UCLouvain

École polytechnique de Louvain

AuditTrust: A Blockchain-based Secure Audit Trail for Data Sharing in Distributed Environment

Authors: **Hugo Lloreda Sanchez, Sophie Tysebaert**
Supervisors: **Annanda Rath, Étienne Rivière**
Readers: **Lionel Dricot, Michał Król**
Academic year 2021–2022
Master [120] in Computer Science

Acknowledgments

We would like to express our sincere gratitude to our supervisors, Étienne Rivière and Annanda Rath, for the continuous support of our thesis throughout this academic year. Their guidance helped us throughout the research and writing of this work, as well as the implementation of our prototype. We would also like to thank the rest of our thesis committee: Lionel Dricot and Michał Król, for taking the time to read and evaluate this work. Our last thanks go to our families and friends for their eternal support and comprehension.

Hugo Lloreda Sanchez and Sophie Tysebaert

Abstract

In recent years, there has been a significant push toward developing and finding trust-building technologies in distributed system environments, especially for sharing data between mutually mistrusting entities. One of the key factors is to ensure data integrity, that is to ensure that data being shared has not been manipulated. To this aim, the ability to trace and audit data usage from its creation is a key component of such trust-building technologies: by recording all data accesses and modifications in an access log, we have a history of any data processing such that we can easily detect when data has been tampered with and who is behind this manipulation. However, for audit trail data to be useful, it must be correct and immutable. Moreover, mechanisms are also needed to enforce disclosure controls: data being shared must not be disclosed to unauthorized individuals. In this work, we present AuditTrust, a blockchain-based secure audit trail for data sharing in a distributed environment. AuditTrust is a prototype dedicated for smart traffic applications, whose main objectives are: 1° allowing secure data sharing between parties that do not trust each other by providing them a reliable audit tool and access control mechanisms; 2° being scalable and 3° minimizing costs imputed by the use of blockchain technology. To this aim, it combines several technologies together, such as Hyperledger Besu, IPFS, Intel SGX and Vault. Our evaluation shows that our system works correctly, even under pressure.

Contents

Chapter 1

Introduction

In the last two decades, the volume and speed with which data is generated have changed. The expansion of Web traffic and online stores contributed to the emergence of big data analytics, collecting search logs, click-rates or IP addresses geolocation for analyzing user behavior. This leads to a massive increase in collected heterogeneous data. Meanwhile, the speed and capability of computing systems constantly increase over time, such that digital data storage becomes more cost-effective and more convenient than storing information on paper [1]. This context has enabled the emergence of numerous systems and applications relying on the generation, storage and processing of an abundance of data. At the same time, cloud storage has become an important business model, providing both individuals and enterprises with different kinds of data storage services, letting them access and share this data anytime, everywhere and with everyone.

However, this abundance of data rises security and privacy issues: profiling methods based on linked records can reveal unexpected details about users' identity and private life [2]. Several privacy frameworks have been proposed to achieve a balance between data utility and user rights to privacy, such as the recent General Data Protection Regulation (GDPR) approved in 2016 in Europe. In a nutshell, such privacy framework gives the rights to a data subject to track who has accessed his/her data, how his/her data was used and to verify if his/her data was transferred without violating his/her consent. This is often referred to as the context of privacy [3]: when the information being shared with people or organizations within a particular community for specific purposes is exposed in another unintended context, we may feel a privacy violation.

These regulations require individuals and enterprises to take control of their data and to track precisely how these are processed. In this context, a very common mechanism is to record all data accesses and modifications in a log, called access

log, so that actions can be audited later [4]. However, the use of third-party cloud storage services has resulted in isolated (and centralized) data silos, where users (both individuals and enterprises) have limited control over their data and how it is used: access logs are generated by these services, such that users have to trust the cloud and application providers about the integrity of those access logs.

Besides the regulatory and privacy-compliant aspect, an access log is also useful for improving the security of a system and detecting any malicious activity: by recording all data accesses and modifications in an access log, we have a history of any data processing such that we can easily detect when data has been tampered with and who is behind this manipulation. However, these logs are traditionally kept in databases and utilize such client-server model of communication and data exchange [5]. Due to this architecture, access logs are vulnerable to a single point of trust: malevolent entities may access the database and manipulate critical audit information. In this case, organizations and individuals need to trust again the cloud provider regarding the security and integrity of these access logs.

These are the reasons why there has been a significant push toward developing and finding trust-building technologies in distributed system environments, especially for sharing data between mistrusting entities [3] [6] [7] [5] [8]. An important aspect of such technologies is to enable transparency on data accountability and provenance tracking without disclosing users' private information. In this context, blockchain technology has gained significant interest with applications spanning cryptocurrencies [9] [10], healthcare [7] [11] and IoT [12] to cite a few. Blockchains enable secure, transparent and immutable record keeping in distributed system environments, without the *need* of trusting a third party. Blockchains are append-only databases with decentralized management, whose content is replicated over the set of (blockchain) peers, providing them with a consistent and transparent view of the system. Applied to a data usage control application, a malicious party would require to change all the access logs kept at each peer if he/she wants to hack the system. Cost and complexity of such an attack are increased, which, in turn, eventually increases the defense capacity of the data usage control application.

1.1 Objectives

This master thesis topic is offered as a collaboration between the Belgian university UCLouvain and the Belgian industry-driven knowledge center Sirris, for ICity FEDER project in Brussels [13]. The purpose of this thesis is to design and prototype a secure blockchain-based data usage audit trail system for a smart traffic application in a distributed environment, with no trust relationships implied

and blockchain reduced costs (if possible). This system should preserve both data integrity and data confidentiality. To achieve this, we seek to implement with tools and technologies adapted for the predefined use cases the different layers of a blockchain-based distributed application, combined with more specific components, such as the use of a distributed file system and a trusted execution environment. To the best of our knowledge, no existing solution does match exactly the requirements related to our use cases, which motivated our choice to work on it.

1.2 Solution

AuditTrust aims to be a self-sufficient prototype of a more complete distributed application used by organizations that need to share data with each other and audit any access and modification made to this data being shared. To this aim, AuditTrust combines several technologies and mechanisms together:

- Blockchain technology: As the underlying technology of decentralized cryptocurrencies, blockchain technology has gained more and more attention in recent years. Blockchain refers to a distributed digital ledger in which information is recorded in data structures called blocks, which are chained together (to form the blockchain) by using a cryptographic hash function;

- Decentralized storage system: Due to several limitations involved by the use of blockchain technology, we cannot record any smart traffic-related data on the blockchain. To address these limitations, AuditTrust is using blockchain-external resources that aim to preserve the key properties introduced by the blockchain technology, such as immutability and availability;

- Role-based access control (RBAC) security policies: For restricting access of shared data to authorized users, AuditTrust is based on role-based access control policies. These assign users to roles and permissions to roles so that users acquire permissions on some data linked to these policies by being members of roles. With this mechanism, a user cannot manipulate data arbitrarily, but only in a constrained way, according to the role he/she is assigned;

- Trusted execution environment (TEE): TEEs are hardware protection mechanisms aimed to guarantee confidential and integral code execution. A TEE is implemented as a secure part of the processor for protecting a dedicated address space from unauthorized access. This mechanism is used in our solution after the user has been authenticated for provisioning encryption/decryption keys securely;

- A tool for secrets management: As our solution must be secure, i.e. as it must guarantee availability, confidentiality and integrity of data, it relies heavily on symmetric and asymmetric cryptography. However, the cryptographic keys generated must be also stored securely to guarantee the overall security of our system

1.3 Structure of the Thesis

This document is organized as follows: we will expose in chapter 2 the problem statement and the collection of the requirements, as a result of the analysis made at Sirris. We will also explain why this topic is of interest. We will provide in chapter 3 background information about the essential concepts that will be discussed throughout this work. Chapter 4 provides an overview of our solution, before describing it more precisely in chapter 5. Chapter 6 provides an evaluation of our system, by measuring its scalability. Chapter 7 compares our solution to the state of the art related to similar use cases. Lastly, we will conclude this master thesis with a discussion of our solution and its possible improvements in chapter 8.

Chapter 2

Problem Statement

AuditTrust aims to solve the problem of building trust in data, especially for data shared between mutually mistrusting entities in a distributed system environment. The current chapter will describe more precisely the problem that our solution should address, based on the requirements collection made at Sirris. Section 2.1 will illustrate it by a use case scenario, whose key ideas are summarized in section 2.2. Then, we will go into the analysis of all its aspects and requirements: section 2.3 describes its actors and the trust relationships they have with each other, while section 2.4 describes the adversary model implied by such a problem. In section 2.5, we will explain how users should be authorized (or unauthorized) to access and modify data within the system. Then, section 2.6 will describe the types of data the users are interacting with, as well as the operations they may execute; section 2.7 will focus on the core functionality of the solution: the audit tool. We will then list in section 2.8 the requirements involved in our problem, before explaining why this problem is of interest, by emphasizing its specificities in section 2.9.

2.1 Motivating Scenario

Several organizations A, B, C and D, referring to both public and private sectors (e.g. security authority, such as police department or municipality), can access traffic data managed by a traffic monitoring company in a city, for example, Brussels (see figure 2.1). The access to such data is governed by a mutually signed contract between one of these organizations and the company offering the traffic monitoring service. Each organization can retrieve images and/or videos shared by this traffic monitoring company, but with a strict access control rule stipulating the purpose of data usage and how data can be processed. The purpose of data access must also be legitimate and allowed by law, for instance, an authority may need to access data (videos or images) from a specific video camera for criminal investigation or

Figure 2.1: Schema of the motivating scenario described in 2.1.

simply traffic incident investigation.

Once shared, these data can be processed by those who have access to them at their destination system. This means that the data source system loses direct control over data. Without proper control of data access and usage at the destination system, data can be easily shared with a third-party system. This can lead to a data breach with serious legal consequences, as it violates the GDPR. For example, such data breach could leak a car's license plate or the identity of people walking in public space.

In this use case, we assume that both data owner and data consumer are untrusted entities, where a data owner refers to an entity getting data from the traffic monitoring company for sharing it with another entity, called data consumer, (authorized) to process such data. This data processing should be transparent to both. Access and usage of data should be auditable in a secure and trustworthy way. To this end, there is a need to have an audit tool able to trace any access to the data they have shared in a trusted and transparent manner.

Each shared data is subject to a specific rule according to which access rights are granted. This rule holds for a bounded amount of time and defines the role or category of users (representing the data consumer's business function) who are authorized to access the data. Thus, the same data can be made available for different organizations, but with different levels of access rights. Data owners might not want to provide the same access to everyone and to every piece of data: for example, A only wants that the managers of B may access and may process the videos that were taken this month, while the lawyers of B can only view the images taken by one A's camera at a specific location within one week. However, this rule can be updated by a data owner, who might decide to change the time validity of the data sharing or the category of users to which the rule was initially assigned.

Furthermore, a data owner does not want that a (malevolent) participant 1° benefits from accessing this data without being authorized and 2° can access the data without being correctly noticed by the audit tool. The first case would mean an access violation and the second case refers to the manipulation of the access logs used for auditing purposes. For instance, in this latter case, D may have had legitimate or illegitimate access to a C's file without this access being recorded by the audit tool, or D could have different access to a C's file (or to another C's file) than the one claimed in the audit tool.

As A, B, C and D are both independent and suspicious of each other, this audit tool cannot be managed by one single entity and the application itself implementing this tool cannot be trusted. Its reliability should be controlled transparently by every organization, rather than relying on the work of a centralized organization handling this tool. Only in this way trust can be built amongst the participants. This decentralized management of the audit tool has also another advantage: other organizations can easily join the system because any possible trust relations are captured by this decentralized system, as it assumes no trust at all. Thus, a decentralized solution is also easier to scale up.

Lastly, all public information needed for the setting up of such a solution is already exchanged beforehand by the different organizations, via a secure communication channel. This includes resources identifying publicly users or an organization.

2.2 Summary of the Motivating Scenario

Based on the scenario described in the previous section, our goal in this project was to build trust in data shared between mutually mistrusting entities (or participants) in a distributed environment, where the data being shared are videos or images

coming from smart traffic applications. In this use case, building trust in data means that 1° any user cannot access them, but only the authorized ones, according to the rule defined by the data owner, that holds only for a certain duration and for a given role (representing a data consumer's business function) and 2° the access logs on which the audit tool relies cannot be manipulated. This latter one is a key component of the system to be designed, as it allows the different parties to verify the access and usage of data. Since no trust is assumed between these parties, the audit tool cannot be managed by a single entity, but rather by the different parties themselves, in a trusted and transparent manner that a distributed environment (in presence of mistrusting actors) can bring.

2.3 Actors and Trust Model

In the context of this project, we consider 2 kinds of actors:

1. Data consumers (DCs): Entities who may process data;
2. Data owners (DOs): Entities that have the governance on some data being shared with DCs. DOs define the rules according to which access to their data is granted or denied

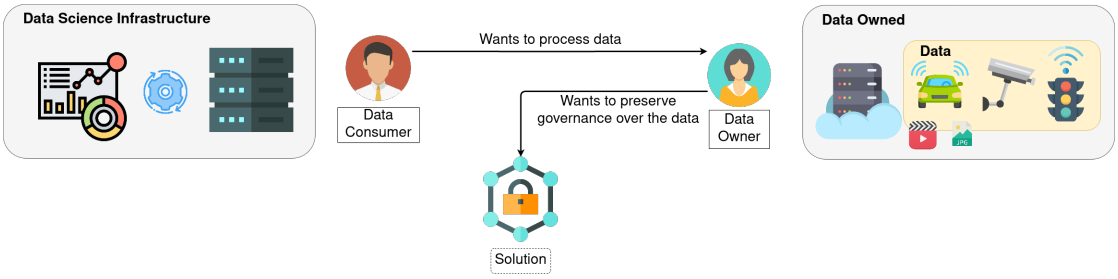

Figure 2.2: Types of actors in AuditTrust and their responsibilities.

Both data owners (DOs) and data consumers (DCs) refer indistinctly to organizations and individuals[1].

There are no trust relationships between DOs and DCs: DOs may suspect DCs to access and process unauthorized data or to tamper with the access logs. DCs may also suspect DOs to modify the logs. However, they assume that DOs will not provide meaningless or falsified shared data (i.e. images or videos) intentionally. DOs

[1]In the rest of this work, we will clarify the meaning given to these terms in case of ambiguity or if the distinction matters.

are also supposed to provide any (cryptographic) resource required for accessing the data being shared.

Besides this aspect, inside an organization, we assume an inherent trust relationship between its users, as being part precisely of the same organization.

2.4 Threat Model

In the context of this project, we assume that all entities act based on self-interest and may not follow the protocol. As described in the use case presented in sections 2.1 and 2.2, 2 major attacks from which the system should prevent are possible:

- An adversary might try to access data that was not disclosed to him/her by impersonating another (legitimate) user to which the data was shared;
- A malevolent user might try to manipulate the (supposedly immutable) access logs used for auditing purposes, for example by trying to compromise each computer system on which the audit tool is available

To perform such attacks, we presuppose that an adversary can also try to reverse engineer the client application to which he/she will connect for interacting with the system, or he/she might compromise the memory of the host machine. Therefore, the client application nor the host machine are trusted.

Besides these major threats, there exist other vectors of attacks, but we assume that they do not exist in our use case, to maintain a reasonable scope for this project. First, although data availability is a desirable security property that the system to be designed should have, this is not a strict requirement implied by the described use case. Therefore, denial-of-service attacks, by which an attacker tries to limit access to other users, are not considered in the prototype to be designed (we also assume that the networks have basic security safeguards, such as firewalls). Furthermore, the prototype does not prevent data leakage (for example via a watermarking mechanism [14], which stores a unique identifier per DC in the shared data before giving him/her access to this data). However, this functionality could be implemented in a future version of the system. Similarly, we do not consider the case of an authorized DC user sharing data to which he/she has access to others via external resources (e.g. by recording his/her screen when a video is launched), nor the case of a leak of private cryptographic resources (private keys) between 2 users (a mechanism for identifying the leak of provisioned keys can also be implemented in a future version of the prototype). Additionally, accessing physically each machine on which the system is running is also out of the scope of this thesis. We also

assume that all technologies used within this project (e.g. blockchain technology, cryptography primitives, etc.) are free of vulnerability. Lastly, we also assume for the scope of this thesis that organizations have exchanged beforehand all the (public) required information needed for setting up the application and that this information is meaningful as well.

2.5 Access Control Policies

As stated above, the access to shared data is constrained to *authorized* users by a rule defining which requests should be granted or denied. This rule is applied to a specific data object, for a certain amount of time (defined by a start date and time and ended with an expiration date and time) and for some users only, who are assigned to the role for which this regulation holds. In this case, a user is considered malicious if he/she accesses data that was not shared with any role to which he/she is assigned; inversely, a legitimate user is the one who accesses data following the rule.

In an information management system, the definition of such (high-level) rule or regulation according to which access is to be controlled is called a security policy [15]. This latter one enforces an access control decision, that determines whether a request for resources and data should be granted or denied [15]. Taken together, these concepts define access control policies. In our case, these access control

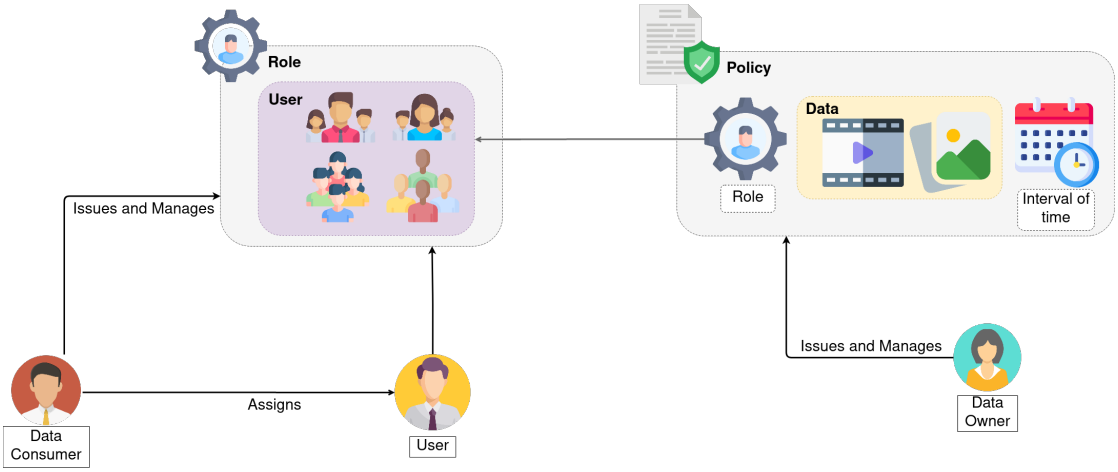

Figure 2.3: Access control policies based on roles.

policies are based on the categories of DC users (or roles): DC organizations assign their users to roles, depending on their business functions (i.e. Alice has the role

of "police inspector") and then communicate these roles within the system with the DOs. These latter ones can thus define the conditions to which a DC user can access their data (see figure 2.3).

This mechanism follows typically the specifications of a role-based access control (RBAC) mechanism, as defined in the literature [15] [16] [17]. The RBAC reference model defines sets of basic elements: users, roles, permissions, operations and objects. It proceeds as follows: users are assigned to roles, permissions for doing some operations on a (resource) object are assigned to roles and users acquire permissions by being members of roles. The same user can be assigned to many roles and a single role can have many users: for example, Alice can have the roles "police inspector" and "employee", but "employee" can also contain Bob, another DC user.

The access control policies of our system are also bounded in time. In total, these contain the following information:

- a DO identifier (from which this security policy was issued);
- a data identifier (that allows authorized users to retrieve the shared data);
- a role identifier (defining which participants should be authorized to access the shared data - where the assignment of participants to this role is performed by a DC -);
- a start date and time (from which an access control policy becomes active);
- an expiration date and time (from which an access control policy becomes inactive);
- the permission and operation a DC user may do with the shared data (data processing)

These policies can also be updated by his/her issuer (a DO) for 3 attributes only: the start date and time, the expiration date and time and the permission given. Similarly, roles can also evolve over time: a DC can change the name of a role, add other users to these roles or revoke them. This respect of privacy and this flexibility are the advantages of RBAC mechanism over other types of access controls [16]: a user gets access rights only via the roles he/she is assigned to, such that his/her identity is not disclosed and the management of access controls is easier, as it is only sufficient to add a user to a role or remove him/her from a role for respectively granting and denying his/her access to a specified data object.

2.6 Data and Data Processing

As shown in the motivating scenario (section 2.1), shared data are files coming from smart traffic applications. These files are either videos or images. No specification on their sizes was given, but we can assume that these are respectively short sequences of filmed traffic (of 1, 5 or 10 minutes) and images issued by cameras in the public space, whose quality is similar to the one we can expect in standard dashcams.

Regarding the data processing, namely the set of operations an authorized user is allowed to perform on a given file, it is restricted in our use case to a read-only operation because of time constraints. DCs may only view the shared data. However, a lot of requests (around 1,000 requests/second) for data processing are expected to appear at the same time.

2.7 Audit Tool and Access Logs

The core functionality of the solution is to provide a reliable and user-friendly audit trail system, that allows to build trust between mistrusting entities (DCs and DOs), by providing them with an audit tool. This latter one is an application for verifying whether there has been suspicious activity or not inside the system, where malicious actions are the ones defined in section 2.4. This audit tool also allows to audit a shared data within a given time interval, i.e. it should list all accesses that have been made to this shared data during the given time interval.

For achieving these goals, this audit tool relies on access logs. These are common security mechanisms for enhancing data integrity protection [4]: access logs are files recording all data modifications. In our case, they contain the answers to the following questions:

- when the access to a shared data was requested?
- by whom?[2]
- for accessing which shared data?
- with which permission?
- is this access intent valid?

[2]The information given here must be a unique identifier, but it can be anonymized, e.g. a long sequence of characters and numbers.

2.8 Requirements

We list in the current section the functional and non-functional requirements discussed in the previous sections that our solution must satisfy.

