





AuditTrust: Blockchain-Based Audit Trail for Sharing Data in a Distributed Environment

Hugo Lloreda Sanchez¹ , Sophie Tysebaert¹ , Annanda Rath² ,
and Etienne Rivière¹ 

¹ EPL/ICTEAM, UCLouvain, Louvain-la-Neuve, Belgium
`etienne.riviere@uclouvain.be`

² Sirris, Brussels, Belgium
`annanda.rath@sirris.be`

Abstract. There has been a significant recent interest in trust-building technologies for decentralized environments, especially for sharing data between mutually distrusting entities. One of the critical challenges in this context is to ensure that shared data cannot be tampered with, and that access to this data can always be traced and audited in a secure and trustworthy way, e.g., by using an access log to detect tampering. However, for audit trail data to be useful, it must be correct, immutable, and tied with access control mechanisms. We present AuditTrust, a blockchain-based secure audit trail for data sharing in a distributed environment. We prototype AuditTrust using several technologies, such as Hyperledger Besu, IPFS, the Intel SGX TEE, and Vault. Our evaluation of AuditTrust examines the latency costs of auditing and access control and shows the effectiveness of the approach.

Keywords: Blockchain · Access control · Access logging

1 Introduction

Before sharing data with third parties, organizations participating to a distributed environment would like to agree with them on the processing of this data, so that they can verify if an access intent is legitimate or not. These organizations would also like to know who has processed this data, by keeping immutable traces of every access intent and effective access. This requires access *logging*, i.e., to maintain a complete history of data access for auditing usages. The traditional 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 over 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 and security of those access logs. In this context, distributed ledger technology (blockchains) has gained significant interest with applications in cryptocurrencies, healthcare [3, 5], or IoT [4] to cite a few.

We propose AuditTrust, a system for secure and trustworthy data access and access logging in a zero-trust distributed environment. AuditTrust is implemented using a combination of decentralized and secure protocols and technologies, and in particular the Hyperledger Besu blockchain, the IPFS decentralized storage service, the Intel SGX trusted execution environment, and the Vault secret management software. Besides its prototype implementation, AuditTrust is intended also as a reference architecture that can find uses in diverse application domains where data needs to be shared between mutually mistrusting entities.

The remainder of this paper is organized as follows. We present in Sect. 2 a motivating scenario allowing to detail our problem statement through a use case. Section 3 explains why this topic is of interest, and discusses the state of the art related to similar use cases. Section 4 provides details about AuditTrust’s architecture and design, Sect. 5 describes our implementation and the evaluation of our system. We conclude this paper in Sect. 6.

2 Use Case: Problem Statement and Motivation

We detail in this section a specific use case, which we use as a motivational example to define our problem statement.

In this use case, several organizations, referring to both public and private sectors (e.g., city authorities, police, a smart traffic company, . . .) would like to access traffic data (e.g., data from CCTV cameras) managed by a traffic monitoring company operating in a specific city. Access to this data is governed by a mutually signed contract between one of these organizations and the company offering the traffic monitoring service. Once shared, this data can be processed by those who have access to it at their destination system. Without proper control of data access and usage at these destination systems, data can be easily shared with a third-party system. This can lead to a data breach and to serious legal consequences. To prevent that, access and usage of data should be auditable in a secure and trustworthy way by any organization participating to the system. To this end, there is a need to have an *audit tool* able to trace any access to the shared data in a trusted and transparent manner, by providing *immutable* access logs. Such an audit tool does not only allow verifying whether there has been suspicious activity inside the system but also to audit the access to shared data within a given time interval. As the organizations are both independent and mutually mistrusting, this audit tool (relying on access logs) cannot be managed by one single entity and the application itself implementing this tool cannot be trusted; rather, the management of such a tool should be decentralized. This decentralization is also necessary to be able to scale up the tool to higher volumes and larger systems, and to make it resilient through redundancy.

Our goal for AuditTrust is to build trust in data shared between mutually mistrusting entities (or participants) in a distributed environment. Typical data being shared could be videos or images originating from smart cities and used notably by smart traffic applications. However, the proposed solution can be

used in any application domains, beyond a smart traffic use case. We consider two kinds of mutually mistrusting actors: 1 data consumers (**DCs**) are entities processing data, and 2 data owners (**DOs**), who obtain data, e.g., from traffic monitoring, and share it with DCs. Naturally, a DO might not want to provide the same access to every DC and for every piece of data: each shared piece of data is subject to a role-based access control (RBAC) policy according to which access rights are granted. This policy holds for a bounded amount of time and defines the role or category of users (representing the DC’s business function) who are authorized to access the data.

