TYPE Original Research
PUBLISHED 23 July 2025
DOI 10.3389/fbloc.2025.1630839

OPEN ACCESS

EDITED BY
Alex Zarifis,
University of Southampton, United Kingdom

REVIEWED BY
Chengzu Dong,
Lingnan University, Hong Kong, SAR China
Qasem Abu Al-Haija,
Jordan University of Science and Technology, Jordan
Abdur Rasool,
University of Hawaii at Manoa, United States

*CORRESPONDENCE
Ivan A. Tarkhanov,
tarkhanov@isa.ru

RECEIVED 18 May 2025
ACCEPTED 10 July 2025
PUBLISHED 23 July 2025

CITATION
Hammoud O and Tarkhanov IA (2025) A scaling distributed access control model for blockchain-based file storage systems.
Front. Blockchain 8:1630839.
doi: 10.3389/fbloc.2025.1630839

A scaling distributed access control model for blockchain-based file storage systems

Obadah Hammoud[1] and Ivan A. Tarkhanov[2,3]*

[1]Engineering Cybernetics Department, Institute of Computer Science, National University of Science and Technology, Moscow, Russia, [2]Department No. 9 “Mathematical Support of Computer Technology” Federal Research Center “Informatics and Management” Russian Academy of Sciences, Moscow, Russia, [3]Scientific Research Department, State Academic University for Humanities, Moscow, Russia

Blockchain is considered as one of the popular solutions for decentralized data storage which offers high availability and data immutability due to the use of a specific structure for storing transaction blocks in combination with consensus algorithms. However, the nature of blockchain makes it not suitable for storing big amounts of data, like access control matrices which are typically used by DAC. This research proposes a new access control model based on DAC and RBAC models that is capable of managing access of various users, by storing minimal data in blockchain, and full data off-chain with the help of Merkle trees. A new model was proposed, which allows compressing access control data off-chain, and storing only Merkle root hash on-chain. The article describes DecStore - blockchain-based file storage system and how access control model can be scaled to more than 1,000 users and 1,000 storage objects using a caching mechanism on the users' side. Experiments were conducted to verify the scaling of the proposed model. Based on the obtained result, it was concluded that the proposed model is applicable to a wide range of systems, including IoT. This model is one of the first to solve the problem of storing large-dimensional DAC RBAC data.

KEYWORDS
access control, blockchain, Merkle tree, decentralized system, file storage system

1 Introduction

Access control management is considered an essential component in various systems. Whether it is a file storage system, a social media website or any other system which provides access to its private resources, it is required to manage how different parties can access these resources. Many blockchain-based systems are not dedicated for managing access control, but use access control as a required functionality within the system, such as (Dong et al., 2020), where the author presents a blockchain-based model for banking, and one of the used smart contracts is called “Controller Contract,” which handles access control, in addition to other control-related tasks. According to (Butincu and Alexandrescu, 2024), web3 will reshape the concept of users identity by making it decentralized. Thus, it is clear that having decentralized access control is required.

Access control can be classified into the following main types (Mudarri et al., 2015):

A. Mandatory access control (MAC): It is a type of access control that has levels of confidentiality. An example is when resources are classified as {common, secret, top secret}. Thus, users can access resources based on the security level they are granted.
B. Discretionary access control (DAC): in this type of access control, the owner or the creator of the resource defines who is allowed to access it. Usually, DAC uses a matrix access model (Benantar, 2006), which is a table that describes privileges in a subject (for example: user) versus object (for example: resource) style. In social media networks like Facebook (2023), a user can limit people who can access and see the content of his profile or specific posts, which is considered as an example of DAC.
C. Role-based access control (RBAC): In this type of access control, users have different roles, and resources are accessed based on the role they have. For example, in e-commercial websites, some pages are limited to administrators, other pages can be accessed by managers and administrators, etc.
D. Rule-based access control (RuBAC): This type of access control defines the rules by which resources can be accessed, which can be various. As an example, some banks might put a condition that currencies cannot be exchanged at night. Rule-based access control is usually used in combination with Role-based access control (Gupta et al., 2014).
E. Attribute-based access control (ABAC): It manages access by assigning policies for different attributes of users, resources and environment (Tarkhanov, 2016).