2.8.1 Functional Requirements

The system must be able to cover the following aspects regarding its functionalities:

- The solution provides an application for verifying whether a suspicious activity (as defined in 2.4) has occurred;
- It allows to audit the data usage of a file for a given time interval;
- Another functionality is the data sharing by a DO user, which is defined by access controls policies that can evolve over time;
- It also allows the definition of roles that can be linked to policies; users can be assigned or revoked to these roles;
- A DC user can request access to a shared data, provided he/she is assigned to the relevant role

2.8.2 Non-Functional Requirements

The system must be able to provide the following guarantees:

- No trust relationships are assumed, except the ones between members of a same organizations;
- It preserves both data confidentiality (i.e. information is not disclosed to unauthorized parties, by the means of cryptography or RBAC mechanism for example) and the integrity of the access logs (in this context, these are immutable);
- The system is user-friendly/easy to use;
- It can handle a large number of requests (or accesses) per second, for example, 1,000 accesses/s;
- Its financial cost is reduced

2.8.3 Architecture and Infrastructure Requirements

The use case described precedently emphasizes the distributed architecture of the solution: each involved party should possess the audit tool so that it cannot be managed by one single entity. This is a consequence of a (strong) non-functional requirement of our system: no trust relationships (at all) are assumed, and even the host system, as well the application to which DOs and DCs users are connected to use AuditTrust's functionalities, are not trusted. A malevolent participant might try to retrieve some sensitive information during its computation on the host machine.

Regarding the infrastructure, there is no constraint, except that it should be deployed on regular machines (i.e. not high costly servers).

2.9 Motivation

Before concluding this chapter, we would like to highlight that the problem described here is not a trivial one, as it combines a lot of constraints making it unique and original. First, there is absolutely no trust between the actors (DOs, DCs); we cannot assume *any* trust relationships at all. Consequently, we cannot have any component relying on a centralized entity or a consortium of organizations, as they are also suspicious. Second, the solution should be secure, namely, it should prevent from 1° unauthorized access to shared data and 2° any modification of the access logs. Third, with our use case, it is very likely that cryptography will be used, meaning that a DO's secrets giving access to his/her data have to be shared with a DC. However, the host system itself cannot be trusted, nor the client application: a malevolent participant could try to reverse engineer this latter one and retrieve the keys used for encryption, decryption and for signing. Thus, the solution to implement should also take this into account. Fourth, the solution should be scalable, being able to handle a large number of accesses per second. Lastly, the financial cost of such a solution should be reduced if possible.

To the best of our knowledge, no existing solution does match exactly the requirements related to our use cases, although there exist already some solutions in the literature that address a similar problem to ours. With PrivacyGuard [3], Xiao et al. implement mechanisms for preserving data integrity and data confidentiality without referring to a trusted third party. Their system also takes into account more attacks than ours. However, they do not implement an audit tool as described in 2.7 and their solution is costly from a financial perspective. Liang et al. propose in [6] a decentralized solution to collect and verify the history of all modifications

made on data, but it assumes some trust relationships, as it relies on an auditor designated by the data provider for verifying data modifications. Moreover, the scope of the implementation of their solution is limited. Miyachi and Mackey also consider under the term "hOCBS" 3 different models of decentralized solutions for healthcare applications, depending on the nature of the data, their purposes and the regulations to which they are subject [7]. However, the authors do not have the same trust issue as ours and therefore they can rely on some entities acting as authorities. Wang et al. also propose a solution for sharing data in a decentralized manner, with access control mechanisms as well as encryption ones for preserving data confidentiality [8]. However, this solution does not implement an audit tool and is not flexible, due to the use of another access control mechanism than RBAC. These several solutions will be discussed in chapter 7.

Chapter 3

Background Information

This chapter provides an overview of the main technologies and mechanisms that are going to be discussed in the rest of this work. Section 3.1 provides an introduction to cryptographic mechanisms that will be included in our solution; section 3.2 presents briefly key problems of distributed systems; section 3.3 gives an overview of the blockchain technology and its core components, while section 3.4 gives an introduction about trusted execution environment (TEE) and more specifically, Intel SGX.

3.1 Cryptography

According to Katz and Lindell, (modern) cryptography is "the scientific study of techniques for securing digital information, transactions and distributed computations" [18]. The following subsections will talk about major cryptography mechanisms that will be used within our project: encryption (subsection 3.1.1), authentication (subsection 3.1.2) and cryptographic hash functions (subsection 3.1.3).

3.1.1 Encryption

Historically, the first purpose of the use of cryptography was to permit secret communication by encoding information. This process refers to any conversion of the original representation of the information (known as plaintext) into another form of representation (known as ciphertext), which can be deciphered by *some* parties back to plaintext, to access the original information (not encoded). When these parties share in advance some secret information, they are in a symmetric-key setting; when no secret information is exchanged in advance, parties are in a setting of asymmetric encryption.

Symmetric-key setting (symmetric cryptography) The secret information exchanged in advance is called a key [18]. Parties use this key for communicating secretly with each other, namely for exchanging information without this latter one being disclosed to someone else than the receiver (e.g. an eavesdropper): the sender uses the key to encrypt his/her message (plaintext) before sending it, while the receiver uses the same key for decrypting the message (ciphertext) and recover the original message. As the same key is used for converting the plaintext into ciphertext and the ciphertext into plaintext, this setting is known as the symmetric-key setting. Sender and receiver hold the same key for both encryption and decryption. This implies that parties must find a way to exchange this key secretly in advance, which is chosen uniformly at random from a set of all possible keys, output by a key-generation algorithm.

Asymmetric encryption or public-key cryptography (asymmetric cryptography) In contrast to the previous situation, asymmetric encryption does not share beforehand secret information. Furthermore, the sender and receiver use different keys for encryption and decryption. In this case, each participant generates a public/private key pair (via a key-generation algorithm) and distributes the public key of this pair to the other participant. Then, the encryption proceeds as follows: a message m is encrypted with the receiver's public key before being sent. The receiver can then decrypt the received ciphertext c with his/her private key (related to the aforementioned public key) and retrieve m. Public and private keys generated together are mathematically related to each other. However, it is infeasible to retrieve a private key from its related public key [19].

It results from this the following basic correctness requirement of encryption: decrypting a ciphertext (with the appropriate key) gives the original message that was encrypted. Without an adequate key for decrypting data, the simplest method to read a ciphertext is to perform a brute-force attack up to the maximum length of this key [18] [20], hence the importance of using a long key length to make such attack computationally infeasible (i.e. it would take too many time and computing resources).

3.1.2 Authentication

Encryption can be used to prevent an adversary from learning anything about the content of messages sent over a communication channel. However, some security concerns are also related to the message integrity: is the message received the same as the message sent by the other party? To do so, a user will prove his/her identity before sending a message. If the receiver cannot retrieve proof of his/her identity

upon reception of this message, this means that this message was modified by a (malicious) third-party [18].

Digital signature (asymmetric cryptography) The public-key counterpart for authenticating a user is called a digital signature. A sender S who has established a private/public key pair (via a key-generation algorithm) signs a message m with his/her private key such that every participant having the public key of S can verify, via S's public key, if the message was originated from S and not modified.

3.1.3 Cryptographic Hash Functions

The last main cryptographic mechanism we will (briefly) present refers to the cryptographic hash functions. Hash functions are functions that take arbitrary-length input strings for compress them into shorter strings of fixed length: for a hash function H, we have $H : \{0,1\}^* \rightarrow \{0,1\}^n$. A cryptographic hash function is a hash function with some additional properties [21]:

- One-way function: Given a hash value y, it is computationally infeasible to find m such that $H(m) = y$;
- Collision resistance: It is infeasible to find 2 messages m_1 and m_2 such that $H(m_1) = H(m_2)$;
- Random oracle property: $H(m)$ is indistinguishable from a random value

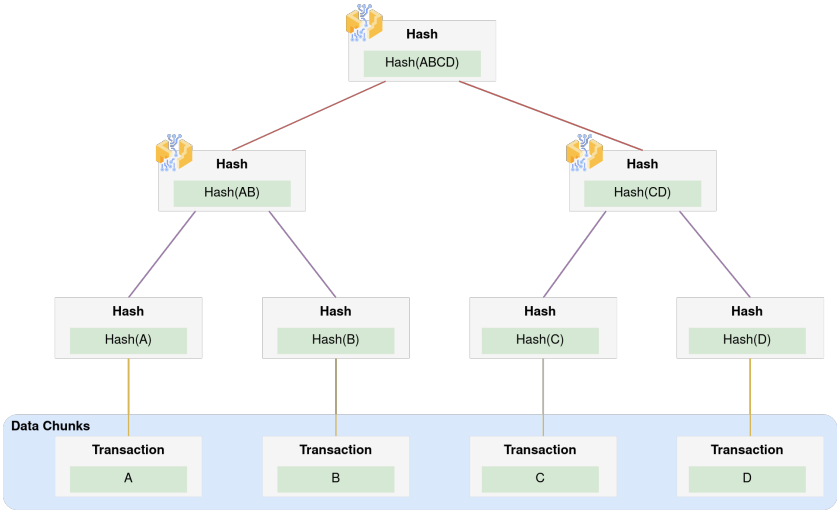

Figure 3.1: An example of Merkle tree.

Merkle tree A data structure, called Merkle tree, can also be used for verifying the integrity of any data stored. It refers to a tree where each node contains a hash value output by a cryptographic hash function, which takes as input the combination of the hashes of its child nodes. Figure 3.1 shows an example of Merkle tree.

3.2 Distributed Systems

A distributed system is a collection of *autonomous* computing elements called nodes, connected by a network, which appear to its users as a single coherent system [22]. In such systems, hardware or software components, located at different networked computers (potentially separated spatially by a large distance) communicate and coordinate their actions only by passing messages [23]. This definition of distributed systems constraints the topology of the network formed by these nodes: as each node communicates directly with each other, spreading and propagating messages across the network, this latter one can only be decentralized. This contrasts with traditional network topologies, which are more centralized, where only a few nodes have full control over the communication of messages (see figure 3.2).

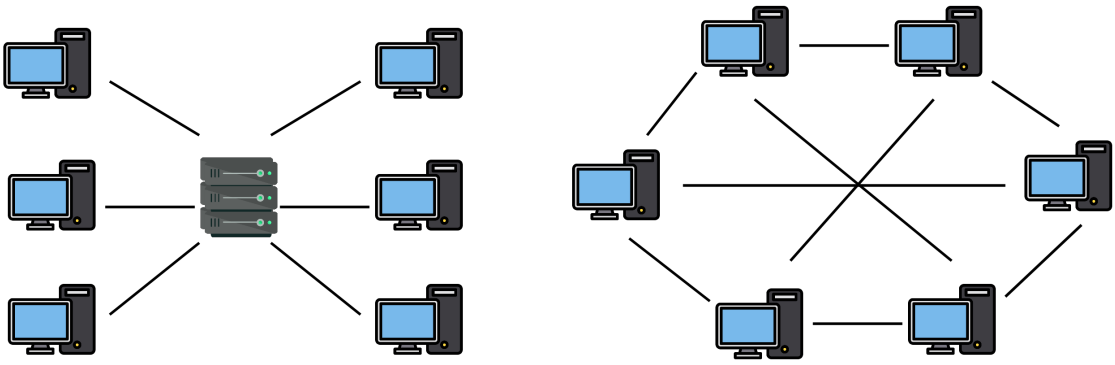

Figure 3.2: Comparison of a centralized (star) network (on the left) with a decentralized one (on the right).

However, this kind of system brings also new challenges: in presence of multiple nodes, we need not only to achieve some form of cooperation among these nodes but also to keep a consistent state of the system, even if a subset of these nodes have failed or have become disconnected [22]. Subsection 3.2.1 will discuss briefly the cooperation problem in such systems, while subsection 3.2.2 discusses their consistency, despite of failures.

3.2.1 Consensus

Agreeing on the same value is a fundamental problem in distributed environments. For instance, if there is a node that controls a motorized valve that receives both m_1 and m_2, how this node can decide if the valve should be opened (as stated by m_1) or closed (as stated by m_2), especially in presence of faulty nodes inside the network? This problem is described as a consensus problem [22]. Therefore, a distributed system must implement an algorithm that ensures that:

- all correct nodes eventually decide (termination property);
- every node decides the same (agreement property);
- nodes only decide on proposed values (integrity property)

A common approach for reaching consensus is for all nodes to agree on a majority value, where this majority can only be obtained if at least 51% of the participating nodes agree on this value [22] [24]. However, one or more nodes could crash, be disconnected or behave arbitrarily, such that a consensus protocol must also take into account these types of failures, by being fault-tolerant or resilient. We will see now what it means.

3.2.2 Consistency and Faults

Lamport et al. were one of the first authors to study the reliability of a computer system despite the failure of one or more of its components, by the so-called "Byzantine Generals problem" [25]. The question asked by this abstract problem is the following one: how to be sure that multiple entities (with some malicious ones) agree on a common value before an action is taken? Simple mechanisms for detecting failures can be implemented, such as sending heartbeat messages regularly or reaching a consensus via a majority vote (where each node has a vote that it can send within a given time interval). Any distributed system able to achieve this common agreement even in presence of malicious nodes is called Byzantine fault-tolerant (BFT), as the name of Lamport et al.'s problem. This property allows a distributed system to encounter the largest set of node failures (see figure 3.3). However, it has a cost: BFT algorithms only tolerate up to 1/3 faulty nodes, while non-BFT algorithms can tolerate up to 1/2 faulty nodes [22].

3.3 Blockchain Technology

Blockchain technology has recently gained extensive interest from both academia and industry. It was pioneered by Bitcoin in 2009 [26] [19] and was originally

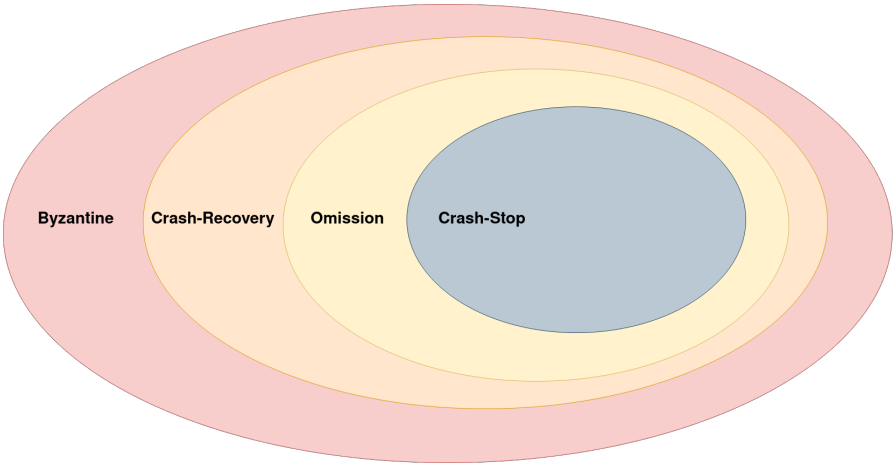

Figure 3.3: Types of possible faults in a network: crash-stop (when the node crashes, it is crashed forever), omission (a node omits sending or receiving message), crash-recovery (a node might crash but may recover after crashing) and Byzantine faults (nodes may behave arbitrarily or maliciously).

intended to enable a secure decentralized payment system and a complete digital money [26]. As Bitcoin and its blockchain technology began attracting attention, alternative blockchain-based cryptocurrencies have emerged such as Ethereum [27], Litecoin [28] or Dogecoin [29]. Then, blockchain technology has expanded into areas beyond financial transactions, such as healthcare [11] [7] [30], energy [31], multimedia [32] or public auditing [6] [5] [8] to cite a few examples, because this technology can also handle (more) complex logic with the use of smart contracts (see subsection 3.3.8). Meanwhile, other blockchain platforms than Bitcoin or Ethereum have emerged, such as Hyperledger Sawtooth [33], Hyperledger Fabric [34] or Cardano [35].

This technology is based on an append-only database (called ledger) with decentralized management, whose content is replicated over the set of blockchain network peers, providing them with a consistent and transparent view of the system, even if these peers do not trust each other. As any single transaction appended to the database by (writer) peers is witnessed and verified by all the participants of the network, the adversary attempts to control the content of this ledger are harder to achieve [6] [36]. For example, one would pretend to be more people than he/she is by faking multiple participants in order to gain a majority vote (Sybil attack); another would have to control at least 51% of the nodes writing in the append-only database for controlling the flow of information being recorded (51% attack [36]). In both cases, the cost and complexity of such attacks are increased,

such that it eventually increases the defence capacity of the system (although the *perfect* security of this one is not guaranteed [18] [36]).

However, while attracting interest from everywhere, blockchain technology is challenging to understand because it combines both cryptographic mechanisms as well as properties of distributed systems. The following subsections will provide an explanation about how the core concepts behind the blockchain technology are tight together: these of transactions (subsection 3.3.1), the blocks (subsection 3.3.2), the blockchain (subsection 3.3.3), the blockchain network (subsection 3.3.4), the cryptographic mechanisms used in this technology (subsection 3.3.5), the main consensus models (subsection 3.3.6), the forks (subsection 3.3.7) and lastly the smart contracts (subsection 3.3.8).

3.3.1 Transactions

Initially, blockchain technology was being used to allow digital money exchanges. These are determined by transactions, namely interactions between 2 parties (one sender and one recipient) that may sometimes involve ownership transfer (e.g. in the case of electronic cash or cryptocurrencies). Each node within the network can send transactions and might be charged for that (there are transaction fees). Transactions contain *at least* the following information:

- data or a list of digital assets to be transferred;
- a reference to the sender of these data or digital assets;
- a proof that the sender has effectively access to the referenced inputs to transfer. This proof is generally ensured through the use of cryptography mechanisms, such as a digital signature (see subsections 3.1.2, 3.3.5);
- the recipient(s) of these transferred data or digital assets

In blockchain networks, the identity of both senders and recipients is not revealed, rather they are using addresses, namely short alphanumeric sequences of characters. Addresses are often derived from the blockchain network user's public key, on which a cryptographic hash function is applied (see subsections 3.1.3, 3.3.5).

3.3.2 Blocks

Blockchain network users submit their transactions on the blockchain network via a software (either a desktop or mobile application, a digital wallet or a web service). This software sends users' transactions to nodes within the blockchain

network. These nodes send also in turn these transactions to other nodes until the information is propagated around the network.

However, these transactions are not yet confirmed or validated: to this aim, they need to be stored in a block published on the blockchain. A block is composed of a block header and a block data. The first one contains metadata for this block (e.g. an identifier of the block, previous block header's hash value, block data's hash value, a timestamp), while block data contains a list of transactions included within that block. Figure 3.4 illustrates this.

3.3.3 Blockchain

A blockchain is a sequence of blocks that are chained together through the reference of the previous block's header for each block. Since each block contains the hash value of the previous block's header (as the output of a cryptographic hash function), if a previously published block were changed, its hash value would also change. It would in turn alter all subsequent blocks since their respective block headers are related to past blocks. There is however one exception: the first block of the blockchain. As it cannot reference a previous block, the first block (called genesis block) contains usually a hash value set to 0. Figure 3.4 provides an example of a blockchain composed of 3 blocks (genesis block included).

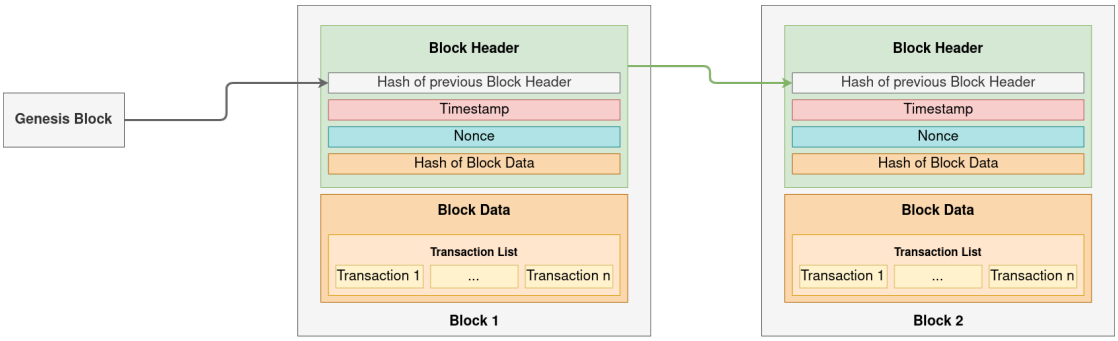

Figure 3.4: An example of blockchain composed of 3 blocks (genesis block included).

3.3.4 Blockchain Network

Blockchain networks are distributed. Consequently, each node is communicating directly with other nodes, as shown in section 3.2. Usually, blockchain networks are categorized into 2 types, depending on which node may add or publish a new block, i.e. write on the blockchain and modify its state [37]:

1. permissionless blockchain networks: anyone can write on the blockchain without any permission;

2. permissioned blockchain networks: only a subset of nodes may write on the blockchain, namely add or publish a new block

Additionally, a network can be public or private: in the first case, everyone can join the network and read *at least* the content of the blockchain (read and write in the case of a permissionless blockchain network); in the second case, only some nodes can join the network.

3.3.5 Cryptography Mechanisms

Cryptographic hash functions A commonly used cryptographic hash function by many blockchain implementations is the Secure Hash Algorithm (SHA), with an output size of 256 bits (namely, SHA-256) [19]. As described previously in subsection 3.3.2, cryptographic hash functions are used for securing the block data and the block header: for a block x, it will provide the hash value of $x-1$'s block header as well as the hash value of x's block data, both stored within block x's header. Cryptographic hash functions are also used for other tasks, such as deriving addresses and creating unique identifiers.

Digital signature Asymmetric cryptography provides a mechanism to verify the integrity and the authenticity of transactions: these latter ones are indeed digitally signed (see section 3.1.2).
With both a private/public key pair and a cryptographic hash function, we can derive addresses [19]: we generate first a public key, then we apply a cryptographic hash function to it and lastly we can convert the hash to text.

Keys management The security of keys, especially private keys, is crucial in a blockchain network: if a user loses his/her private key, he/she cannot digitally sign transactions anymore, nor have access to his/her digital asset (in the case of cryptocurrencies). Usually, users are responsible for managing their keys, by storing them in blockchain client software (as described before).