In this use case, building trust in data means that 1 only authorized users are able to access the data according to the policy defined by the data owner. This policy must hold 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 tampered (corrupted). An adversary might try to impersonate another (legitimate) user or compromise any computer system on which the audit tool is available. We also assume that neither the client application (to which a user connects to for interacting with the system) nor the host machines (and operating systems) are trusted.

3 Related Work

Data auditing solutions were previously described in the literature [1, 2], although none matches exactly the requirements resulting from our use case.

PrivacyGuard by Xiao *et al.* [1] implements mechanisms for preserving data integrity and data confidentiality without referring to a trusted third party in the context of data sharing in a cloud environment. PrivacyGuard uses two domains, a control plane and a data plane. The former is related to blockchain interactions and especially to the deployment and execution of smart contracts; the latter is related to the use of a trusted execution environment (TEE) on the cloud for decryption purposes and data processing. PrivacyGuard takes into account more attacks than what we require (e.g., delayed computation in a trusted execution environment or TEE), but is also costlier to operate due to the intensive usage of the blockchain and smart contracts. Each access intent will lead to a smart contract transaction whereas in our solution, a smart contract transaction (publication of an access log) relates to a large batch of access intents. However, PrivacyGuard is easier to deploy because it does not require a specific audit tool: the distributed ledger is the access log itself.

Liang *et al.* [2] propose ProvChain, a decentralized solution based on ownCloud to collect and verify the history of all modifications made on shared data. However, their approach assumes some trust relationships, as it relies on an auditor designated by the data provider for verifying data modifications. The scope of the implementation of their solution is also restricted to ownCloud as it requires ownCloud’s hooks mechanism. Furthermore, only the provenance data is really decentralized: in ProvChain, Liang *et al.* [2] rely on a trusted entity to audit data. In our solution, the audit tool can be executed by any participant and does not need to rely on a trusted entity.

Miyachi and Mackey [5] propose 3 different models of decentralized data sharing solutions for healthcare applications, depending on the nature of the data, their purposes, and the applicable regulations. Among these models, the one related to consumer health information (CHI) is similar to our use case: it is used for sharing data, based on a decentralized storage system and TEEs. However, the authors do not consider the same trust model, which impacts the type of blockchain solution they can use. The CHI model allows, indeed, the use of a consortium blockchain where the governance is assigned to a specific group of organizations, an assumption we do not make for AuditTrust.

Wang *et al.* [6] also propose a solution for sharing data in a decentralized manner, with access control mechanisms as well as encryption for preserving data confidentiality. Similarly to the approach of Miyachi and Mackey [5], this solution does not implement an auditing mechanism.

AuditTrust relies on off-chain computation using a TEE. We are not the first to propose to combine the advantages of blockchain processing and the use of a TEE. Ekiden [8] offloads computation from blockchain nodes to a collection of computing nodes, so that the blockchain is solely used as persistent state storage. IRON [9] and Ryoan [10] also combine blockchain and TEE to allow computation on encrypted data through *functional encryption*, following access control that leverage a blockchain. Lastly, Intel introduced in 2018 a solution called Private Data Objects (PDOs) [11], based on TEE and a distributed ledger (Hyperledger Sawtooth). PDOs execute smart contract functions off-chain (within an Intel SGX enclave), while the outputs are stored on the blockchain.

4 AuditTrust: Design and Reference Architecture

In this section, we present the technical solutions for the realization of AuditTrust and our reference architecture of the tool with all its security components. We will present an implementation of this architecture in Sect. 5. AuditTrust consists of two key constituents: (1) secure data sharing (i.e. unauthorized access to shared data not possible) and (2) trustworthy access and usage auditing (i.e. access log manipulation not possible). We detail these in the following.