It is important to note that in modern systems a combination of different types of access control models is usually used. Corporate and government systems store millions of objects, including financial, social media data, and IoT data. They need flexible and reliable access control methods.

The usage of DAC can be sometimes unavoidable because it might offer more control over data than other systems. For example, sometimes it is required to provide a specific user access to a specific resource, without giving permission to other users with the same role or set of attributes. The problem is that DAC can result in a big matrix of resources against users. In storage files systems, if there are 1 million files in total, and 1,000 users, we will end up with a billion entries, which might be problematic when considering distributed systems with huge data redundancy like blockchain. When considering RBAC, less data is required to be stored, as a single policy can cover many files. However, the resulting data size might be still big, and unsuitable for blockchain storage. The aim of this research is the development of a distributed DAC+RBAC access control model that solves the problem of scaling for blockchain-based systems without centralized components and minimizes the size of the access rights storage in the blockchain.

2 Study case for proposed model

We will consider DecStore as a study case which is described in detail in a series of articles (Hammoud and Tarkhanov, 2022; Hammoud et al., 2021). DecStore is a decentralized storage system, which uses blockchain (Hyperledger Fabric (HYPERLEDGER, 2019)) to manage files, users and storage nodes. In this system, files and folders are distributed across virtual disks (VDs) of a fixed size using several algorithms. DecStore can be used by government and financial organizations to provide secure and reliable exchange of confidential files between organizations or within a large holding company. This system uses virtual clusters (VCs). Each VC contains 3 VDs, located on different storage nodes. The first VD in any VC contains blocks of files which hold the content of the first halves of files. The second VD holds blocks which contain the second halves. The last VD contains blocks which contain the according Erasure coding (Balaji et al., 2018) of the halves stored in the previous two-halves. Any storage node can hold many VDs which are not located in the same VC. Figure 1 shows the architecture of the proposed system. Described below access control model can be applied in any distributed blockchain-based system.

It is important to note that the proposed access control model is not proposed to be used with DecStore only. VDs can be expressed in other systems by docker volumes, virtual disks (VHD of virtual machines), folders, etc.

3 Related works

There are many studies that discuss applying access control in blockchain. Most of these studies focus on how to implement access control using blockchain for a specific field, especially IoT (Cheng et al., 2022; Zhang et al., 2019). Cheng et al. (2022) suggested deploying policy decision points (PDP) on blockchain, and storing policy administration points (PAP) off-chain to reduce the load size on blockchain. However, this study does not discuss how off-chain data can be stored and how it is replicated or located to prevent data loss in case of storage server loss. So, access control rules are basically centralized (off-chain resource) and are accessed by a decentralized system (blockchain), which is not a real decentralized system. Also, based on the solution they offer, when requests are sent to the blockchain, a request is forwarded to off-chain resources for every single request, which might not be effective in large-scale systems.

Maesa et al. (2017) proposed a hybrid framework that can store policies right in blockchain or link to it. This framework rewrites policies to minimize redundancies and stores a compressed version of it in blockchain. This solution can work well with ABAC access systems, but it does not fit DAC systems, as these policies are based on users, and policy files can be large.

Paillisse et al. (2019) proposed a distributed system that depends on blockchain to store policies. In their paper, they suggested that administrators can control operations using a CLI based on a Group-Based Policy. However, they suggest storing all policies on the blockchain, which is not feasible when considering DAC.

Wang et al. (2018) proposed a model that enables storing data on IPFS, while the access control is managed using smart contracts. When it is required to grant a user access to a data object, the data owner adds his address to the list of people who are allowed to access this file. This method does not consider the possibility of determining the type of access, whether it is read or read/write.