3.3.6 Consensus Models

In a blockchain network, some nodes can add a block to the blockchain and modify its state. These "writer" nodes (or validators or publishing nodes) [37] compete with each other at the same time to publish the next block. However, they need to agree on the next block to add to the blockchain, for keeping a consistent state of

the system.

There exist several consensus models for having the same decision on the next block to publish (see [38], [19] for an extensive presentation of blockchain consensus models). Among them, the most popular ones are:

- Proof of Work (PoW): These consensus protocols rely on a computationally difficult puzzle to elect a leader that writes to the blockchain;
- Proof of Stake (PoS): Participants vote on new blocks weighted by their investment in the system, such as the amount of currency held in the blockchain (stake);
- Delegated Proof of Stake (DPoS): These consensus protocols enable a democratic way of choosing validators: nodes vote to elect and revoke the rights of delegates to validate the next block;
- Proof of Authority (PoA) or Proof of Staked Authority (PoSA): This consensus mechanism is based on identity and reputation as a stake: writer nodes are known entities that put their reputations on the line for the right to publish a block

When joining a blockchain network, a user agrees not only to the initial state of the system (the genesis block) but also to the consensus model of this blockchain network.

3.3.7 Forks and Conflict Resolution

To add a new block to the blockchain, all nodes (or blockchain network users) must eventually come to a common agreement. Some temporary disagreement is however allowed. In case of conflict, there are two valid chains, called forks [39]. Nodes maintain locally the two branches involved in such conflict and wait until the next block is published, in order to see which branch (i.e. which fork) is the mainstream one. The longest chain is viewed as the correct one and will be adopted, as it has the longest transaction history: blocks may be revoked after being appended to the blockchain, but with decreasing probability as they sink deeper into the chain. Figure 3.5 shows an example of such forks.

Forks are closed to another core concept of blockchain technology, (eventual) finality [39] [40]. A transaction has finality when it is part of a block that cannot change, i.e. when a block is validated and cannot be revoked. Finality can be immediate or eventual: in the first case, a block is immediately validated, i.e. it becomes

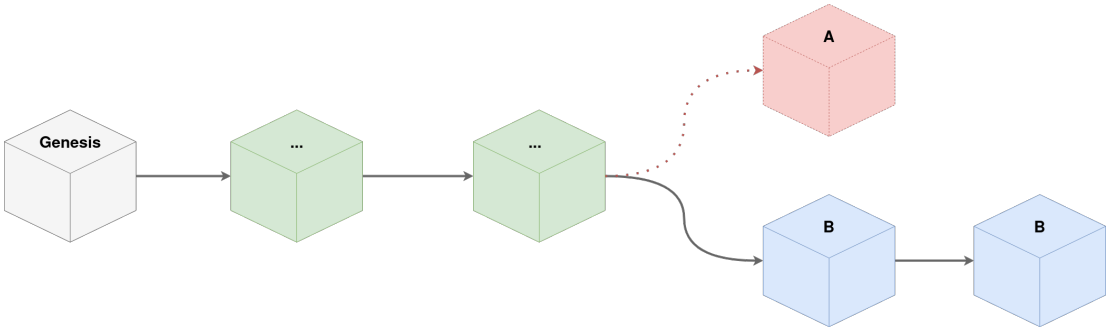

Figure 3.5: An example of blockchain diverging into two potential paths forward: as a new block appends chain B, it becomes the longest chain, thus chain A is now orphaned.

impossible to remove it once it has been appended to the blockchain. Immediate finality favors consistency at the expense of availability [39]. In the latter case, there is some delay before validating a block, hence the risk of forks. Furthermore, transactions rejected on a temporary fork may also have been included in the accepted branch. This is the reason why eventual finality can be defined as the time to wait before considering a transaction irreversible, or as the number of blocks being appended after the one containing this transaction.

3.3.8 Smart Contracts

The last major concept of blockchain technology is the one about smart contracts. These enable the contractual terms of an agreement to be enforced automatically, without any intervention of a trusted third party. With smart contracts, contractual clauses are converted into executable computer programs, run when predefined conditions are met. Each execution of the contract statement is then recorded as an immutable transaction stored within a block of the blockchain. Life cycle of smart contracts is described in Appendix A (following Zheng et al. [41]).

3.4 Trusted Execution Environment

Trusted execution environment (TEE) aims to guarantee confidential and integral code execution [42]. It is implemented as a secure part of the processor, that provides a fully isolated environment with a dedicated address space that prevents other software applications, the operating system as well as the host owner from tampering with or even learning the state of an application running in this secure environment. According to Sabt et al. [43], the foundational component of TEE is

the separation kernel, i.e. the separation of a processor state into a "normal world" and a "secure world". A secure-world software may access all resources, while a normal-world software cannot access the secure-world resources. The purpose of this separation is to ensure that different systems (partitions) can coexist on the same platform while having different levels of security, with the assurance of strong isolation between them. With the advent of such technology, first introduced by the Intel's Software Guard Extensions (Intel SGX) [44], data that must be processed outside of premises (for example in the cloud) can now be protected even from privileged software and physical attacks.

Several TEEs are now available, depending on the targeted processor (CPU) architecture: Intel SGX, AMD (SME, TSME, SEV) and ARM TrustZone. These are briefly compared in Appendix B. We will now present briefly Intel SGX, as being the most popular TEE choice. With Intel SGX, such TEE implementation is called an enclave. A host can instantiate multiple enclaves (or TEEs), which are not only isolated from each other, but also from the host, due to their respective protected address spaces. Data are transferred transparently from a TEE (within the processor) to memory, being encrypted with keys only available to the processor [45].

Chapter 4

An Overview of AuditTrust

This chapter gives an overview of AuditTrust's design and core components. It will show how these are articulated from a high-level perspective. In section 4.1, we will present each component independently. Section 4.2 will put them together and provide a general description of AuditTrust.

4.1 Core Components

For addressing the problem stated in chapter 2, AuditTrust is relying on 5 core components: a blockchain technology (subsection 4.1.1); a blockchain monitoring system (subsection 4.1.2); a decentralized storage system (subsection 4.1.3); several security mechanisms for preserving data confidentiality and data integrity (subsection 4.1.4) and an audit tool (susection 4.1.5).

4.1.1 Use of Blockchain Technology

Our production solution will be based on a permissionless blockchain. Following Wüst and Gervais' methodology [37], we have indeed concluded that a blockchain, and especially a permissionless blockchain, was the best technical solution for our use case (see Appendix C for the application of this methodology to our problem). However, in the context of this project, we have used a permissioned blockchain for ease of development and prototype implementation.

As a reminder, all information shared between the different entities must be transparent, publicly verifiable and trustworthy. These properties can be provided by a blockchain technology [37]. However, by adopting such technology, the question arises of what is stored on the blockchain (on-chain storage) and what is not (off-chain storage), especially since blockchain and blockchain networks are

limited in terms of block size, block creation rate and transaction rate [46] [47]. Therefore, our solution stores hashes of shared data and hashes of access logs on the blockchain (these will be included in transactions), while both shared data and access logs will be stored off-chain.

4.1.2 Blockchain Monitoring System

In AuditTrust, each organization has a monitoring service whose aim is to track any event published on the blockchain related to our use case. For example, it will detect when a data consumer (DC) creates a role or when a data owner (DO) shares a file. Based on such events, this monitoring service will populate a database installed locally at each organization, which maintains a copy of the *useful* information published on the blockchain (i.e. any event related to our use case) for efficiency purposes.

4.1.3 A Decentralized File System for Off-chain Storage

Due to blockchain resources limitations, images, videos and access logs cannot be stored on-chain [46] [47]. Instead, they are stored off-chain, on a distributed and decentralized file system. Such file system has the property to address data based on its content, with the use of a hash. In this way, we can preserve data integrity, by linking off-chain data with its hash pointer stored on-chain [42]. Furthermore, because this file system is distributed and decentralized, data availability is also preserved.

In AuditTrust, every organization maintains a node connected to the network of such distributed file system, for both uploading and retrieving data.

4.1.4 Security Mechanisms

AuditTrust combines several security mechanisms for protecting the system against the attack model defined in chapter 2:

- It uses a role-based access control mechanism enforced by policies issued by DOs, for preventing any unauthorized access to shared data. This mechanism is commonly used for managing information system and improving its security [16] [17];
- For preserving the confidentiality of data, we generate a symmetric key per shared file[1] for encryption purposes. We also encrypt access logs, but in this

[1]We use here shared data and file indistinctly.

case, every entry of these files is only readable by the involved parties (a DO and one or several DCs included in the concerned role). Thus, every line of an access log file is encrypted with the public key of the DO concerned by this access, so that we might have an access log file containing lines encrypted with several public keys, depending on the involved parties;

- As we cannot trust the host system itself on which the client application will run, a malevolent participant might want to leak cryptographic keys by the memory. To minimize the risk of this situation, AuditTrust is using a trusted execution environment, so that sensitive data (for example the symmetric key generated when sharing a file) are provisioned in a secure and isolated hardware-based environment;
- Due to the large number of keys in our system, there is a risk of keys leakage if they are not stored correctly. To this aim, AuditTrust is relying on a secrets management service

4.1.5 Audit Tool and Access Logs

AuditTrust provides an audit tool that can provide 2 types of audit:

- Total audit: this audit will check the integrity of the local database managed by the blockchain monitoring system and outputs a binary value (valid/invalid). For example, if there is any conflicting information between the database and the blockchain, it will output a binary value equal to "invalid". However, this type of audit might be slow, depending on the number of users registered in our system and the number of blocks that have been produced on the blockchain;
- Partial audit: this audit is focused on the data usage control of a particular file. Given two timestamps, it will find all the access logs published during this interval. This operation is faster than the previous one provided that the interval of time is not too large

In both cases, this audit tool needs to decrypt the access logs. In the case of the partial audit, only the entries related to the requested file are decrypted and audited.

Before its hash is published on the blockchain, an access log file is only available at a DC organization: each DC organization records when one of its users has requested the access to a shared data, whether this access was granted or not. A DC organization is incentivized to publish the hash of this access log file on the blockchain because its users cannot access any file until this hash was not shared publicly on the blockchain.

4.2 Global Architecture

Based on the different components described in previous sections, the global architecture of AuditTrust is shown in figure 4.1.

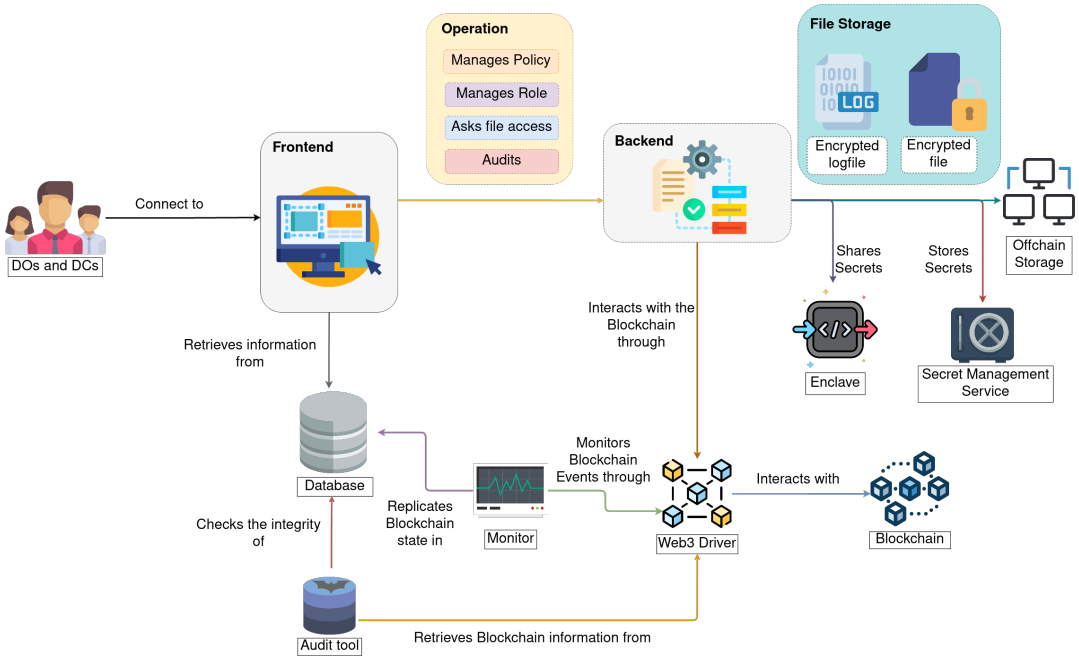

Figure 4.1: Global architecture of AuditTrust, representing from a high-level perspective its core components.

Each organization installs locally a server that will host several core services: a blockchain node (with which it interacts via a Web3 driver), a monitoring system populating a local database, a node for joining the network of a distributed file system (off-chain storage), a node for enabling TEE and a node for storing secrets (cryptographic keys). Besides these core components, AuditTrust also includes a web client application (with several services for rendering web pages) to which users can log in and perform actions, whether as a DO or as a DC. An intermediary service links this web (front-end) and the core services described above (back-end), as shown in figure 4.1.

Chapter 5

Design and Implementation

This chapter will detail the design of our system and its core components, with a focus on the technical choices. Section 5.1 will first recall the setup prerequisites of our solution. Then, section 5.2 will describe our client application. Section 5.3 focuses on the use of blockchain technology within our system and how we interact with the blockchain and the blockchain network. Section 5.4 presents our blockchain monitoring system, before discussing the decentralized storage system used in AuditTrust in section 5.5. Section 5.6 presents the several security mechanisms included in our system. Section 5.7 explains how our audit tool works. Lastly, for summarizing this chapter, we will provide in section 5.8 a technical overview of how AuditTrust's components are interacting together.

5.1 Prerequisites

To work properly and before setting up our solution, AuditTrust assumes that some information has already been exchanged between the different organizations, as stated in chapter 2. These are public information related to the users of each organization:

- their (blockchain) public key;
- their identity (first name and last name), for presenting information related to a user in a more user-friendly way in the client application;
- an identifier of their organization (e.g. IP address);

Furthermore, the system must be provisioned with at least one machine with a processor (CPU) compatible with Intel SGX. However, we recommend for Intel SGX a processor with an enclave size of 512MB or more to get the best possible performance.

5.2 Client Application

In AuditTrust, a user connects to a web application to perform actions, whether as a data owner (DO) or as a data consumer (DC). User accounts are automatically imported (based on information shared in advance by the parties) once this web application is installed. When this application is launched for the first time, a time interval for publishing access logs on the blockchain regularly is also requested. Then, a scheduled task (defined by this time interval) is set.

When a user is logged in as a DO, he/she can add a policy (to which the data he/she wants to share is linked), update it or delete it. He/she can also list the policies that he/she has already created. When a user is logged in as a DC, he/she can add a role, update it (e.g. assign or remove members to this role, edit the role name), list the roles that his/her organisation has already issued and ask access to a file[1]. Whether a user is logged as a DC or as a DO, he/she can use the audit tool for performing a total audit (verifying the integrity of all information for any data) or a partial audit (targeting a specific file).

The technical idea behind this client application was to decouple the front-end and back-end. However, although there are some abstractions in our solution, there is no strict independence between the front-end and back-end. Hence, implementing the application following properly the microservices architecture principles is something we could do in future work to improve our solution (see chapter 8).

The following subsections will describe briefly the frameworks used to implement our front-end and its interactions with the back-end: Django (subsection 5.2.1), FastAPI (subsection 5.2.2), Gunicorn (subsection 5.2.3) and Nginx (subsection 5.2.4).

5.2.1 Django

As our project is developed in Python, we have searched for a web framework in that language as well. Django [48] is the most popular one for this language, well-established, easy to use, open-source, with a massive community of users and well-maintained documentation. Django is also secure by design (preventing for example cross-site request forgeries or CSRF).

An application developed with Django is shipped automatically with a database. In our project, this one is used internally and locally by each organization for

[1] File and data are used here indistinctly.

replicating (partially) the blockchain state. This database is fully managed by Django, using Django's database API.

Given that we need a lot of concurrent write accesses, we needed to choose a database engine that can support multiple simultaneous writers without locking the whole database. This is one of the main reasons we have chosen PostgreSQL. Besides, this database is only available locally, within the network of an organization. Thus, we assume that the quantity of data will be limited and perfectly handled by PostgreSQL. Moreover, a relational database management system (such as PostgreSQL) ensures that all the clients have the same consistent view. Several alternatives were possible but were not selected for diverse reasons: SQLite locks the whole database whenever a write occurs; we cannot tolerate such lock as we should be able to handle $1,000$ requests per second. MongoDB was also a good candidate but as the amount of information stored in the database is limited within a local network and given the complexity that NoSQL databases can bring, we have decided to choose the simplest option for this prototype (i.e. a relational database).

Based on the assumption that members of the same organization trust each other (as stated in chapter 2.3), they have no reason to manipulate the content of this database (it would not be beneficial for them, because it would mean that they hack their own system). Verification of its content is still possible via our audit tool, which will check its integrity based on the content published on the blockchain (see section 5.7.1).

5.2.2 FastAPI

Once we have collected user inputs via the front-end (e.g. when a DO wants to share data by adding a policy), they still need to be communicated with the back-end. A RESTful API was specifically built for this goal, based on FastAPI [49].

Originally, we used Django Ninja [50] before migrating to FastAPI, for having bigger independence between the front-end (Django) and the back-end. A back-end API was built for interacting with the other back-end parts. Therefore, if the front-end is modified in the future, this API can still be reused, with no need of installing Django's full-stack server. Furthermore, as this API is exposed on the network, other services might use it to interact with the blockchain, even if the machine on which Django is running is off. Lastly, FastAPI is popular and described as one of the fastest Python frameworks available [49]. Here, we are using Python for the same reason as before: this is a language we are familiar with.

5.2.3 Gunicorn

Django and FastAPI are both running under Gunicorn [51]. Gunicorn is a web server gateway interface (WSGI), namely a server application which translates an HTTP request to something that Python understands. Django's default WSGI [52] is not recommended for production projects, from a performance and security point of view. For FastAPI, the default WSGI (Uvicorn) is more limited than Gunicorn, as stated on FastAPI website [53]: if there is a need to have a process manager at the Python level, Gunicorn is a better choice. Furthermore, Gunicorn is also recommended for deployment purposes, as stated on Uvicorn website [54].

To avoid a blockage due to synchronous requests, we have set up several workers (i.e. threads managing the requests). According to Gunicorn documentation, a recommended number of workers is equal to 2× the number of cores +1. However, in our case, due to low available resources, we have restricted the number of workers to 2.

5.2.4 Nginx

In front of Django and FastAPI, there is also a Nginx server which is mainly used for reverse proxy and for acting as a web server such that this task is not managed by Django (this is also recommended by the official Django documentation [55]). In our project, Nginx is only used for these purposes but it could also be leveraged later for doing load balancing, logging or restricting all the communications with our services via this proxy, etc.

5.3 Use of Blockchain Technology: Ethereum

Our choice of using blockchain technology is motivated by the requirements implied by our use case. By following Wüst and Gervais's methodology as shown in Appendix C, we have concluded that a blockchain was needed in our use case. As a recall, this one is about the enforcement of a public data usage control between mutually mistrusting entities, as well as secure data sharing.

However, there exist a lot of blockchain platforms. Furthermore, we also need to decide which blockchain platform we will adopt for *development* purposes and which one we will use in *production*. In this latter case, due to our requirements and according to Wüst and Gervais [37], we cannot choose another platform than a permissionless blockchain [37]. One of the most popular representatives of permissionless blockchains is Ethereum [27], which was chosen for this project. There also exist Bitcoin [9] and Cardano [35] as permissionless blockchain alternatives,

but Bitcoin does not support smart contracts (and is primarily intended for money exchange), while Cardano's developer documentation is limited in comparison to Ethereum. See Appendix D for further details.

Ethereum was released after Bitcoin, in 2015, and is inspired by it. Both are permissionless blockchains, which means that every node can participate in the core activities of the blockchain network. In both cases, a proof of work mechanism is used, although Ethereum aims to move towards a proof of stake model. Both are well-established blockchain technologies, still used and actively maintained [56] [57]. Ethereum's main goal is to create an alternative protocol for building a larger class of decentralized applications, by having a built-in Turing complete virtual machine (called Ethereum Virtual Machine, EVM) that enables more complex logic on the blockchain, via the deployment and the execution of smart contracts. Therefore, all nodes in an Ethereum network run an EVM.

As the EVM is a Turing complete virtual machine, a smart contract could contain an infinite loop. For preventing such risk, Ethereum uses its cryptocurrency, the ether, for paying gas, i.e. the internal pricing for executing a transaction or a contract, depending on the number of computational steps required for running the contract. In this case, an infinite loop will require a high amount of gas and thus, high transaction fees, discouraging from having such pieces of code within a contract. Besides this aspect, ether is also used for submitting transactions: the higher the number of ethers is spent, the faster the transaction will be added to a block and confirmed (according to Ethereum's eventual finality [40]).

Ethereum blockchain networks can either be permissionless or permissioned, public or private (for instance, Ethereum's main network, Mainnet [58] is a public permissioned blockchain network). Due to this flexibility (covering all types of blockchains), its popularity, its use of smart contracts enabling more complex logic, but also the ease of software development, we have chosen Ethereum for both production and development purposes. However, due to our requirements, we can only choose a public permissionless blockchain for our production solution, while we are less restricted for our prototype. Therefore, developing a solution targeting Ethereum platform can cover both cases. Other blockchain platforms (with Ethereum smart contracts compatibility) have also been investigated in the case of this project. These are discussed in Appendix D.

In terms of speed, Ethereum's Mainnet blocks are created every 12 to 14 seconds [59] and are validated over 1 minute [40]. In our implementation, a block is produced every 2 seconds and validated directly in order to not wait too long

between operations. We will see now respectively in subsection 5.3.1 how we can set up an Ethereum blockchain network, in subsection 5.3.2 how we can interact with it and in subsection 5.3.3 how we can implement some pieces of code with Ethereum, via the use of smart contracts.