Secure Data Sharing. The enforcement of access/usage policies is necessary before data reaches its destination (i.e., at the DC, data consumer). To this end, we propose to use a role-based access control model expressed in a smart contract to control access to shared data. Roles and policies are defined and represented by smart contracts, allowing all parties to have the same information. Each time a data consumer requests access to some data, this access intent is recorded in the current access log (generated at the data consumer node before being shared with other participants). Each access intent is protected with RSA encryption, relying on the public key of the data owner. To gain access to the file, the local access log first needs to be uploaded to the off-chain platform. From this, a hash can be computed and then can be published on the blockchain, for sharing this access log file with other participants.

When the hash of the access log has been published on the blockchain, the data owner will retrieve the encrypted access log on the off-chain platform. Then, the DO will decrypt and verify the access intents related to its files. This verification process consists of two steps: 1 identity verification and 2 access verification. If the access intent is valid, the data owner will authorize the data consumer to access one of its files.

The data is stored on the off-chain platform and it is protected with a symmetric key issued by the data owner. Instead of transferring the entire file, which would make us lose the benefits of using an off-chain platform, the data owner only sends the cryptography key. However, to preserve the confidentiality of the key, this exchange occurs between secure enclaves hosted in trusted execution environments (TEE). This ensures that the key itself is not accessible to the user (data consumer) itself or to its surrounding software stack, and that the key is used to produce a cleartext version of the file for local use only. Once the enclave on the consumer side has received the key, the file is decrypted inside the enclave and exposed to the user space.

Access and Usage Auditing. To realize trustworthy access and usage auditing, we rely on raw access log stored on the distributed off-chain platform. The address (content identifier) usable to retrieve the log as well as a hash of its content must be stored in an immutable database, which is why we leverage a blockchain for this purpose. The auditing process is simple. First, we retrieve the raw access log and produce a hash-value of it based on a selected cryptographic hashing function. Then, we retrieve the corresponding hash-value stored on blockchain. If they match (hash-value of raw access log on local device and its corresponding hash-value stored in blockchain), there is no tampering, else, tampering occurred. AuditTrust supports two types of audits:

- Total audit: this audit will check the integrity of all access logs available on the off-chain platform and their associated hash published on the blockchain. It outputs a binary value (tampered or not tampered).
- Partial audit: this audit is focused on the data usage control of a specific file. Given two timestamps, it will find all the access logs published during this interval and related to the particular file.

4.1 Reference Architecture

In this section, we present a short description of AuditTrust’s high-level architecture. Figure 1 complements this section with a graphical representation of the architecture components. Note that we defer the discussion of the implementation (technologies) for these components to Sect. 5.

The AuditTrust architecture is formed of the following key components:

1. A **front-end** application used by users (both data owner and data consumer). This interface allows the user to connect to different modules of the system, including the database, the service backend, but also the auditing tools. Via

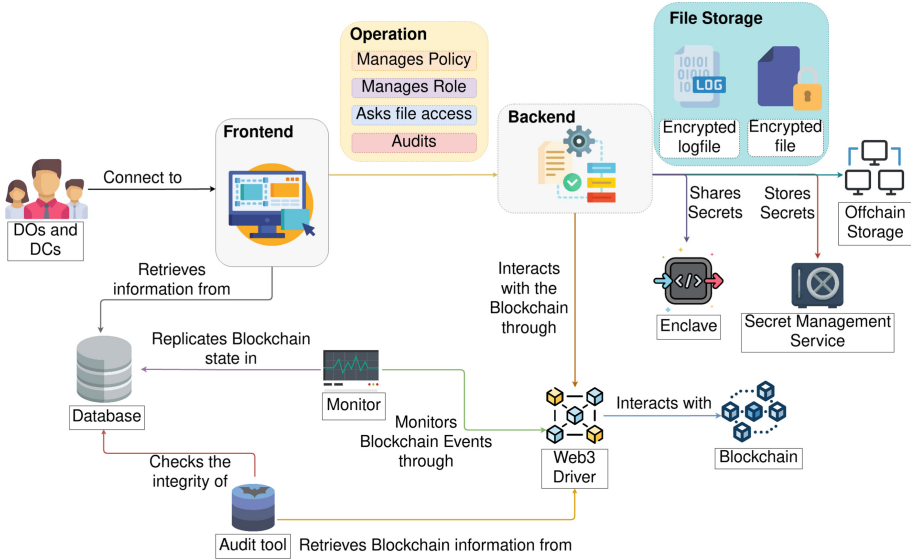


Fig. 1. Global architecture of AuditTrust, representing from a high-level perspective its core components.

this frontend, a user can also manage data access/usage policy and perform auditing on data access if it is authorized to do so.