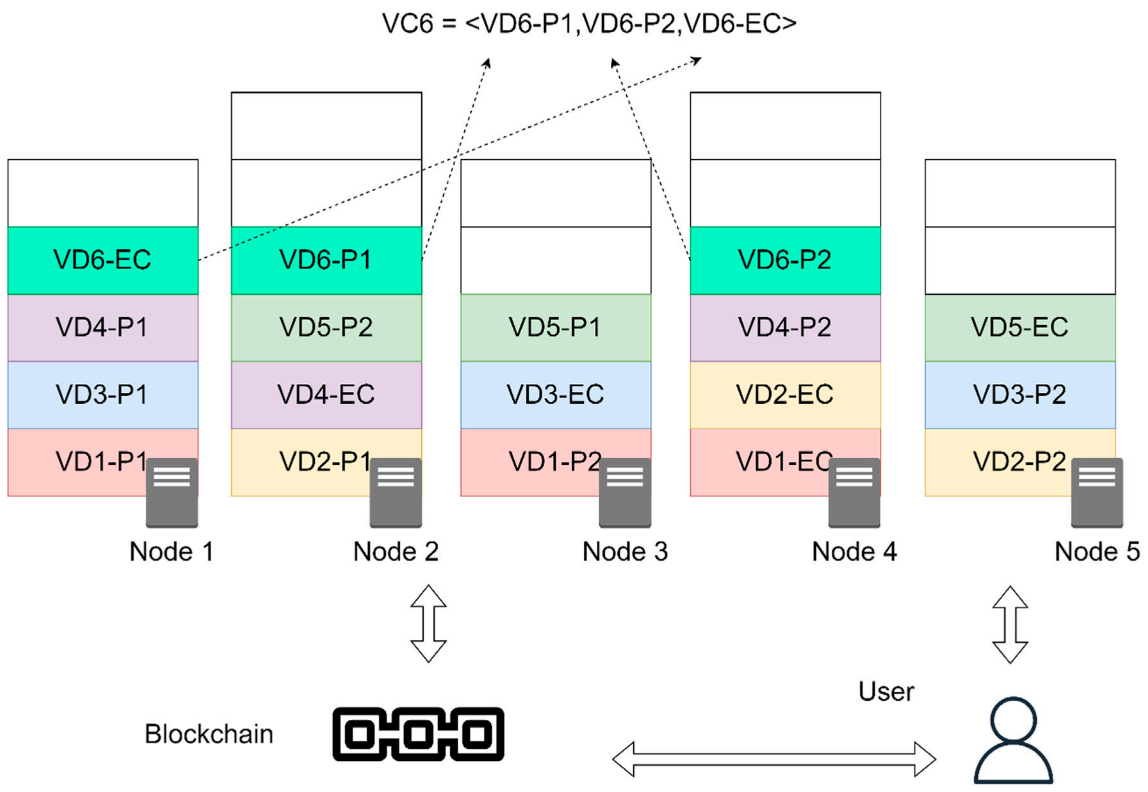

FIGURE 1
Data storage architecture in DecStore.

Also, such a model does not decrease the required data storage on blockchain, as it still stores the full matrix of access control of users.

Sun et al. (2021) proposed a system based on Hyperledger Fabric, which is dedicated for IoT applications, where the edge devices are considered as Policy Enforcement Point, and get the information from the blockchain ledger. Access control attributes and policies are stored on the ledger, which means that the problem of minimizing the size of stored data on blockchain is not considered.

Han et al. (2025) proposed a blockchain-based access control model, that uses Elliptic-curve cryptography for the encryption of data, where data gets encrypted and stored on the blockchain ledger. The method is mainly used for attribute-based access control, and the rules of access control are represented by a tree of AND and OR rules. This method was proposed for controlling requests from drones for accessing data. It is clear that storing the encrypted data on the blockchain does not minimize the data size on the blockchain ledger.