5.3.1 Implementation of the Ethereum Protocol: Hyperledger Besu

For setting up an environment with Ethereum, a client is needed: a client is an implementation of Ethereum that runs an EVM and checks all transactions in a block. For Ethereum, there exist a lot of clients, written in different programming languages (the most popular one is Go Ethereum [60], or Geth, implemented in Go). These implementations have in common that they are following the Ethereum protocol, formally described in the Ethereum Yellow Paper [61]. This diversity is justified for securing the network, making it more diverse and robust (e.g. a bug discovered in an individual client is less of a risk to the network if only a minority of Ethereum nodes are affected). Furthermore, each client has unique use cases and advantages and can be focused on different features or targeting different audiences.

In our case, we have chosen Hyperledger Besu as our Ethereum client. It is an open-source Ethereum client developed by the Hyperledger Foundation in 2019. This latter one is a non-profit organization that aims to develop enterprise-grade and cross-industry open platforms for blockchain-based distributed ledgers. We are using Hyperledger Besu for development purposes, because it implements a consensus protocol supporting immediate finality, i.e. blocks are validated immediately and cannot be revoked (as explained in subsection 3.3.7). Thus, we do not have to handle different versions of the blockchain or forks (see subsection 3.3.7). This consensus is called IBFT 2.0 [62], and it makes the development of our prototype much easier, in comparison to the most popular Go Ethereum implementation, with which we might have forks situations, as it does not provide this nice property (from a developer's point-of-view) of immediate finality. However, as stated by Anceaume et al., this property is provided at the expense of additional synchronization [39].

IBFT 2.0 is a Byzantine fault-tolerant (BFT) algorithm [62]. Consequently, at least 2/3 of writing nodes (called validators) must be correct to create blocks [63] [64]. However, a consensus based on proof of authority (such as IBFT 2.0) cannot be used in production because this type of consensus implies trust relationships between the actors (for recall, they rely on their reputation and identity staked). For development purposes, this is not a problem as it gives us flexibility and allows us to not put an additional load on our machines. However, before putting this

solution into production, some configuration parameters should be changed: the chosen consensus should be modified to Ethash, a proof of work consensus and the network should be connected to the main network of Ethereum, for matching our requirements (we are looking for a BFT consensus algorithm compatible with permissionless blockchain). However, as a result of this change in consensus, we will lose the assumption made about immediate finality, which implies that instead of directly processing information found on the blockchain, we would need to wait a certain number of blocks before processing it (namely once a block has been validated). Ethereum recommends waiting for at least 6 blocks [40], but this threshold can be increased for greater assurance. Nevertheless, the more confidence we want to achieve, the longer we have to wait to process the information, which will result in our implementation to add a queue that will store the retrieved information until we are sure that this information will not disappear from the blockchain.

Lastly, another advantage of Hyperledger Besu is that it can be used for both permissioned and permissionless networks [65].

5.3.2 Interacting with Blockchain: Web3

The most popular interface for interacting with a local or remote Ethereum node is Web3 [66]. It is commonly found in decentralized applications for reading block data, interacting with smart contracts, sending transactions, etc. Main implementation of this Ethereum API is written in JavaScript (web3.js [67]), but a Python implementation of this collection of libraries also exists. We are using this implementation (called Web3.py [68]) in the context of this prototype.

We have encapsulated an instance of Web3 and extended its functionalities in a self-defined class called `Web3Manager` and connect Web3 to the Ethereum (blockchain) node via HTTP. `Web3Manager` interacts mainly with our API built on the top of FastAPI: for almost every action made by a user (add a policy, delete a role member, etc.), the API retrieves the Ethereum account of this user as well as the smart contract associated with this action. Both of these pieces of information are needed to send a transaction on the blockchain, because 1° a transaction must be signed by his/her issuer and 2° we would like to execute the smart contract, by calling one of its functions that will perform the requested action.

`Web3Manager` is also extensively used by another component of our system: our blockchain monitoring system (see subsection 5.4), which reads the content of each published block and the transactions of this latter one via `Web3Manager`. It will request it to decode the event associated with each transaction, contained in the transaction log.

In a nutshell, `Web3Manager` (and more specifically, Web3) is an essential component of our system: once the blockchain network is set up with Hyperledger Besu and once some nodes have joined this network, Web3 lets us interact with the blockchain network, by using these nodes to send transactions, read the content of a block, deriving an account address from a public key, digitally sign a message, etc.

5.3.3 On-Chain Logic: Smart Contracts

Ethereum's main advantage is that the EVM is a Turing complete machine. Therefore, programs can be run on the blockchain and some actions can be automatically executed based on conditions that have been met. These actions are defined in smart contracts. In the context of this prototype, we have developed multiple contracts (shown in figure 5.1) for representing actions that can be made respectively by a data owner (DO), a data consumer member (DC member or DC user) and a data consumer organization (DC organization):

- `DataOwner`: There is one smart contract of this type deployed per DO. This smart contract handles all policies issued by a data owner and all actions related to these policies: add a policy, update a policy or remove a policy. Given that a policy is always related to one DO, it is easier to do the management process in one smart contract (with all policies included in a list of policies) per DO instead of having multiple smart contracts per DO;

- `DataConsumer`: One smart contract of this type is deployed each time a role is created by a DC. We cannot replicate the same structure as before because the management process is easier to manage and audit: by having a smart contract per role, we only have a table of role members, instead of having a more complicated data structure (a matrix) for keeping track of all roles issued by all members of a DC organization (with a table of role members for each role issued by each DC member);

- `DataConsumerOrganisation`: This smart contract is used for sending access logs to the blockchain and making this process much easier and much cheaper, by tracking and aggregating access logs for all managed roles instead of publishing them independently regularly, which would increase the transaction fees of the solution. This smart contract is also responsible of validating the organization's roles. Although it requires first deploying 2 smart contracts and 2 transactions (for creating a role and validating it), this strategy is much cheaper on the long term.

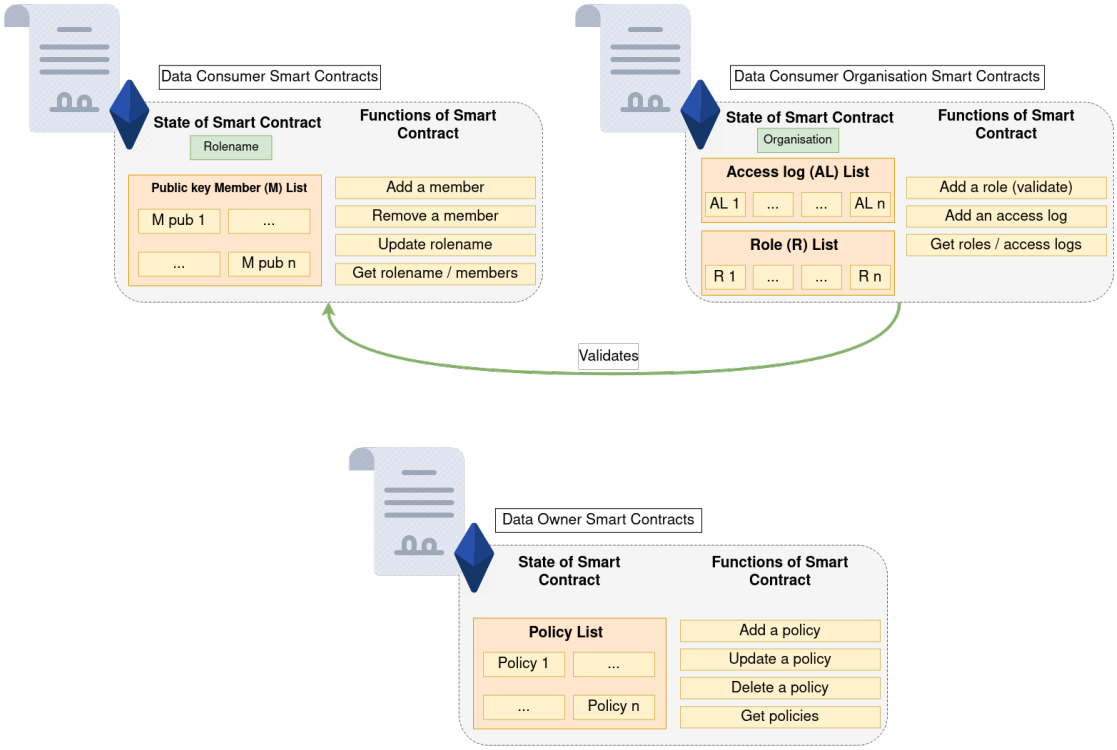

Figure 5.1: Types of smart contracts executed and deployed in AuditTrust.

The different functions included in these smart contracts are also emitting events. These will be stored in the transaction's log, a special data structure that will be incorporated into the blockchain and that can be leveraged by our application for knowing what a transaction is about (e.g. add a policy, remove a role member, ...). We can indeed rely on Web3 filters to retrieve the transactions of interest, based on these events, instead of iterating over all transactions contained in a block. Appendix E provides the list of events produced by our smart contracts.

5.4 Blockchain Monitoring System

As AuditTrust will run eventually on a public permissionless blockchain network (Mainnet) with other nodes that are sending various data, we need to find a way for filtering the transactions and getting the ones that are related to our solution. The events emitted by the execution of smart contracts functions are a way to filter these transactions, as well as the public keys provisioned beforehand by the organizations.

Our monitoring system is "listening" to events emitted on the blockchain and interacts closely with Web3 for decoding these and for knowing what are the parties implied by these events (e.g. what are the associated Ethereum account addresses related to public keys displayed in a transaction). With these decoded events, our blockchain monitoring system aims to populate a local database (running with PostgreSQL, as stated before) with only the relevant information concerning our use case. Having a database that replicates partially the blockchain content is an increase of complexity (from a consistent point of view), but it is motivated for improving the performance and the reactivity of our system (retrieving and finding specific information from a database is faster than retrieving it on the blockchain).

The monitoring system proceeds as follows for populating the local database of its organization (see also figure 5.2): based on the "From" field of a transaction, we will first check if the transaction is related to our system. In this case, we will check if this transaction is about a smart contract getting deployed (such transactions do not contain a recipient address) or about an existing smart contract (such transactions contain a recipient address, which is the smart contract address). If a smart contract is getting deployed, we will try to infer the type of this smart contract, by calling successively specific functions of `DataConsumer` smart contracts, `DataOwner` smart contracts, etc., until a call is successful (i.e. it does not return an error). If there is no successful function call at all, then this smart contract is not monitored; otherwise, we can monitor it by storing its address in the database. It is worth noting that calling a smart contract function is not being charged because there is no modification of the blockchain current state and it can contact the local blockchain node. However, this "try and catch" mechanism is costly in time, this is the reason why it is more efficient to store the information directly within a database. Figure 5.2 illustrates the decision-making process of our blockchain monitoring system.

However, a problem arises when we want to monitor the blockchain: we need to define when the monitor should start analyzing blocks (from the genesis block? from the n latest blocks? etc.). Based on a smart contract address, it is not possible to get the block identifier in which there is a transaction related to the deployment of this smart contract. Here, we assume that the monitoring system is launched before the deployment of the different smart contracts on the blockchain.

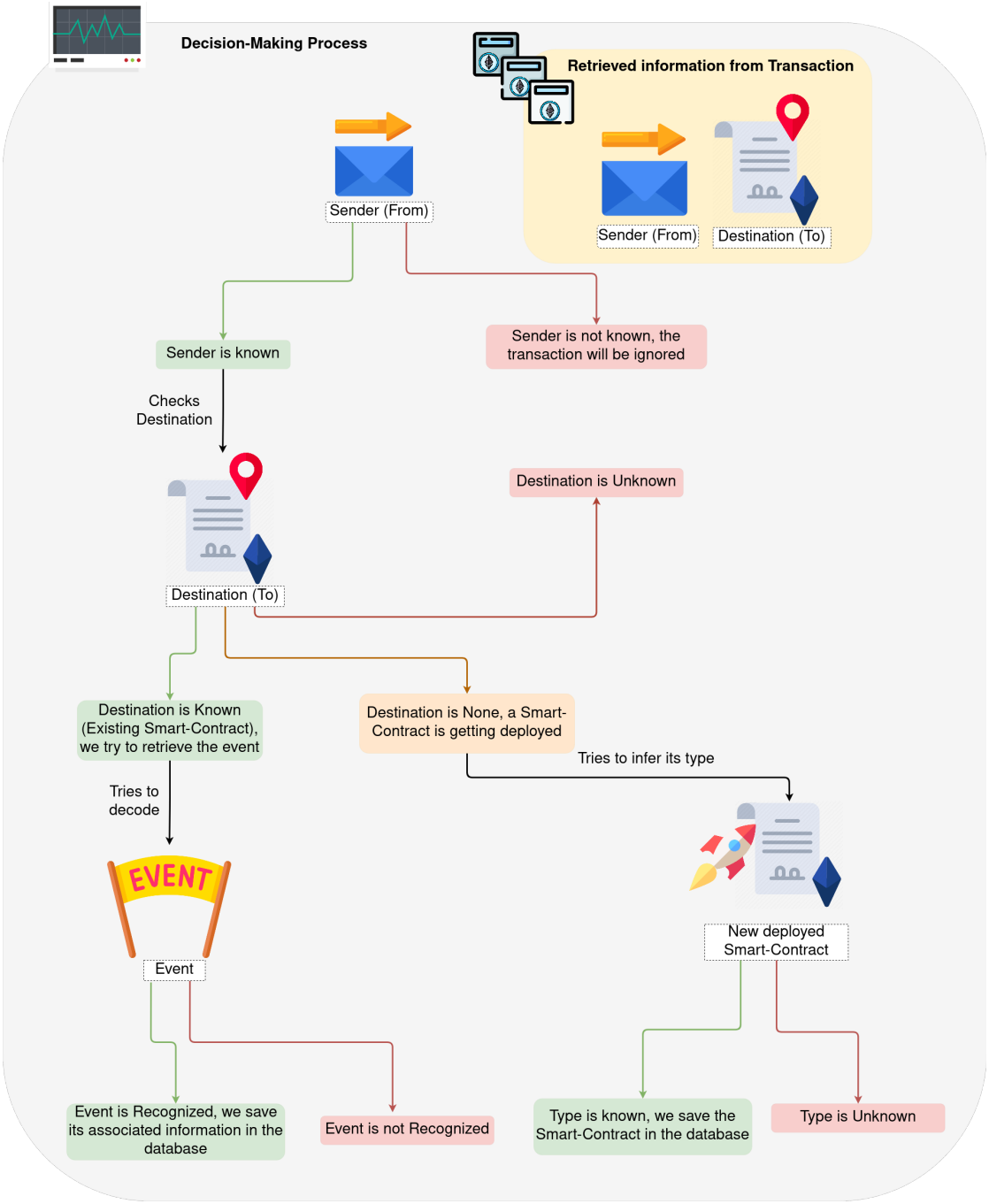

Figure 5.2: Decision-making process of our blockchain monitoring system.

5.5 A Decentralized File System for Off-chain Storage: IPFS

The off-chain storage is a core component of our solution for sharing files between DOs and DCs, but also for storing the access logs. IPFS [69] and Swarm [70] are the most popular decentralized off-chain storage solutions. In our case, we have chosen IPFS because it is more complete than Swarm ([71], see also Appendix F for a presentation of some distributed file systems).

IPFS retrieves a file (or other types of data) by its content, not its location (content-based addressing). Each object stored in IPFS is split into chunks of blocks, where each block is identified by a content identifier called CID (or IPFS hash). For keeping track of these chunks, IPFS uses a Merkle tree and each file can be accessed by the hash of its content. Consequently, once a file is modified, new chunks of blocks are generated, which means that there is also a new Merkle tree, allowing file versioning but also detection of any (file) manipulation. Replication is passive and cache-based with IPFS. It is also possible to store a file locally (i.e. on its own IPFS node) by a pinning mechanism: a node pins a file to store it explicitly (under a persistent manner). Thus, for deleting a file on IPFS, it needs to be removed from every pinning node.

We have set up a private IPFS network so that the files are not being exposed over the Internet and no incentive system (such FileCoin) is used. For being connected to this private network, every participant needs to have the appropriate key.

We are also using IPFS for the redundancy of the access logs, thanks to the pinning mechanism. Once the availability of a new access log file is published on the blockchain (and once this event was detected by the monitoring system), we use the IPFS's pinning mechanism to store automatically this file on other IPFS nodes (belonging to other DOs or DCs). Therefore, file deletion is impossible while it is still pinned by a node. Thus, if a malicious actor wants that an access log disappears, it should unpin this access log on every participating IPFS node. With this mechanism, we can be sure that the file will remain stored in the node even if other nodes have deleted the file, except if it is unpinned at some time.

5.6 Security Mechanisms

AuditTrust's security mechanisms are numerous. We will first provide a detailed description about how we address our threat model in subsection 5.6.1, then how cryptography is used in different places of our system for ensuring data

confidentiality and authentication (subsection 5.6.2) and lastly how secrets are securely shared between the actors (subsection 5.6.3) and how they are managed (subsection 5.6.4).

5.6.1 Data Integrity and Data Confidentiality

As stated previously (see section 2.4), an attacker might search to compromise the confidentiality of the shared data and access logs or to manipulate these latter ones. In practice, this would mean one of the following situations: 1° a malicious actor tries to access another existing file than the authorized one (another valid IPFS CID is provided); 2° a malicious actor tries to access a file that does not exist (an invalid IPFS CID is provided). These situations can occur in 2 different cases: 1° when the organizations have been notified that a new access log file is available (via the appropriate event); 2° when a data consumer wants to access a shared file (image or video) or audit a specific file.

Case 1: New access log available When a new access log is available (i.e. its CID was published on the blockchain), DOs will inspect the content of this file for retrieving and checking the accesses related to their respective files (shared via policies). An access log is structured as follows (see also Appendix G): for each DO (identified by his/her public key), there is a list of accesses intents encrypted with the DO's public key (see figure 5.3). Each access intent (i.e. an entry of an access log) contains the following information (once decrypted): when a file was requested (i.e. date and time), by whom (i.e. DC's Ethereum account address), which file was requested (IPFS hash) and with which permission it was requested (read-only or read and write). Each access intent comes along with a message digitally signed by the data consumer. This message contains all the information stored in the encrypted access intent.

Thanks to this structure, a DO can easily find all the accesses related to his/her shared files and we can inspect if the access log entries are correct. For each line related to this DO, we extract the access intent and the signature. Then, the following procedure, composed of an authentication verification (figure 5.4) and an access verification (figure 5.5), is executed passively and automatically by the monitoring system: first, we will check the signature (authentication verification, see figure 5.4). We decrypt the access intent and retrieve all information upon which a valid signature is built: date and time of the access intent, DC's Ethereum account address, IPFS hash and requested permission. We will derive the Ethereum account address from the provided signature and compare it to the DC's Ethereum account address included in the decrypted access intent. If these addresses are

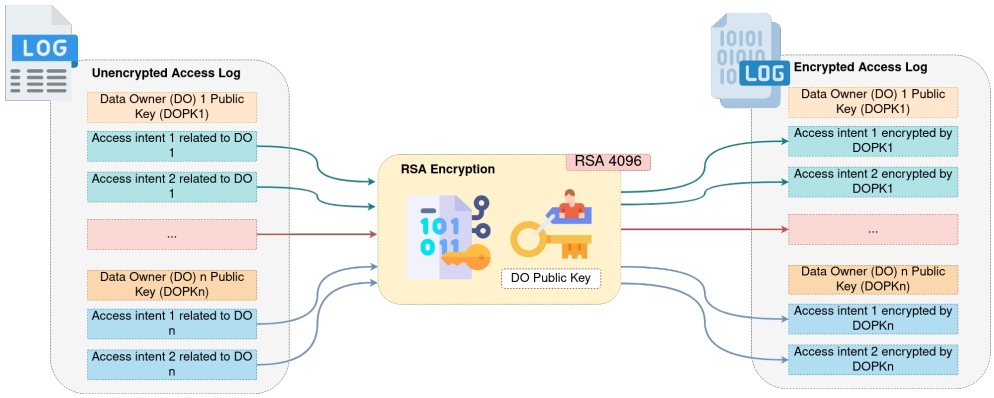

Figure 5.3: Access log encryption (with RSA 4096 bits).

equal, both messages (the decrypted one by the DO and the signed one by the DC) are equal: the identity of the DC is verified, there is no usurpation attempt. We can thus check if the member is authorized to access the requested file (step 2, access verification). If these addresses are not equal, we ignore this access intent because the Ethereum account address is not known and/or one of the 3 situations described above have occurred: if one field changes (date and time, IPFS hash, permission, etc.), the signature does not correspond anymore to the one we should have based on the decrypted information, meaning that there was a data integrity violation. Second, we will check the access (access verification, see figure 5.5). Based on the DC's Ethereum account address, we will retrieve his/her public key; with provided IPFS hash file, we will check within the local database to which policies this file is assigned (with an expiration date greater than the date and time provided in the decrypted access intent). For each policy, we will inspect its associated role and in particular, the list of DC members included in this role (represented by their respective public key): if we find the DC's public key in this list, this access intent is correct and authorized. It will be added as such in DO's database, as well as in the DC's database. In this latter case, it is done once the IPFS hash of a new access log (generated by this DC) has been published on the blockchain and detected by the monitor. This is done regularly, depending on the time interval set during the installation for transforming an access intent into an authorized access intent in the database. On the contrary, if the DC's public key is not found at all (in any role members list), the incident is reported.