2. The **back-end**, with different drivers/interfaces, interacts with the different modules of the system such as the blockchain monitoring tool, the blockchain itself, or the off-chain storage and secret management service for managing security key and other credentials.
3. An **off-chain storage** system is responsible for storing durably and reliably data off-chain, implementing the necessary level of redundancy and distribution of the data. It is used for storing *both* encrypted shared data and encrypted access logs.
4. The **audit tool** module is responsible for data access and usage auditing. It interacts with both the database, storing the state of the blockchain transaction, and with the blockchain system through a blockchain driver.
5. Finally, the **blockchain** is used to store the hash-value of the access logs and enables a transparent auditing process.

4.2 Security Mechanisms

In this section, we present several security mechanisms adopted in the design of AuditTrust. The design and implementation of AuditTrust required carefully thinking of a number of security aspects (establishment of secure communication channels and APIs, secure file transfer protocols, or key management) that employ traditional techniques and best practices but will not be discussed in

the present paper due to space limitations. We focus instead on aspects that are specific to AuditTrust.

Access Control Management. Proper access control is a critical security feature to allow secure data sharing. We analyzed various access control models when designing AuditTrust (such as discretionary access control - DAC - or mandatory access control - MAC -), and concluded that role-based access control (RBAC) was the most suitable and matches the requirements of our use case. RBAC [7] is a well-known access control model where access permission is granted based on role of user(s)/people in an organization, for instance, users can have a role of employee, administrator, . . . RBAC provides an easy way for the user and access rule management since a user is not connected directly to the access rule: the access rule is associated only with role. This means that, to grant or revoke a user’s access rights, we need to simply remove the user from the role they belong. When data is shared with data consumers, the access permission is defined based on the role of users belonging to the data consumer organization. This offers freedom for data consumer to assign, un-assign, and reassign a user to that role without affecting the access control policy defined by data owner. This is particularly beneficial when there is a change in user and role structure in the data consumer organization.

Implementing RBAC with Smart Contracts. In AuditTrust, RBAC is used for expressing access control policies. The RBAC policy is then transformed into a RBAC smart contract, which is considered as an agreement between data owner and data consumer. This idea was taken from Cruz *et al.* [12] and implemented in AuditTrust. This smart contract is only executed when new policies are created, updated or deleted. This provides transparency to all involved parties, while avoiding important costs of calling the contract for every access authorization. Based on information from the contract, any party can verify each access intent and enforce the RBAC.

Secure File/Secret Storage. For preserving the confidentiality of data (files and access logs), we use symmetric and asymmetric (RSA) encryption. We generate a symmetric key per shared file. We also encrypt access logs but, in this 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) and is encrypted with RSA using the DO’s public key. The use of encryption enables a DO to share files with a DC, and a DC to audit logs, but bears a risk that the key present in memory of one of the involved DC party will leak and be used to access data without authorization and without logging by malevolent parties. To prevent this risk, AuditTrust relies on a trusted execution environment or TEE. A TEE can execute code (a so-called secure *enclave*) that is certified and can be provisioned with secrets (here, the symmetric key) without revealing these secrets to the machine hosting the enclave or its operating system. The sharing of symmetric keys happens between secure enclaves in TEEs on both sides and the cleartext data is exported, without the key, to regular memory.

Blockchain Usage. AuditTrust leverages a blockchain for two primary purposes. First, it is used as a immutable storage medium to store roles and policies

as well as indexing information (content identifiers and hash values) allowing to retrieve content for DC and DO alike. The immutability of the blockchain storage is a necessary asset for a trustworthy auditing process. Second, the blockchain is used as a shared execution medium supporting smart contracts for the management of access control, i.e., setting rules, roles, and associated policies. The execution of these operations is auditable by any party ensuring the transparency of the information exchange agreement between DO and DC.

5 Prototype Implementation and Evaluation

We discuss in this section the implementation of the reference architecture presented for AuditTrust in the previous section, and motivate the technological choices for its different modules.

