Transactions on Dependable and Secure Computing* (2021).

[31] Damiano Di Francesco Maesa, P Mori, and L Ricci. "A blockchain based approach for the definition of auditable Access Control systems". In: (2019).

[32] Damiano DI FRANCESCO MAESA, Paolo Mori, and LAURA EMILIA Ricci. "Blockchain based access control services". In: *IEEE Symposium on Recent Advances on Blockchain and its Applications, Canada*, 2018.

[33] A. Dorri et al. "Blockchain for IoT security and privacy: The case study of a smart home". In: *IEEE Int. Conf. PerCom Workshops*. 2017.

[34] Ashutosh Dwivedi et al. "A Decentralized Privacy-Preserving Healthcare Blockchain for IoT". In: *Sensors* (2019).

[35] EU. *General Data Protection Regulation - EUR-Lex 32016R0679*. 2016. URL: https://eur-lex.europa.eu/eli/reg/2016/679/oj.

[36] Dinuni Fernando et al. "SciBlock: A blockchain-based tamper-proof non-repudiable storage for scientific workflow provenance". In: *2019 IEEE 5th International Conference on Collaboration and Internet Computing (CIC)*. IEEE. 2019, pp. 81–90.

[37] Luciano García-Bañuelos et al. "Optimized execution of business processes on blockchain". In: *International Conference on Business Process Management*. Springer. 2017, pp. 130–146.

[38] Yolanda Gil et al. *Privacy enforcement in data analysis workflows*. 2007.

[39] Christian Gorenflo, Stephen Lee, et al. "Fastfabric: Scaling hyperledger fabric to 20,000 transactions per second". In: *IEEE ICBC*. 2019.

[40] Kristen Griggs, Olya Ossipova, et al. "Healthcare Blockchain System Using Smart Contracts for Secure Automated Remote Patient Monitoring". In: *Journal of Medical Systems* (2018).

[41] Hyperledger Fabric Working Group. *A Blockchain Platform for the Enterprise*. URL: https://hyperledger-fabric.readthedocs.io/en/latest/fabric_model.html.

[42] Hyperledger Fabric Working Group. *A Blockchain Platform for the Enterprise*. URL: https://hyperledger-fabric.readthedocs.io/en/latest/ledger/ledger.html.

[43] Hyperledger Fabric Working Group. *A Blockchain Platform for the Enterprise*. URL: https://hyperledger-fabric.readthedocs.io/en/latest/network/network.html.

[44] Hyperledger Fabric Working Group. *A Blockchain Platform for the Enterprise*. URL: https://hyperledger-fabric.readthedocs.io/en/latest/identity/identity.html.

[45] Hyperledger Fabric Working Group. *A Blockchain Platform for the Enterprise*. URL: https://hyperledger-fabric.readthedocs.io/en/latest/membership/membership.html.

[46] Hyperledger Fabric Working Group. *A Blockchain Platform for the Enterprise*. URL: https://hyperledger-fabric.readthedocs.io/en/latest/network/network.html#creating-a-channel-for-a-consortium.

[47] Hyperledger Fabric Working Group. *A Blockchain Platform for the Enterprise*. URL: https://hyperledger-fabric.readthedocs.io/en/latest/peers/peers.html.

[48] Hyperledger Fabric Working Group. *A Blockchain Platform for the Enterprise*. URL: https://hyperledger-fabric.readthedocs.io/en/latest/orderer/ordering_service.html#raft.

[49] Hyperledger Fabric Working Group. *A Blockchain Platform for the Enterprise*. URL: https://hyperledger-fabric.readthedocs.io/en/latest/smartcontract/smartcontract.html#terminology.

[50] Hyperledger Working Group. *Hyperledger Whitepaper*. URL: https://docs.google.com/document/d/1Z4M_qwILLRehPbVRUsJ3OF8Iir-gqS-ZYe7W-LE9gnE/edit#heading=h.m6iml6hqrnm2.

[51] *Guidelines 4/2019 on Article 25 Data Protection by Design and by Default*. URL: http://www.planalto.gov.br/ccivil_03/_ato2015-2018/2018/lei/l13709.htm.