Case 2: Access a shared file or audit a file Once an access intent was transformed into an authorized access intent (by the scheduled task whose time interval was defined at the system setup), a DC can communicate with a DO via a socket to ask for access to one of his/her file. First, a DC says if he/she wants

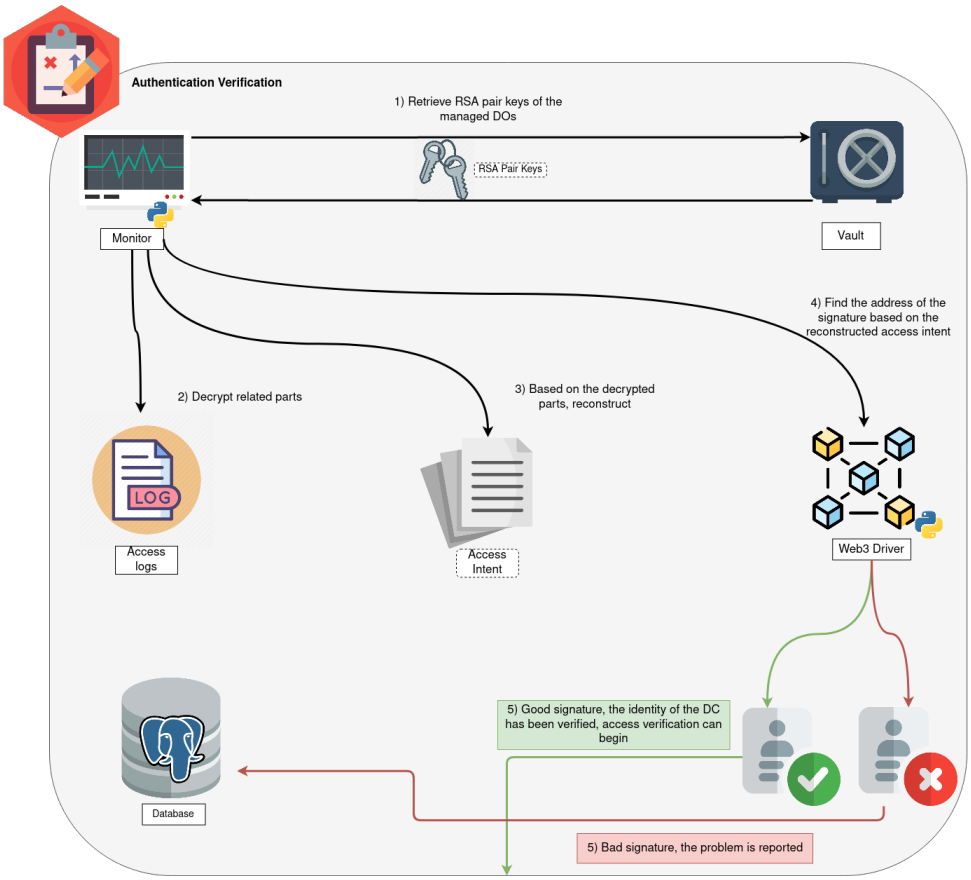

Figure 5.4: Verification process when an access log has been published (1/2).

to access a shared file or if he/she wants to audit a file. Then, we implement a challenge/response protocol (taken from Cruz et al. [72]) for authenticating this DC (see figure 5.6). First, the data consumer sends his/her Ethereum address account as well as the IPFS hash of the file he/she wants process. At the DO side, we will check whether this access intent is authorized or not (based on the background process described in case 1). If it is an authorized access intent, the DO will generate a random message (a challenge) to be signed by the DC for authentication purposes. It is worth mentioning that the DO knows DC's public key. If the DC wants to access the file, he/she needs to sign the challenge and give it back to the DO. The DO will check the signature following a similar logic to the one shown in case 1: it will retrieve the DC's Ethereum account address from the signed message and compare it to the Ethereum address provided by the DC when the connection was established. If the addresses are equal, the signature is valid: the DO will communicate the port on which the data consumer can connect

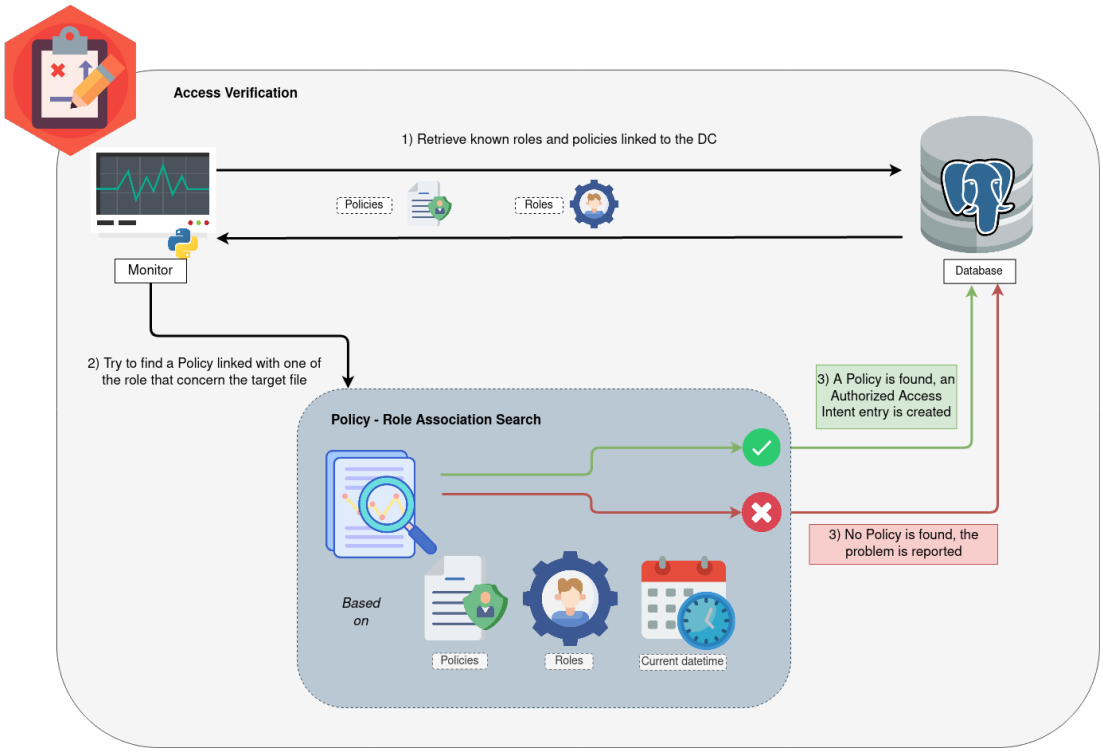

Figure 5.5: Verification process when an access log has been published (2/2).

to retrieve either the symmetric key for decrypting the requested file or the RSA private key for decrypting part of access logs. If the access intent is not authorized or if the addresses are not equal, the connection is closed. Figure 5.6 shows this process.

Another possible data integrity violation is the modification or the deletion of the access log files. As we have seen before (in section 5.5), if the content of a file published on an IPFS network was to change, it would generate another IPFS hash file. This case should not happen given the immutability nature of IPFS. Additionally, in the case of an access log file, once its IPFS hash is published on the blockchain, the organizations store this version of the file by pinning it. Thus, even if an access log file would have been compromised, they still have a pointer (and a copy) to the correct version and its complete deletion cannot happen while it is still pinned.

It is worth mentioning that an access log file (even if encrypted) can be falsified by a DC organization *before* the publication of its IPFS hash on the blockchain. However, a DC user does not have any interest of doing such denial of service because he/she cannot access files until the access log is not published and he/she

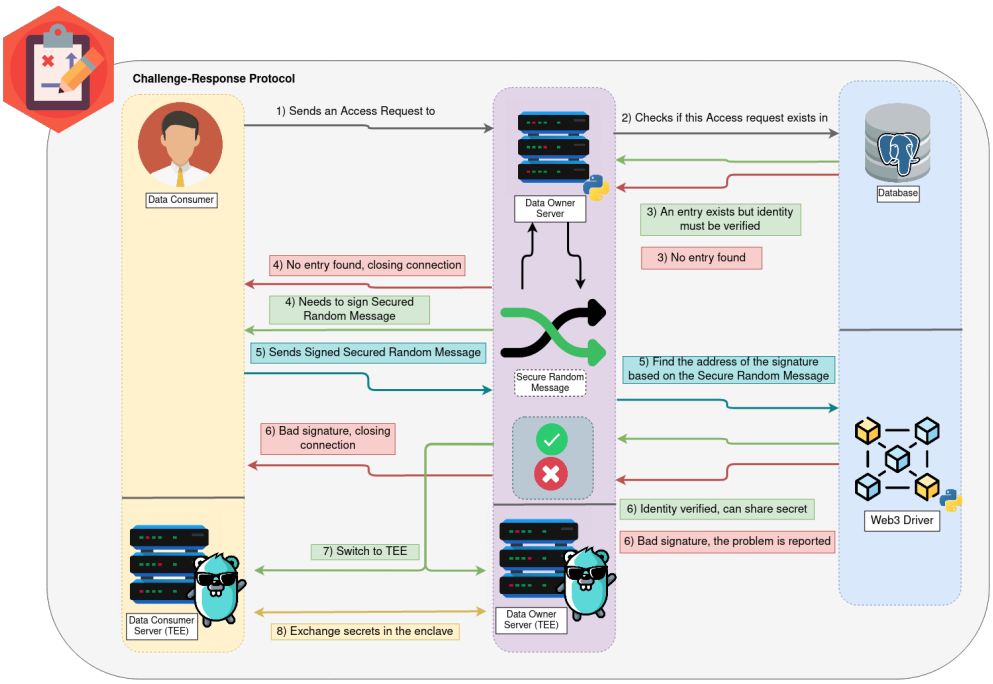

Figure 5.6: Our challenge-response protocol.

would pay for publishing garbage data. Furthermore, DOs do not trust the access logs: they run several verifications (as shown before) to check whether the access intents are correct. Except for processing time, it does not impact a DO. However, we could put in place a mechanism for punishing the organization (as a whole) whose users try to add noise within the access log files for discouraging such attempts, e.g. by forbidding access to data, etc. Let us see now dive into the details of how cryptography is used in different places of our system.

5.6.2 Cryptography

Encryption is used for ensuring the confidentiality of our data: even if a malicious actor gets files stored on IPFS (and bypasses our authentication and authorization mechanisms put in place), he/she would not be able to read the content of these files:

- in case of data put at disposal by DOs (images and videos), we are using symmetric encryption. A symmetric key is used to encrypt the file before this one is uploaded on IPFS. The encryption process relies on the AES-256 algorithm and one symmetric key (of 32 bytes) is generated per uploaded file. Symmetric keys are managed and stored in a specific service, called Vault

(see subsection 5.6.4).

- in case of access logs, these files are encrypted with multiple public keys: for preserving the confidentiality, every entry of these files (i.e every access intent) can only be read by the involved parties (the concerned DO and authorized DCs). Consequently, each entry is encrypted with the public key of the data owner related to this entry (which represents an access intent). We are using RSA keys of length 4096 bits to do that. Figure 5.3 show this.

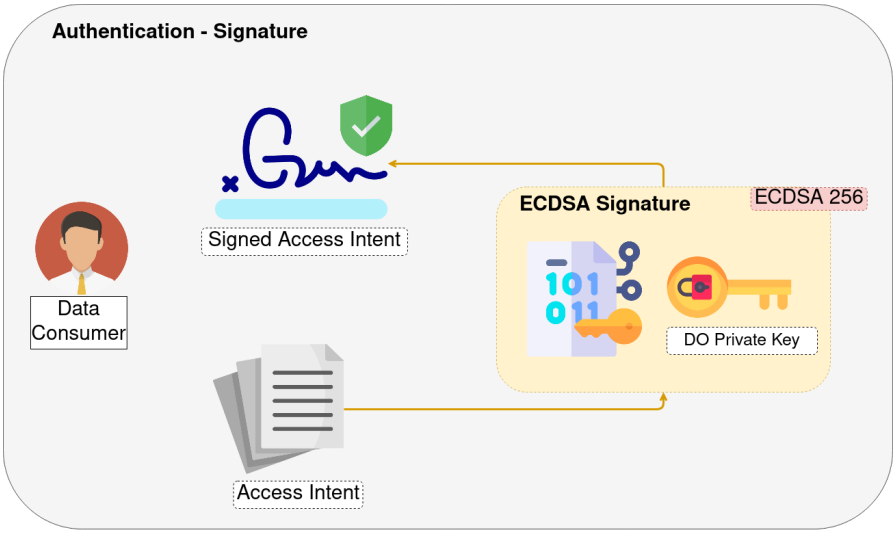

Figure 5.7: Signature of an access intent with the signature algorithm used by Ethereum (ECDSA - Elliptic Curve Digital Signature Algorithm).

Besides encryption, we also use cryptography for authentication purposes: as shown before, in addition to encrypting an access intent with the data owner's public, each DC has to sign this message with his/her private key. Our challenge/response mechanism also requires that a data consumer signs a message for being authenticated. This digital signature is shown in figure 5.7.

These 2 aspects (encryption and authentication) are implemented by our solution, but these are also used recurrently by the technologies we are using, as security layers: a transaction is also signed with an Ethereum private key, for setting up an IPFS network a key is also shared between the IPFS nodes, etc.

5.6.3 Sharing Secrets with a TEE: Intel SGX

In our use case, there is no trust at all between the different entities. Thus, our solution cannot rely on a trusted third party for accessing the data. A solution

could be that the DO decrypts the requested file before sending it to the DC. However, this would put a lot of pressure on DO's nodes and limit the scalability of the system, especially if DO and DC are dealing with large files (such as videos). Therefore, it is more suitable to only share the symmetric key or the RSA private key per request instead of a large decrypted file. However, according to our problem, we also suspect the host system to which these keys will be shared to be malicious. Therefore, we need to find a safe manner for exchanging keys without these being leaked, even by the memory. As shown before in subsection 3.4, a trusted execution environment (TEE) is a novel hardware-based technology for providing such guarantee.

Several TEEs solutions exist, depending on the processor architecture. In the case of this project, we have first tried to use ARM TrustZone with OP-TEE library [73] before switching to Intel SGX, due to package dependencies issues. 2 main libraries exist for developing with Intel SGX (besides Intel SGX SDK): Open Enclave [74] and Ego Go [75]. This latter one was preferred over the first one for ease of development, although the safety of this library should still be analyzed: Weijie et al. have found numerous weaknesses in the libraries used for developing with Intel SGX, but they do not have analyzed Ego Go [76].

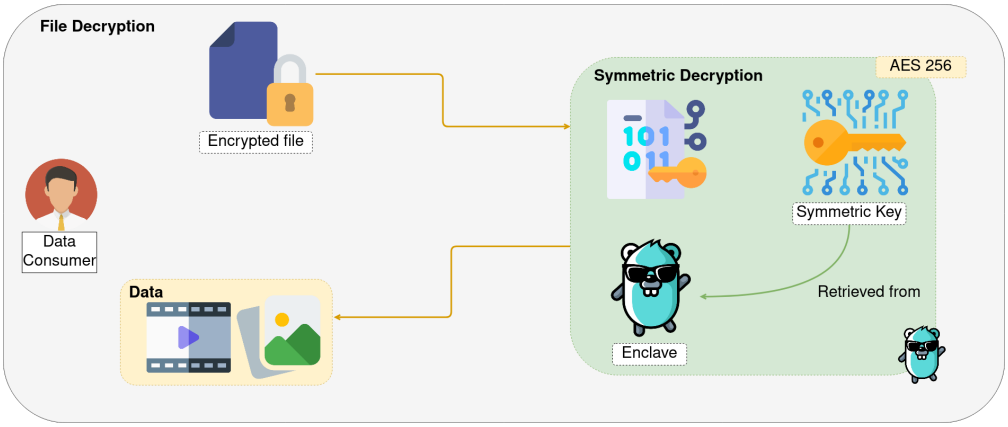

Figure 5.8: File Decryption (Symmetric).

We are using the enclave once the challenge/response has been successful, as described in case 2 in section 5.6.1 (figure 5.6): at the DO's side, a server is running and waiting for DC's request for launching the TEE phase. If the challenge/response is successful, the port of this server is communicated. However, this communication does not happen between DO's server (we call it "DO node" in our implementation) and a DC's workstation, because it would imply that this workstation must have the required hardware for running Intel SGX and it would have made the development

of the protocol much harder. Instead, we are using a specific node at a DC's side, called "DC node".

2 operations can be handled by the enclave, but only one at a time:

- the decryption of a shared file (figure 5.8): the symmetric key used to encrypt the file is sent to the DC node. The DC node decrypts the file in its entirety using the received key;
- the decryption of an access log file by a data consumer: as an access log contains access intents towards DO's shared files encrypted with the public keys of these DO, a DC cannot audit a file because he/she cannot decrypt the entries of an access log file without the involved DO's private key. The enclave is thus used here for sharing this key (securely) with the DC

Once the decryption phase is finished, both sides leave the enclave, leaving those keys from any eavesdropping who might try to retrieve the secrets from the memory. Furthermore, the socket communications in the normal world and in the enclave use TLS for preventing any eavesdropper (e.g. Man-in-the-Middle attack) to retrieve any secret.

5.6.4 Secrets Storage with Vault

Since the prototype we are developing must be user-friendly, we cannot assume that all users will be able to store securely secrets such as symmetric keys and RSA key pairs. This is the reason why we are using an additional security layer: a secret management service. In our case, we will rely on Hashicorp Vault [77], developed by HashiCorp.

If a secret needs to be retrieved, the request needs to be authenticated and allowed to access such secrets through access control list (ACL) (not implemented in our prototype), otherwise, the incident is reported. Every secret stored in Vault is encrypted and each time a secret is retrieved, the transit phase is also encrypted, guaranteeing that a Man-in-the-Middle attack could not happen. Vault can also scale up: several Vault instances can co-exist (based on a Raft consensus to do so).

In our prototype, each organization has one local Vault instance, on which every symmetric key and RSA key pair are stored. This instance is not exposed, limiting the attack surface on this instance. For this prototype, we are importing RSA key pairs during the initialization of the Vault node, similar to the import of Ethereum user accounts when setting up the client application.

5.7 Audit Tool and Access Logs

The audit tool is the central component of our solution because our goal is to provide a reliable tool for tracking the data processing of the data shared by mutually mistrusting entities. This tool allows to do 2 types of audit:

1. A *total* audit: This audit will look if there is any wrong information included in the database, by comparing the information stored in it with the one included in the blockchain. It will output a boolean: valid/invalid state, according to the integrity of the database. However, this operation could take a lot of time depending on the number of participants, the number of operations published and the number of blocks that have been produced;

2. A *partial* audit: This audit is focused on a particular file. Given two timestamps, the audit tool will find all the access logs related to this file during the time interval defined by the timestamps. This operation should be quicker than the previous one if the interval of time is not too large.

The following subsections will detail these 2 types of audit: subsection 5.7.1 explains how is performed the total audit; subsection 5.7.2 explains how is performed the partial audit.

5.7.1 Total Audit

The integrity of the database is checked when we want to know whether our system has been compromised or not (with possibly tampered information about access logs) because we are relying on this database (local to an organization) before or after sending a transaction on the blockchain. We are storing in it some crucial information, such as smart contract addresses, public keys of the role members, etc. Even if this database is local to an organization and members of the same organization are trusting each other, it might be accessed by an external attacker if one finds a way to compromise a DO node or a DC node, as these nodes are the only ones that are exposed directly to the other organizations. Thus, for checking the integrity of the database, we will compare its content to the one included in the blockchain. However, this mechanism does not report an attack that modified *temporarily* the content of the database.

The first step of this total audit is to find the smart contracts issued by our system. Based on the public keys shared beforehand (as stated in section 5.1), we can associate the Ethereum account address related to each public key. However, we do not know from which block the analysis of the blockchain should start. As we do not know yet what are our smart contracts addresses, we could also not

retrieve easily the oldest `DataConsumerOrganisation` smart contract, as indicated in section 5.4. The only way would be to go through the whole blockchain because we do not know where and when the solution has started, at least not from a trustable source. This operation would take a lot of time depending on the number of blocks in the Blockchain.

Another solution is to leverage the mechanism of Ethereum for generating a smart contract address: in Ethereum, a smart contract address is computed from the Ethereum account address and its associated nonce (where a nonce is an incremental value equal to 1 initially) [78]. Thus, we will do a brute-force for finding an existing smart contract address issued by a user of our system, based on his/her Ethereum account address and the nonce associated with this account address (retrieved via Web3). This brute-force mechanism is a convenient hack to prevent having to do a slow and time-consuming search of the blockchain. Once a smart contract address is generated, we will infer its type by following the same "try and catch" mechanism as before (see section 5.4): we will call a specific function of a `DataConsumer` smart contract, if the call is successful, we know that it is a `DataConsumer` smart contract; otherwise, we will call a function of a `DataOwner` smart contract, etc. We store in memory the generated smart contract address if it is related to a known smart contract type (`DataConsumer`, `DataConsumerOrganisation` or `DataOwner`).

Once we have retrieved all smart contract addresses emitted by our users, we can start the process of checking the integrity of the database: for each smart contract, we will call its different getter functions and check if the information kept in the database is the same as the one retrieved by these getters. For recall, there is (at least) one blockchain node per organization and calling a smart contract function on this local node is free (blockchain nodes keep a local copy of the blockchain data).

5.7.2 Partial Audit

The partial audit is focussing on all accesses that have been made to a particular file within a given time interval. Here again, we will compare the information present in the database to the one included in the blockchain, for having an additional security mechanism that proves the integrity of our data. As for the total audit 5.7.1, we first need to retrieve the smart contracts that are emitted by the members of the different organizations. We are following the same mechanism as explained in the previous section for getting the smart contract addresses.

Once these smart contract addresses have been retrieved, the first challenge is to find which blocks were emitted during the time interval provided by the user. We

rely on the average block time (i.e. the average time it takes to add a block on the blockchain) for inferring this information.

Once we know which blocks must be analyzed (by retrieving their respective identifiers - block number), the next step is to find the policies to which the file we want to audit is associated, as well as its owner (a DO). The goal is to find which organizations are associated to these policies through their roles so that we can know from which organization we need to look at their access logs. With all this information (IPFS hash of the file and smart contract address of organizations having roles associated with the policies assigned to this file), it becomes easy to find the access logs related to this file: we are iterating over the `DataConsumerOrganisation` smart contracts related to this file to filter the events notifying the availability of a new access log[2]. As the transactions containing this type of events contain the IPFS hash of the related access log file, we will download the file associated with this IPFS hash for each relevant transaction to the file we want to audit. From this downloaded file, we only keep the entries that are potentially related to the file to audit, i.e. we only keep the accesses made for any file shared by the DO associated with that file (for recall, the structure of an access log file is provided in Appendix G).

If the audit is requested by the DO of this file, each entry is decrypted with his/her RSA private key. If this decrypted entry is about the file we want to audit, we show the information of this entry in the audit tool; otherwise, we skip the entry. If the audit is requested by a DC having access to this file, the mechanism is a little bit more complex, as he/she does not have the DO's RSA private key for decrypting the entries. In this case, as shown before, we will use an enclave for sharing this key securely with the DC and allowing him to decrypt the relevant entries (see section 5.6.3).