Blockchain Selection. A large number of blockchain technologies exist, targeting different membership models (open vs. consortium) and deployment models (public vs. private). An important selection criteria for AuditTrust was that the blockchain used should support *immediate finality*, i.e., the fact that a block appended to the chain (and the transactions therein) are never revoked and cancelled later, as can happen with fork events in most open, public blockchains such as “classical” Ethereum. Other important criteria was the compatibility with common toolsets and languages used for smart contracts, and the ability to support various deployment models in the future (i.e., not to be “vendor-locked” in a private solution).

We selected Hyperledger Besu, an Ethereum client supporting both public and private deployments. We use Besu with IBFT 2.0 [13], a proof-of-authority Byzantine fault-tolerant (BFT) algorithm, ensuring immediate finality.

We developed different contracts to handle policy management by data owners (DO), and by specific users of a data consumer (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 the management of policies;
- **DataConsumer:** One smart contract of this type is deployed every time a role is created by a DC;
- **DataConsumerOrganisation:** This smart contract is used for periodically sending access logs to the blockchain, by tracking and aggregating access logs for all managed roles instead of publishing them independently, which would increase the transaction fees of the solution. This smart contract is also responsible for validating the organization’s roles.

Off-Chain Storage and Anchorage on the Blockchain. Storing data directly on the blockchain is undesirable and would not scale to the requirements of AuditTrust. It is also undesirable to store data in a centralized location or even at a pre-defined consortium organization under our zero-trust assumption. We use instead a decentralized storage system that does not rely on pairwise

trust relations between entities of the system. We selected IPFS [14] for this purpose, one of the currently most popular decentralized store. We point out that any store offering a minimalistic put/get interface could be used in replacement. IPFS addresses data in a content-centric fashion, i.e., the content identifier of a file is the hash of its content. IPFS handles the replication and diffusion of data such that deletion or eclipsing by an adversary requires significant computational and communication power.

For each uploaded file, the content identifier (hash of the file) is published on-chain. This “anchorage” of off-chain data to in-chain records allows checking for data integrity and ensuring availability [15]. As IPFS is distributed and decentralized, data availability is also preserved, which is especially important for files such as access logs. A final positive aspect of decentralized off-chain storage is to avoid having a single point of failure.

Trusted Execution Environment (TEE). The two major choices for TEEs at our disposal from a material perspective or allowing a simulation are ARM TrustZone and Intel SGX. For the purpose of AuditTrust, the features offered are equivalent (even if the programming model differs). We considered both options in the implementation of AuditTrust and settled on Intel SGX due to the simpler programming environment and existence of more complete support libraries. We used specifically the Ego Go library for the Go language [16]. We point out that a production implementation of AuditTrust would probably have to realize a thorough analysis of the library safety as previous work found several weaknesses in other, similar solutions [17].

Client Application. Depending on the type of the user, several operations are possible through the client application. If the user is a data consumer, she can create and manage roles, demand access to some shared file, or access a shared file once access has been validated by AuditTrust. If the user is a data owner, he can create and manage policies for enforcing the RBAC mechanism as explained earlier. Both data consumer and data owner can audit the solution to check if any tampering occurred (resolution or penalization of such occurrences are out of scope of the present paper but would typically leverage on-chain proof-of-misbehavior checking using another smart contract).

Whenever a data consumer wants to access a particular file, an entry is appended in the current access log. This access log is published on the blockchain at periodic intervals. Once the access log is published, the data owner system performs verifications to see if these accesses were legitimate or not. These verifications are composed of two steps: (1) the verification of the identity of the data consumer through a signature mechanism and (2) the verification of the access, i.e. making sure that a policy authorizing this access exists and is still valid. This mechanism incentivizes DC organizations to publish the hash of this access log file on the blockchain because until then its users are unable to access any file.

Once the access is validated by the DO, the DC who made the request can contact the DO node to retrieve the key associated with the file. The DO node will again perform the identity verification, as mentioned before, to make sure

it does not share any information with a malicious or unauthorized entity. If the verification phase is successful, both the DO and the DC will attest enclaves in a pair-wise fashion (for SGX, this step requires using an Intel-provided service to attest the genuineness of the communicating SGX enclave and of its running code). Once enclaves are validated, both parties exchange necessary keys over a secure channel visible to only the enclave code.