[52] Md Islam et al. "Wearable Technology to Assist the Patients Infected with Novel Coronavirus (COVID-19)". In: *SN Computer Science* (2020).

[53] Faisal Jamil et al. "Towards a Remote Monitoring of Patient Vital Signs Based on IoT-Based Blockchain Integrity Management Platforms in Smart Hospitals". In: *Sensors* (2020).

[54] Mayssa Jemel and Ahmed Serhrouchni. "Decentralized access control mechanism with temporal dimension based on blockchain". In: *2017 IEEE 14th International Conference on e-Business Engineering (ICEBE)*. IEEE. 2017, pp. 177–182.

[55] Myong H Kang, Joon S Park, and Judith N Froscher. "Access control mechanisms for inter-organizational workflow". In: *Proceedings of the sixth ACM symposium on Access control models and technologies*. ACM. 2001, pp. 66–74.

[56] Alexia Dini Kounoudes and Georgia M. Kapitsaki. "A mapping of IoT user-centric privacy preserving approaches to the GDPR". In: *Internet of Things* (2020).

[57] Leslie Lamport. "Time, clocks, and the ordering of events in a distributed system". In: *Concurrency: the Works of Leslie Lamport*. 2019, pp. 179–196.

[58] *Lei Geral de Proteção de Dados (LGPD)*. URL: http://www.planalto.gov.br/ccivil_03/_ato2015-2018/2018/lei/l13709.htm.

[59] Xueping Liang et al. "Provchain: A blockchain-based data provenance architecture in cloud environment with enhanced privacy and availability". In: *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE. 2017, pp. 468–477.

[60] Orlenys López-Pintado et al. "Caterpillar: A Blockchain-Based Business Process Management System." In: *BPM (Demos)*. 2017.

[61] Faiza Loukil et al. "PATRIoT: A data sharing platform for IoT using a service-oriented approach based on Blockchain". In: *ICSOC*. 2020.

[62] Nancy A Lynch. *Distributed algorithms*. Elsevier, 1996.

[63] Damiano Di Francesco Maesa, Paolo Mori, and Laura Ricci. "Blockchain based access control". In: *IFIP international conference on distributed applications and interoperable systems*. Springer. 2017, pp. 206–220.

[64] Gábor Magyar. "Blockchain: Solving the privacy and research availability tradeoff for EHR data: A new disruptive technology in health data management". In: *IEEE 30th NC* (2017).

[65] Satoshi Nakamoto. "Bitcoin: A peer-to-peer electronic cash system". In: (2008).

[66] Szabo Nick. "Smart contracts". In: *Unpublished manuscript* (1994).

[67] Diego Ongaro and John Ousterhout. *Raft consensus algorithm*. 2015.

[68] Aafaf Ouaddah, Anas Abou Elkalam, and Abdellah Ait Ouahman. "FairAccess: a new Blockchain-based access control framework for the Internet of Things". In: *Security and Communication Networks* 9.18 (2016), pp. 5943–5964.

[69] Christoph Prybila et al. “Runtime verification for business processes utilizing the Bitcoin blockchain”. In: *Future Generation Computer Systems* (2017).

[70] Christoph Prybila et al. “Runtime verification for business processes utilizing the Bitcoin blockchain”. In: *Future Generation Computer Systems* (2020).

[71] Christian Rondanini et al. “Blockchain-based controlled information sharing in inter-organizational workflows”. In: *2020 IEEE International Conference on Services Computing (SCC)*. 2020, pp. 378–385. DOI: 10.1109/SCC49832.2020.00056.

[72] Sara Rouhani and Ralph Deters. “Blockchain based access control systems: State of the art and challenges”. In: *arXiv preprint arXiv:1908.08503* (2019).

[73] Gokhan Sagirlar, Barbara Carminati, and Elena Ferrari. “Decentralizing privacy enforcement for Internet of Things smart objects”. In: *Computer Networks* (2018).

[74] Odnan Ref Sanchez, Ilaria Torre, and Bart P Knijnenburg. “Semantic-based privacy settings negotiation and management”. In: *Future Generation Computer Systems* (2020).