5.8 Technical Architecture

Now that all AuditTrust's components have been deeply described, we can present a more complete view of the architecture of our solution, as well as the articulations between our different components. This is what our figure 5.9 shows: a user connects to the web client application via Nginx, Guvicorn and Django. He/she performs on it some actions (add a policy, request access to a file, etc.). Django will use API calls defined by FastAPI for transferring these requests to the right components: Web3, IPFS, Vault (each service is running as a node within our

[2]For recall, we do not need to go through all the transactions contained in a block, thanks to the events: there are Web3 filters for retrieving the transactions of interest directly.

system). If authorized, each request will execute a specific smart contract function that will issue a transaction containing an event from the blockchain node (on which Hyperledger Besu is installed and configured). This event will be caught by the monitoring system of each organization and the related information will be added to a local PostgreSQL database. If the request is about the upload of a file, the IPFS node will be contacted. If after some verification processes a request requires the secure provision of encryption keys, the enclave will be invoked to perform this secure provision.

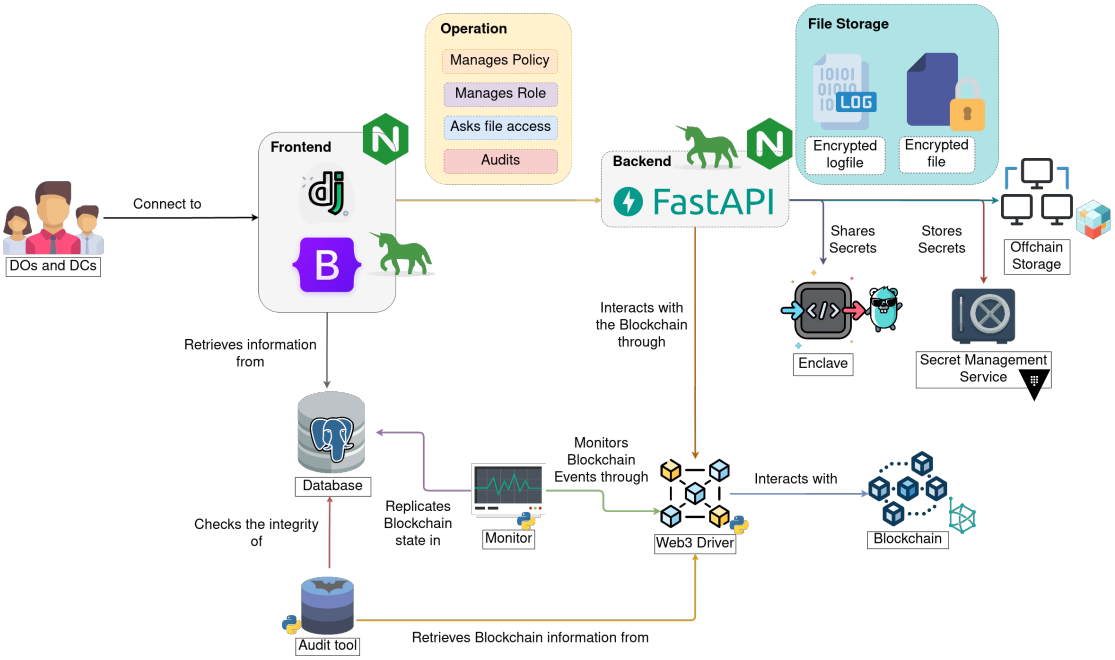

Figure 5.9: Technical architecture of AuditTrust.

Chapter 6

Evaluation

This chapter will provide a performance evaluation of AuditTrust, by sending a large number of requests to put it under pressure and to observe how it will behave. By this way, we can also prove the scalability of our system, as well as its proper functioning. Section 6.1 provides technical details about the testing environment. Section 6.2 describes how our tests have been conducted and section 6.3 shows the results we have obtained. In section 6.4, we will discuss the limitations of this performance evaluation.

6.1 Testing Environment

The setup environment we use for evaluating our system is 4 machines virtualized by KVM [79]. The tests were conducted on a laptop with the following technical specifications:

CPU	**Memory**	**Hard Drive**	**Running Programs**
AMD 5800h (8 Cores / 16 Threads)	32 Go of RAM	2 TB (SSD NVME)	KVM VMs + PyCharm launching the testing script

2 cores and 5 GB of RAM with 20 GB of disk space are allocated for each virtual machine (VM). We rely on 4 VMs in order to have a functional PoA with IBFT 2.0, as specified in Hyperledger Besu's documentation [80]. As a reminder, here are the different services that run on each VM:

- Hyperledger Besu with a block production every 2 seconds (blockchain node);
- IPFS (for off-chain storage);
- Nginx (as reverse proxy and as web server);

- Django running under Gunicorn (for the client application);
- FastAPI, also running under Gunicorn;
- A socket server on which a simulated version of Intel SGX is enabled;
- A Vault instance

6.2 Evaluation Procedure

Our evaluation procedure is composed of 2 phases, that will overload different services: first, the pressure is put on the blockchain network, by generating a lot of requests regarding the management of policies and roles (adding a policy, removing a policy, updating it, adding a role, adding members to a role, etc.). Second, the load will be put on FastAPI and the enclave process (TEE). In this phase, the requests that will be sent are about accessing a file or asking access to a particular file. This second test phase is highly dependent on the first one, as there can be no access if the user does not belong to roles with which files have been shared (policies).

The tests were conducted in such a way that they are following a predefined sequential order of requests interacting with our system. Within this context, we send 100 requests per user (with 8 users in total). The requests are distributed as follows for the first phase (about policies and roles management):

1. 30 requests for creating roles (9 members are added at each role creation);
2. 3 requests for updating roles;
3. 60 requests for creating policies (we suppose that there are a lot of files to share in the context of a smart traffic application);
4. 3 requests for updating policies
5. 2 requests for removing a member from a random role;
6. 2 requests for removing a random policy

For the second phase (interacting more with the enclave), the requests are distributed as follows:

1. 50 requests for making an access intent;
2. 50 requests for making an access request (namely, access to a file once the access log has been published on the blockchain and verified)

6.3 Results

Plots 6.1 and 6.2 show respectively 1° the number of valid and invalid requests (depending on the HTTP status provided by FastAPI) as well as 2° the global response time for each type of requests sent within a given amount of time.

From figure 6.1, we can see that our system works as expected (all of our requests are executed correctly) and that we have the correct number of requests specified in the previous section 6.2. We can observe that a lot of "get all roles" queries are executed. This is because this type of queries encompasses different functions related to roles: adding a role, updating a role, but also when adding a policy, etc. We can also see that the number of requests for signing a message is higher than the number of requests for launching the enclave. This is because our testing tool makes as many requests as possible even if there is no authorized access intent (because it needs to wait until the next publication of the access log file). This explains the surplus of requests for signing a message, in comparison to the requests for launching the TEE.

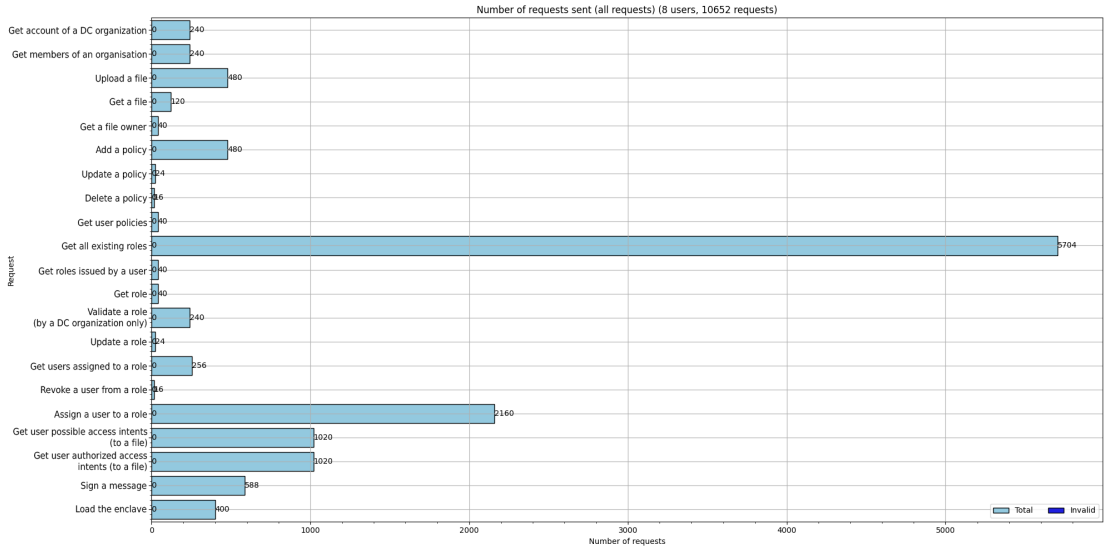

Figure 6.1: Number of requests sent in total, per type of requests.

Based on figure 6.2, we can observe and discuss several points. First, all the requests that interact with the blockchain generally take the same time (e.g. update a policy, role validation, etc.), except for the role creation which seems to be more variable. Second, it is worth mentioning that a policy creation implies 2 requests: the *creation* of a smart contract, as well as the addition of a policy. In contrast, adding a role is only the *deployment* of a smart contract. Once a DO smart

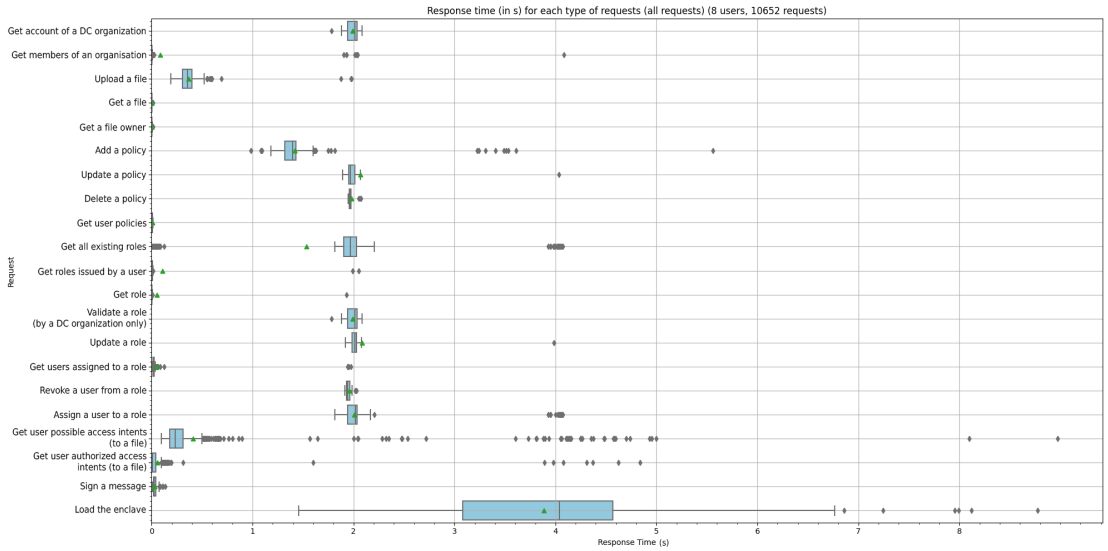

Figure 6.2: Global overview of the response time (the vertical bar represents the median and the green triangle is the mean).

contract has been deployed, it seems that adding a policy takes less time than for the first requests issued by a DO, when he/she has not yet created a policy (and hence, a smart contract). For the requests sent during the second phase of our evaluation procedure (access intents and access requests), it seems that a longer delay happens, especially for entering in the secure mode of an enclave. This could be explained by the fact that we only have a limited set of resources for our testing environment. We only have provided 2 cores per node, which in turn runs a lot of services (whereas some are computationally intensive). This delay gets worse for each request received by a DO node (responsible of enabling the secure provisioning of keys with Intel SGX), since it will create a thread associated with each connected client.

The type of requests with the longest response time is about file access. This can be explained by several reasons:

- If an authorized access intent (i.e. a waiting access request) was not recorded in the database when the DC node contacted the DO node, this latter one will sleep for 2 seconds before rechecking the content of the database. This process is repeated up to 5 times, in order to give some time for the monitor to catch up the delay that this component could have accumulated. However, this mechanism can add a considerable delay (up to 10 seconds: 5×2 seconds);

- The context switch between the normal world and the enclave is not instan-

taneous, it takes about 1 to 2 seconds to just start running a program in the enclave (furthermore, this switch is simulated by Ego Go);

- Furthermore, with this context switch, the DC node must reconnect to the DO node (each in their own enclave to ensure that no information will leak from either side) to resume where they have left off. If the DO node is congested, this reconnection can be slowed down or even result in a failed request, if the DC fails to reconnect after 10 attempts. For each failed attempt, the DC node falls asleep for 2 seconds and tries again. This mechanism can add up to 20 seconds (10×2 seconds) of delay in the worst case.

For all of these reasons, launching the enclave might be subject to a variable response time.

6.4 Discussion

Although our evaluation provides already some preliminary results, our tests should be conducted more extensively and with another testing setup than the one shown in section 6.1. Allocating 2 virtual cores, 5 GB of RAM and 20 GB of hard disk space per virtual machine, with a limited set of threads and Gunicorn workers, is not ideal for simulating a real scenario of an overloaded system. These tests should be done on more powerful machines, allowing the different services to work properly without sharing resources so that we could have more relevant results than the ones obtained in our evaluation procedure.

Moreover, we do not respect the minimal requirements of our services to work properly or their recommendations. For example, Hyperledger Besu needs 4 GB of memory (and 8 GB of memory, if running on Mainnet) [81]; Vault requires a minimum of 2 CPU cores and 8 GB of RAM [82]; IPFS recommends 2 GB of RAM and 2 CPU cores [83] and Gunicorn recommends a number of workers equal to $2\times$ the number of cores $+1$ [53]. Since all of these services have to share limited resources (on a same virtual machine), we expected these performance results.

Having to switch from one language to another one also impacts performance (we are using both Python and Go). If the solution would have been implemented totally in Go, the performance would have been increased, especially since Python is not the fastest language. If performance in terms of latency matters in a production environment, this would most likely involve rewriting our system (although here, for these tests, Python is not the bottleneck). Similarly, being able to switch in the enclave without changing the language (from Python to Go and Go to Python) would probably improve performance compared to the current situation, where each

request has to restart Ego Go. This is time-consuming and cannot be improved in the current situation due to the limitations of Ego Go: with this library, no multithreading is possible [84]. Such restriction implies that no server can run continuously, which impacts every request received and therefore the performance of the solution.

Our solution also suffers from some weaknesses in its design and some technology choices. For example, we have noticed during these stress tests that Web3.py does not support asynchronous calls (these are possible but still unstable, as stated on Web3.py documentation [85]). Furthermore, our client/server architecture uses also a synchronous socket. These both aspects do not allow to take advantage of better performances in terms of the number of requests per client and it makes difficult any attempt to overload the system with only synchronous requests (especially on virtual machines). We could send the requests and not wait for their responses, but it would not make sense for the following reasons: 1° we would not have performance results; 2° nodes will close the socket connection if a client did not connect for 60 seconds, which puts incomplete pressure on the system (the process will sleep most of the time). We could have done a better architectural choice, which is more loosely coupled. Asynchronous and micro-service-oriented architecture would have been more easily scalable and would provide more independence between the different services. However, it would have required an almost complete rewriting of our solution, even if the logic would have not changed.

Our solution was a trade-off between ease of development, security and performance, although being still a proof of concept (or prototype). Furthermore, implementing security is at the expense of performance, or adding security constraints increases the complexity of the system. In our use case, we had a lot of such constraints.

Chapter 7

Related Work

This chapter discusses the state of the art. It presents in sections 7.1 and 7.2 where the security concepts and mechanisms we use in our solution come from or how they are related to other concepts. Section 7.3 highlights a few representative works addressing a *similar* problem to ours. Comparing our system with those should be enough to understand the place our system occupies within the literature. However, it is worth noting that no existing solution solves the *same* problem as ours, that is implementing a reliable audit trail and ensuring confidential data sharing in the context of a distributed environment with no trusted party and reduced financial costs. To the best of our knowledge, no existing solution does match exactly our requirements. This makes our solution unique, putting together several known security mechanisms or technologies described in previous chapters 4 and 5.

7.1 Access Control, Access Logging and Security Policies

Access control refers to the "process of mediating every request to resources and data maintained by a system and determining whether the request should be granted or denied" [15]. Access control is generally enforced by a mechanism implementing regulations established by a security policy. Clark and Wilson [4] are among the first ones to distinguish security policies into 2 distinct sets: those related to disclosure controls enforcement and those related to integrity controls enforcement. In the first case, the system must protect from authorized as well as unauthorized users, for preventing information from being disclosed to unauthorized individuals. For the second type of security policies, the major goal is less controlling the confidentiality of the information than ensuring its integrity, by preventing fraud and errors. Clark and Wilson highlight 2 mechanisms at the heart of fraud and error control: well-formed transaction and separation of duty. Well-formed transaction

refers to the control of data processing by a user: he/she should not manipulate data arbitrarily, but only in constrained ways that preserve or ensure the integrity of data. A common mechanism for achieving that is to record all data modifications in a log (and more precisely, access log) so that actions can be audited later. Separation of duty refers to the sharing of responsibilities by several entities or users.

Access control, access logging and security policies are thus concepts tight together. Traditionally, 3 main classes of security policies (or access control policies) are defined [15], [17]:

1. Mandatory access control (MAC): MAC policies are based on (mandated) regulations defined by a central authority;

2. Discretionary access control (DAC): DAC policies are based on the identity of the requestor as well as on access rules which state what requestors are allowed to do (or not);

3. Role-based access control (RBAC): RBAC policies are based on roles that users have within the system and on rules telling what is allowed to do for members assigned to a given role;

Besides these 3 main classes, access control can also be enforced through attribute-based encryption (ABE) technology. This latter one refers to an asymmetric cryptography technique in which the secret key of a user as well as the ciphertext are dependent on attributes (e.g. for which organization an employee works, what is his/her IP address, etc.). Thus, the decryption of a ciphertext is possible only if the set of attributes of the user key matches the attributes of the ciphertext. Some notable works in this field include Goyal et al. for a presentation of ABE used in cloud environment [86] and Lewko et al. [87] for designing a multi-authority attribute-based encryption (MA-ABE) scheme, where any party can generate and issue key pairs to users by becoming an authority.

In our system, as stated in the previous sections, we are using an RBAC mechanism, as described by Sandhu et al. and first defined by UNIX `groups` [16]: users are assigned to roles, permissions are assigned to roles and users acquire permissions by being members of roles. We cannot use MAC due to our trust model, nor DAC for preserving the privacy of the users, by hiding their respective identities behind account addresses or public keys (especially on a public permissionless blockchain network). RBAC mechanism is implemented within our solution through the use of smart contracts. This idea was taken from Cruz et al., which define RBAC-SC and a challenge/response authentication protocol [72]. The key ideas behind RBAC-SC are to publish all relevant information about user-role assignment in a deployed

smart contract and to employ a challenge/response protocol for checking whether a user is assigned to a given role or not. The smart contract contains several functions about the assignment of a user to a role or his/her revocation from a role, etc. while the challenge/response protocol proceeds as follows:

1. The organization requests the public key of the user claiming to be assigned to a role;
2. It verifies then if the given key is well linked to the required role by checking a public medium (here, the distributed ledger);
3. If this key is linked to the role, the organization requests the user to sign a given message (a "challenge") with its private key (corresponding to that public key), to be sure that he/she owns this public key;
4. The organization checks the response sent by the user and if it is valid, the user can access the service/data

As stated in chapter 5, we use the same approach in our solution (with some additional verifications). In both cases, however, private key leakage is not handled.

7.2 On-chain and Off-chain System Designs

Traditionally, blockchain systems store data content locally on blockchain nodes, i.e. on-chain. However, this approach has several drawbacks: first, the limited size of blocks on a blockchain makes it difficult to store more complex data other than hashes, transaction history, state of data or registry entries. Second, the transaction rate is also limited by the network's throughput. Third, the average block creation time is also predefined and limited. For example, the current block size of Bitcoin blockchain network is currently set at 1MB [46] [47], with a transaction processing capacity of 3 to 7 transactions per second for typical transaction sizes (this transaction rate was about 1 to 3.5 transactions per second as of 2016 [46]). Concerning the average block creation time or block interval, Bitcoin currently targets a conservative 10 minutes between blocks, thus there are 10 minutes expected latencies for transactions before being encoded in the blockchain. Besides these aspects related to the scalability problem of blockchains, storing data on-chain also means higher operating costs, as a user has to pay transaction fees for sending a transaction to the network.

Off-chaining has been proposed as one approach to address these limitations [42]. The idea is to reduce data storage as well as computational efforts on the

blockchain by using blockchain-external resources, without compromising availability and immutability, which are the key properties introduced by blockchain technology. Eberhardt et al. [42] provide a categorization of different types of off-chaining models, as well as reference pointers to applications of each of them:

- off-chain storage: Blockchain data content is stored on an off-chain node, i.e. an arbitrary node which is not necessarily part of the blockchain network;
- off-chain computation: Blockchain's state transition function is computed by an off-chain node before persisting the resulting state persists on-chain, once the computation has been verified. Blockchains can indeed be considered as global state machines that process transactions leading to state transitions;
- hybrid off-chaining: This type of off-chaining models refer to the set of designs that combine off-chain state and off-chain computations in arbitrary ways, potentially in conjunction with on-chain processing

In our case, we are using both off-chain storage via the use of IPFS and off-chain computation via the use of Intel SGX as enclave-based off-chain computation. Off-chain storage can itself be categorized into three main groups: hardware storage (or direct attached storage, DAS), centralized cloud storage and distributed storage. IPFS falls into this last category (and is one of its main representatives): information is stored on multiple nodes on a decentralized network and is addressed by its content, usually by its hash. By doing so, data integrity is preserved because a data object's address can be recalculated by applying a hash function to the data received and it can be compared to its on-chain reference. Such an approach can also be combined with economic incentives (e.g. Filecoin [88]) to enhance the data availability. In the next section 7.3, we will present and discuss some solutions which leverage off-chain storage. A comparison of existing distributed file systems can also be found in Appendix F.