5.1 Evaluation

Testing Environment. We use a setup environment with 4 virtualized machines on a single host. This host is equipped with a AMD 5800 h 8-core, 16-thread CPU, 32 GB of RAM, and fast NVMe SSD for storage. In addition to the four VMs hosted by KVM, the host runs simple Python scripts to orchestrate the evaluation. Each VM is allocated 2 virtual cores, 5 GB of RAM and 20 GB of disk space. Each virtual host is similar with a Hyperledger Besu node, an IPFS node, and other services implemented as web services. As the CPU does not support SGX, we employ the SGX emulation functionality of the Ego Go library [16].

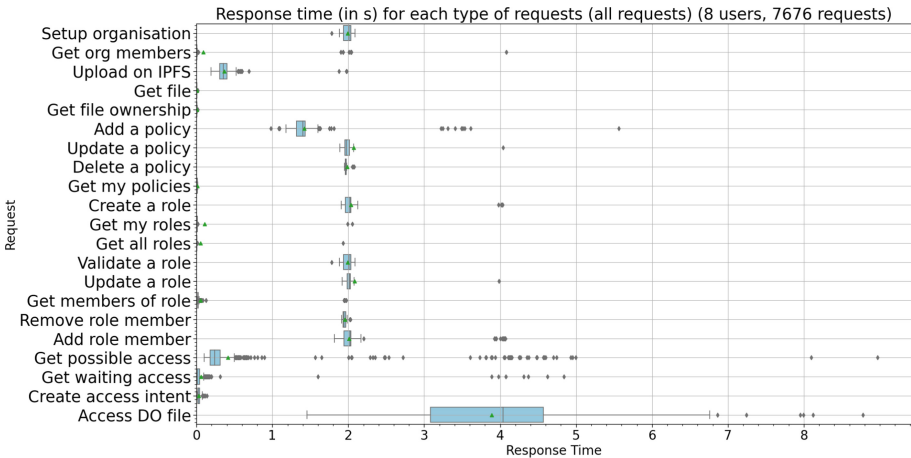


Fig. 2. Global overview of the response time (the vertical bar represents the median and the green triangle is the mean). (Color figure online)

Results. Figure 2 presents the distribution of response times for the various operations involved in the AuditTrust workflows. The latency of the different operations is impacted by calls to the blockchain (and the subsequent wait for the corresponding transaction to appear as finalized in a block), and calls to IPFS to lookup data. Operations are all successful with a majority under 3s of latency and an average of 3.89s. The `sign` method, which creates an access intent, is the fastest with an average of 0.03s.

The majority of requests that interact with the blockchain with a single transaction (e.g., update a policy, role validation, and other management requests.) take a similar time (around two seconds, which is the period of generation of blocks by Besu). The role creation call that requires a slightly more complex call on a smart contract and shows slightly higher variability. Other requests that only require reading the content of the chain replicated in the local database (e.g., getting the file ownership, get the list of roles or retrieving the last state of the solution.), have negligible latency, with the exception of getting the possible access that needs to verify if an existing access intent or authorized access does not exist already; in contrast to queries that only replicate the state of the blockchain without additional verifications. Furthermore, this request, but not only, suffers from the limited performance of the hardware used in our test environment. We also observe that the main function, accessing a file of a data owner by a data consumer, may take a relatively significant time. This is explained by two factors. First, the establishment of secure enclaves on both sides (emulated by Ego Go [16]) and the actual exchange of secrets take time, about 1 to 2s in total. Second, the relative load on the machine increases due to the in-enclave decryption of the file, leading to throttling mechanisms to trigger to avoid overloading the system in other modules. Overall, our evaluation shows that the cost of auditing and access mediated by a blockchain is not insignificant but remains justified for the level of security that it brings.

Discussion. Our evaluation already provides interesting preliminary results but more extensive tests with a distributed testbed would be necessary to evaluate AuditTrust in a production-like environment. We expect, nonetheless, that the availability of more resources per service (instead of sharing resources for different services in the same VM, and between VMs in the same host) will only provide enhanced performance, e.g., by allowing to decrease the default inter-block period in Besu.

We also note that performance is impacted by some convenience implementation choices and in particular the use of a synchronous communication pattern between our modules, e.g., between the back-end and the IPFS Web3 access library, or between the front-end and back-end components. We believe, nonetheless, that the refactoring of AuditTrust to support asynchronous calls does not incur more than engineering and implementation challenges, and that our proof-of-concept implementation validates the approach.