[75] U. Satapathy, B. K. Mohanta, et al. “A Secure Framework for Communication in Internet of Things Application using Hyperledger based Blockchain”. In: *Int. Conf. ICCCNT*. 2019.

[76] Peter Schaar. “Privacy by design”. In: *Identity in the Information Society* (2010).

[77] Fred B Schneider. “Least privilege and more [computer security]”. In: *IEEE Security & Privacy* 1.5 (2003), pp. 55–59.

[78] Mohamed Seliem, Khalid Elgazzar, and Kasem Khalil. “Towards privacy preserving iot environments: a survey”. In: *Wireless Communications and Mobile Computing* 2018 (2018).

[79] Iago Sestrem Ochôa et al. “A cost analysis of implementing a blockchain architecture in a smart grid scenario using sidechains”. In: *Sensors* 20.3 (2020), p. 843.

[80] Tanmay Sinha et al. “Trends and research directions for privacy preserving approaches on the cloud”. In: *Proceedings of the 6th ACM India Computing Convention*. 2013, pp. 1–12.

[81] R. Soni and G. Kumar. “A Review on Blockchain Urgency in the Internet of Things in Healthcare”. In: *Int. Conf. ICISS*. 2019.

[82] Melanie Swan. *Blockchain: Blueprint for a new economy*. " O’Reilly Media, Inc.", 2015.

[83] Rajesh K Thiagarajan et al. “BPML: A process modeling language for dynamic business models”. In: *Proceedings Fourth IEEE International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems (WECWIS 2002)*. IEEE. 2002, pp. 222–224.

[84] William Tolone et al. “Access control in collaborative systems”. In: *ACM Computing Surveys (CSUR)* 37.1 (2005), pp. 29–41.

[85] Florian Tschorsch and Björn Scheuermann. “Bitcoin and beyond: A technical survey on decentralized digital currencies”. In: *IEEE Communications Surveys & Tutorials* 18.3 (2016), pp. 2084–2123.

[86] Jane O Umekwudo and Junho Shim. "Blockchain Technology for Mobile Applications Recommendation Systems". In: *The Journal of Society for e-Business Studies* 24.3 (2019), pp. 129–142.

[87] Guozhang Wang et al. "Building a replicated logging system with Apache Kafka". In: *Proceedings of the VLDB Endowment* (2015).

[88] Shangping Wang, Yinglong Zhang, and Yaling Zhang. "A blockchain-based framework for data sharing with fine-grained access control in decentralized storage systems". In: *IEEE Access* 6 (2018), pp. 38437–38450.

[89] Ingo Weber et al. "Untrusted business process monitoring and execution using blockchain". In: *International Conference on Business Process Management*. Springer. 2016, pp. 329–347.

[90] Hyperledger Architecture Working Group (WG). *An Introduction to Hyperledger*. URL: https://www.hyperledger.org/wp-content/uploads/2018/07/HL_Whitepaper_IntroductiontoHyperledger.pdf.

[91] Hyperledger Architecture Working Group (WG). *Hyperledger Architecture, Volume 1*. URL: https://www.hyperledger.org/wp-content/uploads/2017/08/Hyperledger_Arch_WG_Paper_1_Consensus.pdf.

[92] Hyperledger Architecture Working Group (WG). *Hyperledger Architecture, Volume 2*. URL: https://www.hyperledger.org/wp-content/uploads/2018/04/Hyperledger_Arch_WG_Paper_2_SmartContracts.pdf.

[93] Stephen A White. "Introduction to BPMN". In: *Ibm Cooperation* 2.0 (2004), p. 0.

[94] Shomir Wilson et al. "The creation and analysis of a website privacy policy corpus". In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2016, pp. 1330–1340.

[95] Gavin Wood. *Ethereum: A secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper 151 (2014)*. 2014.

[96] Xiao Liang Yu, Xiwei Xu, and Bin Liu. "EthDrive: A Peer-to-Peer Data Storage with Provenance." In: *CAiSE-Forum-DC*. 2017, pp. 25–32.

[97] Yan Zhu et al. "TBAC: transaction-based access control on blockchain for resource sharing with cryptographically decentralized authorization". In: *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 1. IEEE. 2018, pp. 535–544.