There also exist several approaches under the expression "off-chain computation" [42]: verifiable off-chain computation, off-chain computation using trusted execution environment (TEE), secure multiparty computation-based (SMPC) off-chaining and incentive-driven off-chain computing (IOC). The first one refers to a technique where a *Prover* executes a computation and publishes its result as well as a cryptographic proof attesting its computation's correctness to the blockchain. Then, an on-chain *Verifier* verifies the proof and persists the result if the verification is valid. Off-chain computation based on TEE stores on-chain a new state after this one has been computed exclusively inside a trusted enclave whose attestation has been verified. SMPCs enable a set of nodes to compute functions on secret data such that none of the nodes has access to the data in its entirety. Lastly, IOC models

leverage incentive mechanisms to motivate off-chain computation and guarantee its computational correctness: while computational tasks are executed by off-chain *Solvers*, *Verifiers* are incited to find errors in their solutions. The *Judge* (i.e. the blockchain network itself) decides who is right between a *Solver* and a *Verifier* and thus who can obtain a reward.

AuditTrust is relying on off-chain computation using TEE, but this is not the first attempt to combine blockchain and TEE. There already exist numerous works in the literature that adapt hardware-assisted TEE as a tool to support blockchain-based applications: Ekiden [45] offloads computation from blockchain nodes to a collection of computing nodes, so that the blockchain is solely used as persistent state storage. By doing this separation between computation and consensus, Ekiden achieves greater performance than Ethereum's Mainnet, while preserving privacy-preserving computation with the use of TEE. IRON [89] and Ryoan [90] are also other works combining blockchain and TEE. In this latter case, information flow is defined by a directed acyclic graph, rather than by the use of smart contracts. Lastly, Intel has also introduced in 2018 a solution called Private Data Objects (PDOs) [91], based on TEE and a distributed ledger (Hyperleder Sawtooth). PDOs execute smart contract functions off-chain (within an Intel SGX enclave), while the outputs are stored on the blockchain. A comparison of existing hardware solutions for data-in-use protection can also be found in Appendix B.

7.3 Blockchain-based Data Usage Control with Off-Chain Models

We will present in this section a few close existing blockchain-based applications to our solution: PrivacyGuard (subsection 7.3.1), ProvChain (subsection 7.3.2), hOCBS (subsection 7.3.3) and a framework based on ABE (subsection 7.3.4). They all have in common that they match our criteria only *partially*, as we will show it now.

7.3.1 PrivacyGuard: Smart Contracts and Trusted Execution Environment in Cloud

With PrivacyGuard [3], Xiao et al. have developed the most similar system to ours. PrivacyGuard is based on Ethereum and Intel SGX. It leverages blockchain smart contracts and TEE for enforcing individuals' control over the access and use of their data in cloud environment. PrivacyGuard aims to provide privacy protection techniques for covering 1° what kind of information can be exposed to

whom, under what conditions and 2° what is the intended purpose or expected use of this information. For addressing both of these privacy aspects, PrivacyGuard's architecture is made of 2 domains, the control plane and the data plane:

1. The control plane is related to blockchain interaction: in this plane, individual users (data owners or DO) publish on blockchain the availability as well as the usage policy of their data as smart contracts. Users who want to use the data (data consumers or DC) interact with these smart contracts to gain authorization to access the data.

2. The data plane is related to the use of a TEE on the cloud: in PrivacyGuard, the DO's data are not exposed on the blockchain but are stored off-chain, in encrypted forms in the cloud. Once a DC is authorized to access data, computation on those latter ones as well as the provision of secret keys for decrypting them are accomplished with Intel SGX as TEE. Then, only the computation result is accessible by the authorized user (i.e. the plaintext source data is not available to him/her).

Similarly to us, there are 2 types of actors in PrivacyGuard: data owners (DO) and data consumers (DC). PrivacyGuard's workflow proceeds in a very similar way to AuditTrust: a DO shares data stored under an encrypted form in a cloud storage, before defining its usage policies as well as a requested price, via a smart contract. DC invokes this smart contract for asking permission to use these data, and a TEE enclave program will be loaded if the permission is granted on the blockchain. In this enclave, DO's data encryption and decryption keys are provisioned. However, PrivacyGuard is also relying on a *trusted* delegate (a data broker, DB) to represent a group of DOs. This one is also used for attesting the enclave on behalf of all relevant DOs, instead of having each DO attesting and verifying the TEE enclave individually. Besides the fact that a DB allows multiple owners of data, Xiao et al. motivate the use of DB for optimization purposes: 1° to avoid the limited transaction throughput in public blockchains, as well as overhead and network traffic; 2° to minimize the costs of deploying one smart contract per DO. However, these justifications imply that PrivacyGuard's transactions are mainly DO's transactions (that are issuing and deploying smart contracts for sharing their private data). In our case, we assume that the majority of blockchain transactions are asking for access to data. This is the reason why we do not send a transaction immediately after a DC has requested access to a file, but we have implemented an access log file mechanism aggregating these requests, whose availability is published on the blockchain after a predefined time interval. Thus, unlike Xiao et al., we do not use a trusted DB for DO, as there is no need for such component in our system (a file is owned only by one DO); rather we use a mechanism similar to DB for

DCs via the smart contract `DataConsumerOrganisation`: in our system, members of the same organization are assumed to be trusted and thus, we can aggregate their respective access intents also for optimization purposes. Consequently, our setting-up phase (roles assignment, sharing data via policies) will have a higher cost for sending the different transactions, but in a long run, the number of transactions sent will be far more limited and therefore less costly from a financial perspective than PrivacyGuard.

For ensuring the security of their solution, Xiao et al. also tackle the problem of the atomic completion of a smart contract by designing a multistep commitment protocol, aborting a transaction if the computation has been tampered with; they have also added a timeout mechanism for preventing any paused or delayed computation by an adversary taking the control of the TEE. These mechanisms were not implemented in our prototype.

Lastly, PrivacyGuard does not implement an audit tool (the distributed ledger is considered sufficient enough) and does not use a distributed file system like ours. This solution is thus simpler to be implemented than ours, but the use case, although very similar, is not the *same*.

7.3.2 ProvChain: Blockchain-based Audit of Real-Time Monitored User Activities

Xueping et al. have developed an architecture called ProvChain for collecting and verifying cloud data provenance, that is the history of the creation and operations performed on a cloud data object [6]. The authors are collecting provenance data in real-time via a monitoring system, before embedding them into blockchain transactions and a local database. These provenance data are lastly validated via a provenance auditor (PA). Thus, ProvChain operates in 3 phases: 1° provenance data collection; 2° provenance data storage and 3° provenance data validation.

The system proceeds as follows: a cloud user connects to a cloud service provider and performs some operations on files (stored in the cloud). As any file operation is monitored (here: via a system of hooks and listeners built on top of ownCloud [92]), every user activity related to a file will be collected and recorded (*provenance data collection*). This data record (or provenance data) will be uploaded (under a hashed form, following Chainpoint's standard [93]) to the blockchain network via Tierion API [94] as well as stored in a local database (*provenance data storage*). This latter one (called "provenance database") will be used for further updates and validation: the service provider can assign a (trusted) provenance auditor (PA) to

verify the data from the blockchain network. PA will request a blockchain receipt (via Tierion API) to validate any transaction containing provenance data. Each blockchain receipt contains hashed versions of data records which can be used together for building a Merkle tree, as well as a Merkle root value. The validation step consists thus in building the Merkle tree and comparing its root with the one provided.

Like PrivacyGuard and our system, ProvChain relies on blockchain for ensuring data usage control. But contrary to PrivacyGuard, there is a local provenance database as well as a provenance auditor which validates its content. Moreover, contrary to PrivacyGuard and our system, ProvChain does not use TEE for sharing keys. In ProvChain, numerous keys are used for 1° user registration at the cloud service provider; 2° encrypting and decrypting the data stored in the cloud; 3° proving a user data ownership with a public/private key pair and 4° provenance data encryption. In this latter case, the key is shared with the assigned PA for decrypting the data records. However, the authors do not specify how this process occurs. Additionally, this system cannot use smart contracts due to implementation limitations (Tierion API combined with Chainpoint is only compatible with Bitcoin network [93]).

However, there are also some similarities between our system and ProvChain: AuditTrust's blockchain monitoring system is similar to the mechanism (involving hooks and listeners) used by ProvChain for collecting provenance data. Though, in this latter case, it relies heavily on a specific cloud platform feature, ownCloud's hooks, restricting its scope of application. In our case, we are using a data structure present inherently in blockchain networks, a transaction log, to track any user activity by decoding the event emitted in this log. Besides, ownCloud is a centralized cloud platform, while our solution works in a distributed manner (with IPFS).

Like ProvChain (but unlike PrivacyGuard), we also use local databases (one per organization) for storing information about data usage and for replicating (partially) blockchain content. Similar to ProvChain, the content of these local databases will be compared to the content of the blockchain when performing an audit (via our audit tool). However, a major difference is that ProvChain relies on a trusted provenance auditor for ensuring the data provenance, designated by the service provider. It means that this is not an automatic process and that it requires a human operator (a user). In our case, this is done automatically by the organization itself, because there is no trusted third party in AuditTrust.

Lastly, the authors do not talk about the possible financial cost of their solu-

tion, but we can imagine that it will increase rapidly if each user activity generates a data record sent on the blockchain network. This is not acceptable in our use case, as it could have a lot of user operations at the same time.

7.3.3 hOCBS: Off-chain Blockchain Systems for Healthcare Data

Miyachi and Mackey's work [7] shows how to leverage blockchain technology in the context of healthcare applications for facilitating the transition toward patient-driven interoperability. Miyachi and Mackey conceptualize hOCBS [7], a blockchain system interacting with off-chain resources, and translate it for being suited to 3 different types of healthcare data, resulting in 3 reference models, based on their respective needs, regulatory compliance, redundancy and security requirements.

General architecture of hOCBS includes smart contracts, digital wallets and tokens. Miyachi and Mackey are using smart contracts for controlling access privileges, using predefined roles for validating digital identity (similarly to Cruz et al.'s RBAC-SC solution [72]); digital wallets are used for holding verifiable credentials about the identity of a user and enabling digital signatures; lastly, tokens are used to encourage certain behavior on the network (e.g. via incentives). As a general approach, hOCBS also assumes that on-chain storage of each reference model will only act as a large access log and provide linkage to off-chain data. Each model also uses a proof of authority (PoA) consensus.

After an extensive study of off-chain models (distinguishing off-chain storage, off-chain computation and hybrid off-chaining, see also Eberhadt and Heiss [42]) and the existing solutions implementing them, Miyachi and Mackey introduce in their paper [7] 3 distinct models, based on 3 types of healthcare data:

- protected health information (PHI): this model is the most restrictive one, as it is governed by permission rules defined by the patient and any underlying legal agreements, all executed through smart contracts logic. Data linkage from on-chain to off-chain resources can be done within TEEs, while the off-chain storage remains centralized for maintaining regulatory compliance;
- consumer health information (CHI): this model focuses on the control, management and sharing of *consumer* information, subject to less stringent requirements of privacy frameworks, such as GDPR. Here, the key feature of the architecture is privacy and the user has to enable explicitly data sharing. Types of data are more various than for PHI (focused mainly on electronic health records or EHR). In this case, the authors choose IPFS for off-chain

storage management because it enables a fault-tolerant, scalable and highly available file system, whereas TEE is still used for executing any modification to IPFS. CHI data can be retrieved in exchange for token payments, with the consumer deciding what underlining data or metadata he/she wants to share per smart contract terms;

- genomic data: this type of data is characterized by its large volume which needs to be analyzed. Thus, the model related to this type of data should facilitate off-chain computation-intensive analysis

Although our use case is not related to healthcare data and although hOCBS is only a theoretical blockchain system, there are some similarities with these 3 models, and in particular with the CHI's one: we are also using TEE for the data linkage from on-chain to off-chain resources, we also use IPFS as distributed file storage for improving the security of our system as well as data availability and lastly, we have also written some smart contracts for defining in which conditions and to whom data are shared, while preserving the confidentiality of these data. However, in hOCBS, the authors do not have the same trust issue as ours and therefore can rely on a private permissioned blockchain based on identity validation, with a predefined number of nodes acting as authoritative nodes (according to the PoA consensus protocol). hOCBS can also use tokens for adding an incentive layer. We cannot do this in AuditTrust, because it would not be an economically viable solution if a DC had to pay for each access requested to DO's data, given the fact that they can send a lot of requests.

7.3.4 Blockchain-based Framework with Attribute-based Encryption

In the previous cited works [3] and [7] (see 7.3.1 and 7.3.3), access control was defined by rules implemented in smart contracts logic. Before concluding this section, let us present another way of defining granular access control in a blockchain-based system, relying on attribute-based encryption technology (ABE).

In their work [8], Wang et al. are describing a blockchain-based framework that combines IPFS, Ethereum blockchain and ABE. 2 types of actors are recognized within this framework, as in our solution: data owners (DOs) and data users (DUs). The workflow of data sharing proceeds as follows: the DO distributes a secret key for DU and encrypts shared data by specifying an access policy, based on ABE: only the users whose attributes set meet the access policy can access the data. At the same time, a (data sharing) smart contract is deployed for implementing the keyword search function on the ciphertext (i.e. the encrypted data stored on

IPFS), to search data based on interesting keywords to which these are assigned. Information about this smart contract is transmitted to DU. The latter can use it to search relevant transaction data (containing IPFS hashes) based on search results returned by this smart contract. It is worth noting that a DU can only perform a successful search if his/her attributes are the ones defined by the access policy. In this case, it can retrieve the adequate key for generating a search token and invoking the smart contract.

This solution is also similar to AuditTrust in the choice of used technologies (Ethereum and smart contracts, IPFS). However, the authors do not specify how information about a smart contract is disclosed to DU: they only mention that these are communicated through a secure channel. Their solution also suffers from another drawback, related to the use of ABE: if the user's attributes had to change, its corresponding secret key must also be changed accordingly. It means that this solution is not as dynamic as ours: by generating a key for a role rather than an attribute, we do not encounter this problem of revocation. Lastly, Wang et al.'s solution is costly: each time a DU wants to invoke the search function of a data sharing smart contract, he/she has to pay, but also when a DO uploads files to IPFS, because he/she has to select a keyword set from each file for storing the corresponding keyword index in the smart contract. For all of these reasons, this framework could not match our criteria as well.

Chapter 8

Conclusion and Future Work

Sharing data and controlling its usage in a distributed context when the participants are mutually mistrusting entities is a real challenge: defining how these data can be processed through strict access control rules, and being able to audit its usage transparently, with confidentiality and integrity of data preserved, in a distributed environment where participants do not trust each other at all, is not a trivial problem.

This master thesis addresses these challenges, by designing and prototyping a solution called AuditTrust based on innovative technologies such as blockchain, distributed storage, trusted execution environment (TEE) and modern cryptography mechanisms. This prototype allows data owners (DOs) and data consumers (DCs) respectively to share data and to request access to it, alongside the opportunities to check fraudulent accesses within the system or for a given (shared) file.

This prototype is aimed to work later on a public permissionless blockchain network (Ethereum's Mainnet), once in production, for allowing each participant to verify independently and transparently data usage, while being currently on a private permissioned blockchain network for development purposes (Hyperledger Besu). As they will be connected to a network with other activities than the ones of the system, DOs and DCs also maintain a local PostgreSQL database replicating (partially) the content of the blockchain through a blockchain monitoring system. Due to several blockchain limitations, DOs and DCs cannot rely on the blockchain for storing their data (shared images and videos, access logs). Hence, they are using a distributed storage system, IPFS. Because it is a content-based system, IPFS guarantees that data will not be tampered with without being detected: it produces a content identifier (or hash) which is defined by the content of a file: if this one was to be modified, its identifier would have also changed.

However, data confidentiality must also be preserved, hence our prototype relies heavily on symmetric and asymmetric encryption to address this concern. The keys used for this purpose must however be shared with the other participants, so that they can decrypt the data. Due to our strict requirement of no trust at all between the participants, we cannot even trust their machines. To be sure that secrets cannot be leaked, even by the memory, we use a trusted executed environment compatible with Intel processors, namely Intel SGX. Furthermore, as keys are crucial resources, they should be stored securely. To this aim, we are using a key management service, Vault.

These components put together, alongside verification protocols for authentication and authorization purposes (such as the implementation of a challenge/response protocol), defines our solution, called AuditTrust. As we have seen in the evaluation, this prototype works well even with restrained resources and a large number of services running in parallel. However, it has also exposed some weaknesses in design choices. As future work, we plan to add a watermarking process for detecting any leakage of shared data: currently, in our system, nothing prevents a DC to record an image or a video through his/her smartphone or a recording application on his/her laptop. This watermarking process should happen in the enclave during the decryption of the shared file. Our solution does not protect either from collusion: a DC might share the private key of his/her Ethereum account with another DC, so that this latter one might pretend to be the first one and sends transactions on his/her behalf. A first naive solution could be to monitor the IP addresses from which a user is usually connected to detect anomalous activity. A better solution is to use digital wallets. These are hardware or software key vaults that can be used for limiting as much as possible the untrusted storage of sensitive information by the end-users. An integration with MetaMask [95] for example should be preferred; however, it would require rewriting a part of our solution (the frontend) to enable the interactions between the digital wallet and our services, without the use of an untrusted server. Other programming languages should be preferred as well as the ones chosen for this prototype if the performance is a critical point. Additionally, a clearly defined micro-services architecture is also a better way to manage our distributed application in the future.

Our off-chain storage could also be improved for taking into account mutable data and the read and write operation, which was not implemented in our prototype: we could use IPFS' InterPlanetary Name System (IPNS) in a future version of our system for keeping an immutable alias for files being updated. However, another question arises about the key to using for encrypting this file (do we use the same as for the original file? Do we generate another key? If so, how do we handle

the key exchange? etc.). Another promising way (but still in development) would be to use a publisher-subscriber mechanism [96]. Another point to mention is that IPFS does not implement authentication or access control. Every node that has joined the IPFS network could retrieve any file (although encrypted) via its content identifier. Other mechanisms should be put in place to limit the interactions of the participating nodes. Lastly, the encryption algorithms used for this prototype are not quantum resistant. This could be also another way of improving AuditTrust's security. However, as always, there is a trade-off between security and performance.

Appendix A

Life Cycle of Smart Contracts

The life cycle of smart contracts is the following [41]:

1. creation: similarly to a transaction, a smart contract is defined as an interaction between two parties. Similarly to a physical contract, a smart contract must define contractual clauses to which both parties must agree. These contractual clauses are written with a programming language, before being compiled. The output of this compilation (called machine code or bytecode) is then used for the deployment of the smart contract;

2. deployment: the machine code containing the smart contract will be deployed to the blockchain network. Once a smart contract has been deployed, it is impossible to make any revisions: it is immutable and therefore should be bug-free;

3. execution: once the contractual conditions reach, the contractual procedures will be automatically executed. The transaction containing this execution will be broadcasted to the blockchain network, before being validated. During this time, another smart contract cannot be executed;

4. completion: once validated, the transaction is appended to a list of transactions to form the next block of the blockchain. Once the current contract is completed, another contract can be executed

Appendix B

Comparison Between TEEs

	Intel SGX	**AMD (SME, TSME, SEV)**	**ARM TrustZone**
Required Hardware	Mainly for Intel Xeon CPUs (business purposes) Desktop version but support ended (only from 6th to 11th CPU generations)	Available for pro and business CPUs only (except for TSME)	Broader availability (compatible with low-end devices such as Raspberry Pi) but only for Cortex-A and Cortex-M CPUs
Enclave Memory Size	At first 256 MB, 512 GB up to 1 TB for newer generations (from Intel SGX v3)	Same as RAM for AMD SEV	depending of the device, for Raspberry Pi 3B, it is limited to 16MB [97]
Memory Integrity Protection [98]	yes	no	
Software Development	Many libraries, with 2 main ones: Open Enclave [74] and Ego Go [75]		Only OP-TEE library [73] for Raspberry Pi

Table B.1: Comparison of the most popular TEEs.

Appendix C

Do We Need a Blockchain?

Wüst and Gervais have proposed a structured methodology to determine the appropriate technical solution to solve a particular application problem, and whether or not using blockchain is indeed the adequate technical solution for a particular use case [37]. According to the authors, a blockchain is not needed if at least one of the following conditions are met:

- no data needs to be stored;
- if there is only one writer (where a writer refers to the first kind of peers mentioned above, i.e. an entity which modifies the state of the system);
- if there is a trusted third party which is always available online (in this case, write operations can be delegated to it)

The authors ask the following questions to determine which technical solution is the most adapted for a problem:

1. do we need to store state?
2. are there multiple writers?
3. can we use an always online trusted third party?
4. are all writers known? if so, are all writers trusted? In the case of some writers known that are not trusted, is public verifiability required?

In our case, information needs to be stored (when an organization adds a role or a policy, update them, when a member asks for access to a file, when a problem during the audit has occurred, etc.). We cannot rely on a single writer because organizations mutually mistrust each other. Consequently, multiple peers can

modify the state of the system (and at least one per organization). Due to this strong requirement of no trust between the participants, AuditTrust cannot also rely on a trusted third party. Lastly, this requirement also implies that there is no governance or consortium inside the network: nobody can act as a certificate authority and decide who can join the network and modify its state; rather, the number of writers is not known in advance.

Appendix D

Comparison of Blockchain Platforms

For development purposes, we are not restricted to specific types of blockchain networks (permissionless, permissioned). Thus, our criteria for choosing a blockchain platform are related to the ease of development: active development of the project, its maturity, its cost, if it provides well-furnished documentation and lastly, if we can reuse some parts of it for production purposes (whereas Ethereum blockchain platform was chosen). We have eventually also chosen Ethereum platform for our prototype, for its ease of development and the fact that it can also be used for production purposes (for a public permissionless blockchain network). Several solutions are discussed here: Hyperledger Fabric [34], Hyperledger Sawtooth [33], BigchainDB [99], Cardano [35] and Chainlink [100].