6 Conclusion

Sharing data and controlling its usage in a distributed context, when participants are mutually mistrustful, is a real challenge. Defining how this data can be processed through strict access control rules, and being able to audit its usage transparently with confidentiality and integrity in a distributed environment where participants do not trust each other is not a trivial problem.

This paper addresses this challenge by designing and prototyping a solution based on recent decentralized and zero-trust technologies including blockchain,

distributed storage, and trusted execution environment in addition to classical cryptography. AuditTrust allows data owners (DOs) and data consumers (DCs) to share and to request access to data, alongside the ability to check for fraudulent accesses within the system. AuditTrust, as shown in the evaluation, works well in a modest distributed setup and even with restrained resources and numerous services running in parallel. The proof-of-concept is also limited in its implementation, a point that we intend to address in our future work as well as with the deployment of a larger proof-of-concept in a smart city scenario.

References

1. Xiao, Y., Zhang, N., Li, J., Lou, W., Hou, Y.T.: PrivacyGuard: enforcing private data usage control with blockchain and attested off-chain contract execution. In: Chen, L., Li, N., Liang, K., Schneider, S. (eds.) ESORICS 2020. LNCS, vol. 12309, pp. 610–629. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59013-0_30
2. Liang, X., Shetty, S., Tosh, D., Kamhoua, C., Kwiat, K., Njilla, L.: ProvChain: a blockchain-based data provenance architecture in cloud environment with enhanced privacy and availability. In: 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), pp. 468–477. IEEE (2017)
3. Kuo, T.-T., Kim, H.-E., Ohno-Machado, L.: Blockchain distributed ledger technologies for biomedical and health care applications. *J. Am. Med. Inform. Assoc.* **24**(6), 1211–1220 (2017)
4. Shafagh, H., Burkhalter, L., Hithnawi, A., Duquenois, S.: Towards blockchain-based auditable storage and sharing of IoT data. In: Proceedings of the 2017 on Cloud Computing Security Workshop, pp. 45–50 (2017)
5. Miyachi, K., Mackey, T.K.: hOCBS: a privacy-preserving blockchain framework for healthcare data leveraging an on-chain and off-chain system design. *Inf. Process. Manag.* **58**(3), 102535 (2021)
6. Wang, S., Zhang, Y., Zhang, Y.: A blockchain-based framework for data sharing with fine-grained access control in decentralized storage systems. *IEEE Access* **6**, 38437–38450 (2018)
7. Sandhu, R.S., Coyne, E.J., Feinstein, H.L., Youman, C.E.: Role-based access control models. *Computer* **29**(2), 38–47 (1996)
8. Cheng, R., et al.: 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 (2019)
9. Fisch, B., Vinayagamurthy, D., Boneh, D., Gorbunov, S.: IRON: functional encryption using Intel SGX. In: ACM SIGSAC Conference on Computer and Communications Security (CCS), pp. 765–782. ACM (2017)
10. Hunt, T., Zhu, Z., Xu, Y., Peter, S., Witchel, E.: Ryoan: a distributed sandbox for untrusted computation on secret data. *ACM Trans. Comput. Syst.* **35**(4), 13:1–13:32 (2018)
11. Bowman, M., Miele, A., Steiner, M., Vavala, B.: Private data objects: an overview. arXiv, 5 November 2018
12. Cruz, J.P., Kaji, Y., Yanai, N.: RBAC-SC: role-based access control using smart contract. *IEEE Access* **6**, 12240–12251 (2018). <https://doi.org/10.1109/ACCESS.2018.2812844>

13. IBFT 2.0 - hyperledger besu. <https://besu.hyperledger.org/en/stable/HowTo/Configure/Consensus-Protocols/IBFT/>
14. Benet, J.: IPFS-content addressed, versioned, P2P file system. arXiv preprint [arXiv:1407.3561](https://arxiv.org/abs/1407.3561) (2014)
15. Eberhardt, J., Heiss, J.: Off-chaining models and approaches to off-chain computations. In: 2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers (SERIAL), pp. 7–12 (2018)
16. EdgeLess systems, Ego-Go library. <https://github.com/edgelessys/ego>
17. Liu, W., et al.: Understanding TEE containers, easy to use? Hard to trust. arXiv preprint [arXiv:2109.01923](https://arxiv.org/abs/2109.01923) (2021)