Hyperledger Fabric was not selected as there was too little documentation for the installation of such a platform. Hyperledger Sawtooth was also rejected due to its integration of Ethereum smart contracts which is not actively maintained. Another alternative was to write smart contracts in WebAssembly, but we are not familiar with this language. The last alternative would have to write smart contracts with transaction families, but we would have been dependent on the Hyperledger platform, which was not retained for the production solution (we are looking for a permissionless blockchain). BigchainDB does not really support smart contracts and is not intended to do so (Smart-Contracts are replaced by conditions in this system) [101]. Cardano was another possible choice, but the documentation compared to Ethereum was limited, thus we preferred to choose the most documented platform. Lastly, for Chainlink, a third-party service (oracle) is needed to connect smart contracts with off-chain data. Based on our trust model, we cannot rely on such a solution, especially in production. Thus developing on such a platform would need to rewrite our solution for the production, which is something we want to avoid.

Regarding the blockchain platform used for the production, we are constrained to permissionless blockchains for our use case, as stated by Wüst and Gervais [37]. We can only choose blockchain platforms that are public and permissionless, with a lot of nodes in order to prevent Sybil attacks or 51% attacks. The only valuable solutions are Bitcoin [9], Ethereum [27] and Cardano [35]. Our choice of the Ethereum platform is based on its well-furnished documentation, its popularity and its maturity. Moreover, given that Ethereum 2.0 should be out in the future, this could lower drastically transaction fees. Besides Ethereum, we did not choose Bitcoin as its main goal is to transfer digital money. Bitcoin is also slower in the production of blocks compared to Ethereum and Cardano is not compatible with EVM smart contracts. While Cardano is also a public permissionless blockchain, we would have to rewrite entirely what we would have implemented for development purposes (prototype), as Cardano does not provide any private blockchain to develop with.

Appendix E

Smart Contracts Events

Here is the list of the events emitted by our different smart contracts. Given that any contract inherits from a parent contract called `BaseContract`, `BaseContract`'s events will also be available for the other smart contracts (`DataOwner`, `DataConsumer` and `DataConsumerOrganisation`):

- `previousContractAddressChanged` (in `BaseContract`) indicates that a smart contract is linked to a new one;
- `activeContract` (in `BaseContract`) emits the status of a smart contract (active/inactive), if it is active or inactive (disabled);
- `policyAdded` (in `DataOwner`) indicates that a new policy has been added in the list of policies related to a particular DO;
- `policyUpdated` (in `DataOwner`) indicates that a policy related to a DO is updated;
- `policyRemoved` (in `DataOwner`) emits that a policy related to a DO is removed;
- `roleNameChanged` (in `DataConsumer`) emits that the rolename of a given role is updated;
- `dataConsumerAssigned` (in `DataConsumer`) emits that a new DC member is added to a given role;
- `dataConsumerRevoked` (in `DataConsumer`) indicates that a DC member is not assigned anymore to a given role;
- `newRole` (in `DataConsumer`) indicates that a new role is issued by a DC member;

- `newOrganisation` (in `DataConsumerOrganisation`) emits that a new organisation is deployed;
- `newOrganisationRole` (in `DataConsumerOrganisation`) indicates that the role whose smart contract address was provided is validated by the organization;
- `newAccessLog` (in `DataConsumerOrganisation`) emits that a new file containing the access logs is published on the distributed file system

Appendix F

Comparison of Distributed Storage Solutions

The current chapter provides a comparison of IPFS [69], Swarm [70] and Hypercore Protocol [102]. The two first ones are the most popular options, while the third, although being less popular, is also worth mentioning, as it matches our criteria for this project: we are looking for a solution that can be combined with blockchain technology (i.e. a decentralized solution), which is fast and not expensive. For this last reason, we will not discuss other solutions such as Storj [103] or Arweave [104], because some fees are applied for these solutions when files are stored and/or retrieved (which happen regularly in our case).

- IPFS is a distributed system for storing and accessing files, websites and applications, released in 2015 by Protocol Labs. IPFS retrieves a file (or other types of data) by its content, not its location (content-based addressing). Each object stored in IPFS is split into chunks of blocks, where each block is identified by a content identifier (CID or IPFS hash). For keeping track of these chunks, IPFS uses a Merkle tree and each file can be accessed by the hash of its content. Consequently, once a file is modified, new chunks of blocks are generated, which means that there is also a new Merkle tree, allowing file versioning. IPFS does not have active replication: a file is stored on the IPFS node closest to the CID, following a distributed hash table (DHT). Other IPFS nodes can replicate it by a pinning mechanism, that allows them to store locally the data pinned. Thus, for deleting a file on IPFS, it needs to be removed from every pinning node.

- Swarm also provides decentralized storage based on content-based addressing, but this solution relies more on Ethereum and is more linked to incentives than IPFS [71]. Swarm also adds the concept of "area of responsibility", for

letting the content-based addressing decides the storage location of a file. As for IPFS, a file is also split into chunks arranged in a Merkle tree, but unlike IPFS, content cannot be deleted. Swarm also provides an access control feature thanks to manifests. IPFS does not have such an equivalent, although encryption can be used for protecting data from unauthorized access.

- Hypercore Protocol is a distributed data network built on Hypercore logs, consisting of multiple sub-components (Hypercore, Hyperdrive, Hyperswarm, Hyperspace API, etc.). These projects aim to provide a "lightweight blockchain crossed with BitTorrent" [105] and are designed for sharing data and supporting incremental versioning of the content (and metadata). Unlike IPFS and Swarm, Hypercore does not use primarily content-based addressing. This one is used when sharing a file partially [71]. Otherwise, files are stored within directories, where each directory is dealt with its own peer-to-peer network. Hypercore provides a subscription mechanism to be notified of the last changes within a directory and each node can decide which data and which version it wants to store.

Appendix G

Structure of an Access Log

Here is a fictive example of the structure of an access log:

```
{
    Public key DO 1: {
        Encrypted access intent 1 - DC's signature,
        Encrypted access intent 2 - DC's signature,
        Encrypted access intent 3 - DC's signature,
        ...
    },
    Public key DO 2: {
        Encrypted access intent 1 - DC's signature,
        Encrypted access intent 2 - DC's signature,
        Encrypted access intent 3 - DC's signature,
        ...
    },
    ...
}
```

Each access intent (once decrypted) contains the following information (in this order):

- date and time of the access
- DC's Ethereum account address
- IPFS hash (CID) of the requested file
- permission asked

A signature contains all of this information separated by a comma and signed with the DC's private key.

Bibliography

[1] R. J. T. Morris and B. J. Truskowski, "The evolution of storage systems," vol. 42, no. 2, pp. 205–217. [Online]. Available: https://doi.org/10.1147/sj.422.0205

[2] S. Wachter, "Normative challenges of identification in the internet of things: Privacy, profiling, discrimination, and the GDPR," vol. 34, no. 3, pp. 436–449. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0267364917303904

[3] Y. Xiao, N. Zhang, J. Li, W. Lou, and Y. T. Hou, "Privacyguard: Enforcing private data usage control with blockchain and attested off-chain contract execution," in *European Symposium on Research in Computer Security*. Springer, 2020, pp. 610–629.

[4] D. D. Clark and D. R. Wilson, "A comparison of commercial and military computer security policies," in *1987 IEEE Symposium on Security and Privacy*, pp. 184–184, ISSN: 1540-7993.

[5] A. Ahmad, M. Saad, M. Bassiouni, and A. Mohaisen, "Towards blockchain-driven, secure and transparent audit logs," in *Proceedings of the 15th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, ser. MobiQuitous '18. Association for Computing Machinery, pp. 443–448. [Online]. Available: https://doi.org/10.1145/3286978.3286985

[6] X. Liang, S. Shetty, D. Tosh, C. Kamhoua, K. Kwiat, and L. Njilla, "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.

[7] K. Miyachi and T. K. Mackey, "hocbs: A privacy-preserving blockchain framework for healthcare data leveraging an on-chain and off-chain system

design," *Information Processing & Management*, vol. 58, no. 3, p. 102535, 2021.

[8] S. Wang, Y. Zhang, and Y. Zhang, "A blockchain-based framework for data sharing with fine-grained access control in decentralized storage systems," vol. 6, pp. 38 437–38 450, conference Name: IEEE Access.

[9] Bitcoin - open source p2p money. [Online]. Available: https://bitcoin.org/en/

[10] "Ethereum whitepaper." [Online]. Available: https://ethereum.org/en/whitepaper/

[11] T.-T. Kuo, H.-E. Kim, and L. Ohno-Machado, "Blockchain distributed ledger technologies for biomedical and health care applications," vol. 24, no. 6, pp. 1211–1220. [Online]. Available: https://doi.org/10.1093/jamia/ocx068

[12] Towards blockchain-based auditable storage and sharing of IoT data | proceedings of the 2017 on cloud computing security workshop. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/3140649.3140656

[13] Icity feder project. [Online]. Available: https://icity.brussels/

[14] P. Papadimitriou and H. Garcia-Molina, "A model for data leakage detection," in *2009 IEEE 25th International Conference on Data Engineering*, pp. 1307–1310, ISSN: 2375-026X.

[15] P. Samarati and S. C. de Vimercati, "Access control: Policies, models, and mechanisms," in *Foundations of Security Analysis and Design*, ser. Lecture Notes in Computer Science, R. Focardi and R. Gorrieri, Eds. Springer, pp. 137–196.

[16] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-based access control models yz," p. 22.

[17] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli, "Proposed NIST standard for role-based access control," vol. 4, no. 3, pp. 224–274. [Online]. Available: https://doi.org/10.1145/501978.501980

[18] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*. CRC Press, google-Books-ID: RsoOEAAAQBAJ.

[19] D. Yaga, P. Mell, N. Roby, and K. Scarfone, "Blockchain Technology Overview," *arXiv:1906.11078 [cs]*, p. NIST IR 8202, Oct. 2018, arXiv: 1906.11078. [Online]. Available: http://arxiv.org/abs/1906.11078

[20] "Key generation," page Version ID: 1084900046. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Key_generation&oldid=1084900046

[21] B. Preneel, "Cryptographic hash functions," *European Transactions on Telecommunications*, vol. 5, no. 4, pp. 431–448, 1994.

[22] R. Guerraoui and L. Rodrigues, *Introduction to Reliable Distributed Programming*. Springer Science & Business Media, google-Books-ID: afEGuktwz_UC.

[23] G. F. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems: Concepts and Design*. Pearson Education, google-Books-ID: d63sQPvBezgC.

[24] "Consensus (computer science)," page Version ID: 1086676591. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Consensus_(computer_science)&oldid=1086676591

[25] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," vol. 4, no. 3, p. 20.

[26] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," pp. 1–9.

[27] Home. [Online]. Available: https://ethereum.org

[28] Litecoin - open source p2p digital currency. [Online]. Available: https://litecoin.org/

[29] Dogecoin. [Online]. Available: https://dogecoin.com/

[30] T. K. Mackey, T.-T. Kuo, B. Gummadi, K. A. Clauson, G. Church, D. Grishin, K. Obbad, R. Barkovich, and M. Palombini, "'fit-for-purpose?' – challenges and opportunities for applications of blockchain technology in the future of healthcare," vol. 17, no. 1, p. 68. [Online]. Available: https://doi.org/10.1186/s12916-019-1296-7

[31] M. Andoni, V. Robu, D. Flynn, S. Abram, D. Geach, D. Jenkins, P. McCallum, and A. Peacock, "Blockchain technology in the energy sector: A systematic review of challenges and opportunities," vol. 100, pp. 143–174. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1364032118307184

[32] D. Bhowmik and T. Feng, "The multimedia blockchain: A distributed and tamper-proof media transaction framework," in *2017 22nd International Conference on Digital Signal Processing (DSP)*, pp. 1–5, ISSN: 2165-3577.

[33] Hyperledger sawtooth. [Online]. Available: https://sawtooth.hyperledger.org/

[34] Hyperledger fabric. [Online]. Available: https://www.hyperledger.org/use/fabric

[35] Home. [Online]. Available: https://cardano.org/

[36] M. Saad, J. Spaulding, L. Njilla, C. Kamhoua, S. Shetty, D. Nyang, and A. Mohaisen, "Exploring the attack surface of blockchain: A systematic overview," number: arXiv:1904.03487. [Online]. Available: http://arxiv.org/abs/1904.03487

[37] K. Wüst and A. Gervais, "Do you need a blockchain?" in *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*, Jun. 2018, pp. 45–54.

[38] S. Bano, A. Sonnino, M. Al-Bassam, S. Azouvi, P. McCorry, S. Meiklejohn, and G. Danezis, "SoK: Consensus in the age of blockchains," in *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, ser. AFT '19. Association for Computing Machinery, pp. 183–198. [Online]. Available: https://doi.org/10.1145/3318041.3355458

[39] E. Anceaume, A. Pozzo, T. Rieutord, and S. Tucci-Piergiovanni, "On finality in blockchains," *arXiv preprint arXiv:2012.10172*, 2020.

[40] Proof-of-work (PoW). [Online]. Available: https://ethereum.org/en/developers/docs/consensus-mechanisms/pow/#finality

[41] Z. Zheng, S. Xie, H.-N. Dai, W. Chen, X. Chen, J. Weng, and M. Imran, "An overview on smart contracts: Challenges, advances and platforms," *Future Generation Computer Systems*, vol. 105, pp. 475–491, Apr. 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X19316280

[42] J. Eberhardt and J. Heiss, "Off-chaining models and approaches to off-chain computations," in *Proceedings of the 2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*, ser. SERIAL'18. Association for Computing Machinery, pp. 7–12. [Online]. Available: https://doi.org/10.1145/3284764.3284766

[43] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted execution environment: what it is, and what it is not," in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 1. IEEE, 2015, pp. 57–64.

[44] N. Weichbrodt, P.-L. Aublin, and R. Kapitza, "sgx-perf: A performance analysis tool for intel sgx enclaves," in *Proceedings of the 19th International Middleware Conference*, 2018, pp. 201–213.

[45] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song, "Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts," in *2019 IEEE European Symposium on Security and Privacy (EuroS P)*, pp. 185–200.

[46] I. Eyal, A. E. Gencer, E. G. Sirer, and R. v. Renesse, "{Bitcoin-NG}: A scalable blockchain protocol," pp. 45–59. [Online]. Available: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/eyal

[47] "Bitcoin scalability problem," page Version ID: 1066501654. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Bitcoin_scalability_problem&oldid=1066501654

[48] The web framework for perfectionists with deadlines | django. [Online]. Available: https://www.djangoproject.com/

[49] FastAPI. [Online]. Available: https://fastapi.tiangolo.com/

[50] Django ninja. [Online]. Available: https://django-ninja.rest-framework.com/

[51] Gunicorn - python WSGI HTTP server for UNIX. [Online]. Available: https://gunicorn.org/

[52] How to deploy with WSGI | django documentation | django. [Online]. Available: https://docs.djangoproject.com/en/4.0/howto/deployment/wsgi/

[53] Server workers - gunicorn with uvicorn - FastAPI. [Online]. Available: https://fastapi.tiangolo.com/deployment/server-workers/#uvicorn-with-workers

[54] Uvicorn. [Online]. Available: https://www.uvicorn.org/#running-with-gunicorn

[55] How to deploy static files | django documentation | django. [Online]. Available: https://docs.djangoproject.com/en/4.0/howto/static-files/deployment/

[56] Bitcoin core integration/staging tree. Original-date: 2010-12-19T15:16:43Z. [Online]. Available: https://github.com/bitcoin/bitcoin

[57] Nodes and clients. [Online]. Available: https://ethereum.org/en/developers/docs/nodes-and-clients/#execution-clients

[58] Enterprise on ethereum mainnet. [Online]. Available: https://ethereum.org

[59] Blocks. [Online]. Available: https://ethereum.org/en/developers/docs/blocks/

[60] Go ethereum. [Online]. Available: https://geth.ethereum.org/

[61] Ethereum: A secure decentralised generalised transaction ledger. [Online]. Available: hhttps://ethereum.github.io/yellowpaper/paper.pdf

[62] IBFT 2.0 - hyperledger besu. [Online]. Available: https://besu.hyperledger.org/en/stable/HowTo/Configure/Consensus-Protocols/IBFT/

[63] "Besu ethereum client," original-date: 2019-09-04T21:11:20Z. [Online]. Available: https://github.com/hyperledger/besu/blob/6f85e1f83b92b0a7918de859b18ec1a0fa2a7045/CHANGELOG.md

[64] Proof-of-work (PoW). [Online]. Available: https://ethereum.org

[65] Besu changelog. [Online]. Available: https://github.com/hyperledger/besu/blob/main/CHANGELOG.md#15-additions-and-improvements

[66] Web3 foundation. [Online]. Available: https://web3.foundation/

[67] web3.js - ethereum JavaScript API — web3.js 1.0.0 documentation. [Online]. Available: https://web3js.readthedocs.io/en/v1.7.3/

[68] Introduction — web3.py 5.29.1 documentation. [Online]. Available: https://web3py.readthedocs.io/en/stable/

[69] IPFS powers the distributed web. [Online]. Available: https://ipfs.io/

[70] Swarm. [Online]. Available: https://www.ethswarm.org/

[71] E. Daniel and F. Tschorsch, "Ipfs and friends: A qualitative comparison of next generation peer-to-peer data networks," *arXiv preprint arXiv:2102.12737*, 2021.

[72] J. P. Cruz, Y. Kaji, and N. Yanai, "Rbac-sc: Role-based access control using smart contract," *Ieee Access*, vol. 6, pp. 12 240–12 251, 2018.

[73] Open portable trusted execution environment. [Online]. Available: https://www.op-tee.org/

[74] Open enclave SDK. [Online]. Available: https://openenclave.io/sdk/

[75] EGo. [Online]. Available: https://www.edgeless.systems/products/ego/

[76] W. Liu, H. Chen, X. Wang, Z. Li, D. Zhang, W. Wang, and H. Tang, "Understanding tee containers, easy to use? hard to trust," *arXiv preprint arXiv:2109.01923*, 2021.

[77] Vault by HashiCorp. [Online]. Available: https://www.vaultproject.io/

[78] eth. Answer to "how is the address of an ethereum contract computed?". [Online]. Available: https://ethereum.stackexchange.com/a/761

[79] KVM. [Online]. Available: https://www.linux-kvm.org/page/Main_Page

[80] Use IBFT 2.0 (PoA) - hyperledger besu. [Online]. Available: https://besu.hyperledger.org/en/stable/Tutorials/Private-Network/Create-IBFT-Network/

[81] Hyperledger besu system requirements. [Online]. Available: https://besu.hyperledger.org/en/1.3.5/HowTo/Get-Started/System-Requirements/

[82] Vault with consul storage reference architecture | vault. [Online]. Available: https://learn.hashicorp.com/tutorials/vault/reference-architecture

[83] "go-ipfs," original-date: 2014-06-26T08:14:34Z. [Online]. Available: https://github.com/ipfs/go-ipfs

[84] Limitations of ego go. [Online]. Available: https://docs.edgeless.systems/ego/#/knowledge/limitations

[85] Providers — web3.py 5.29.2 documentation. [Online]. Available: https://web3py.readthedocs.io/en/stable/providers.html

[86] V. Goyal, O. Pandey, A. Sahai, and B. Waters, "Attribute-based encryption for fine-grained access control of encrypted data," in *Proceedings of the 13th ACM conference on Computer and communications security*, ser. CCS '06. Association for Computing Machinery, pp. 89–98. [Online]. Available: https://doi.org/10.1145/1180405.1180418

[87] A. Lewko and B. Waters, "Decentralizing attribute-based encryption," in *Advances in Cryptology – EUROCRYPT 2011*, ser. Lecture Notes in Computer Science, K. G. Paterson, Ed. Springer, pp. 568–588.

[88] Filecoin. A decentralized storage network for humanity's most important information. [Online]. Available: https://filecoin.io/

[89] B. Fisch, D. Vinayagamurthy, D. Boneh, and S. Gorbunov, "IRON: Functional encryption using intel SGX," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. Association for Computing Machinery, pp. 765–782. [Online]. Available: https://doi.org/10.1145/3133956.3134106

[90] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A distributed sandbox for untrusted computation on secret data," vol. 35, no. 4, pp. 13:1–13:32. [Online]. Available: https://doi.org/10.1145/3231594

[91] M. Bowman, A. Miele, M. Steiner, and B. Vavala, "Private data objects: an overview," number: arXiv:1807.05686. [Online]. Available: http://arxiv.org/abs/1807.05686

[92] ownCloud - share files and folders, easy and secure. [Online]. Available: https://owncloud.com/

[93] ChainPoint - connecting supply chains. [Online]. Available: https://www.chainpoint.com/

[94] Tierion: Blockchain proof engine. [Online]. Available: https://tierion.com

[95] The crypto wallet for defi, web3 dapps and NFTs | MetaMask. [Online]. Available: https://metamask.io/

[96] Take a look at pubsub on ipfs. [Online]. Available: https://blog.ipfs.io/25-pubsub/

[97] OP-TEE TrustZone memory. [Online]. Available: https://github.com/OP-TEE/optee_os/blob/332dec4a4bb9935e650b9925577cef5d485af724/core/arch/arm/plat-rpi3/platform_config.h#L54-L64

[98] S. Mofrad, F. Zhang, S. Lu, and W. Shi, "A comparison study of intel sgx and amd memory encryption technology," in *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, 2018, pp. 1–8.

[99] BigchainDB • • the blockchain database. [Online]. Available: https://www.bigchaindb.com/

[100] Blockchain oracles for hybrid smart contracts | chainlink. [Online]. Available: https://chain.link/

[101] BigchainDB and smart contracts — BigchainDB 0.6.0 documentation. [Online]. Available: https://bigchaindb.readthedocs.io/projects/server/en/v0.6.0/topic-guides/smart-contracts.html

[102] Hypercore protocol. [Online]. Available: https://hypercore-protocol.org/

[103] Storj - decentralized cloud storage. [Online]. Available: https://www.storj.io/www.storj.io/

[104] arweave. [Online]. Available: https://www.arweave.org/

[105] How the hypercore protocol works. [Online]. Available: https://hypercore-protocol.org/protocol/

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**