Dai et al. (2024) managed to define the rules for controlling the access of users to hazardous materials using smart contracts. The method can be summarized by adding the hash of the user to the list of hashes of users allowed to access the material, which means that the method does not reduce the size of data on the blockchain.

Another solution (De Oliveira et al., 2022) suggests storing the required resources off-chain while maintaining the verification on blockchain. According to this solution, several smart contracts can be deployed: Policies Enforcement Smart Contract (PEPSC), Policies Decision Smart Contract (PDPSC), Policies Smart Contract (PAPSC) and Policies Information Smart Contract (PIPSC). Policies are stored using XACML model (Masi et al., 2012). This method works well for ABAC, but storing the rules on blockchain in the case of DAC is not recommended for the reasons mentioned above.

One interesting method for decentralized access is using zero-knowledge proofs (e.g., zk-SNARKs) (Chen et al., 2022). This method allows users to send proof of identity to the verification entity instead of sending the real identity. However, this method has a limitation, which is that the rules for all files/objects should be stored, so the verifier can decide if this proof satisfies the rules for accessing the required resource.

In this article, we propose a DAC and RBAC access control model which results in minimum data storage size on blockchain, and high scalability without compromising the expected security level of the system or access speed.

4 Methods

4.1 Proposed access control model

The proposed access control model can be summarized as following points:

- Creating Merkle trees for various storage entities in each physical server
- Combining and compressing these trees

TABLE 1 DAC access control data storage in blockchain.

User	Merkle tree root hash	Roles
f970e2767d0cfe75876ea857f92e319b	006d2143154327a64d86a264aea225f3	Administrator
7694f4a66316e53c8cdd9d9954bd611d	76d80224611fc919a5d54f0ff9fba446	Developer, team-leader

TABLE 2 RBAC access control data storage in blockchain.

Role	Merkle tree root hash	Nodes
Administrator	0cc175b9c0f1b6a831c399e269772661	1, 4
Developer	8a8bb7cd343aa2ad99b7d762030857a2	2, 3

- Store only the hash of the total Merkle tree hash for each user in blockchain, instead of writing the access control policy for each file on blockchain
- User Merkle proof to verify authorization

To achieve an access control model for DecStore that supports scalability and requires minimal data storage in blockchain, the system architecture and storage model can be modified as follows:

- Each virtual disk (VD) is provided with a unit called “Permissions Storage unit.” For each user who has access to at least one file in the specified VD, a tree is created in this unit. This tree represents files organized within directories which the user has access to by DAC. The storage units located in VDs which belong to the same VC have the same copy of trees.
- Each storage node has a single tree for each user which represents the combination of his DAC trees in the VDs existing in this storage node.
- On the blockchain side, each user has 1 hash stored that represents the total hash of his Merkle tree root (Merkle Tree: A Fundamental Component of Blockchains | IEEE Conference Publication | IEEE Xplore, 2023). If there are a million files, and 100 users, only 100 hashes will be stored. Table 1 shows how users access information is stored in blockchain. Merkle tree root hash in this table represents user’s DAC Merkle tree root hash.

Also, the roles are stored in blockchain in a similar way, as in Table 2. For each role, a Merkle tree is built in which files that are accessible by this role are addressed as tree leaves.

A typical flexible access control system is managed by policies and consists of (Adams, 2005):

- Policy administration point (PAP): It is the component that creates access policies.
- Policy information point (PIP): It is the source which holds the policy information based on which the user’s access is granted or denied.
- Policy Decision Point (PDP): It is the component that decides whether a user is granted access or not.
- Policy Enforcement Point (PEP): It is the component that creates a request and sends it to PDP. It collects required information from PIP and other resources, like user request information to PDP
- Policy Retrieval Point (PRP): This component is not mandatory. It is used only when there are several PDPs and it is required to provide a centralized point for sending or retrieving access policies.

In our proposed model, policy information point (PIP) is not stored on a centralized server. When PDP (which is represented in the system by a blockchain smart contract) makes a decision, it retrieves information from two entities: the user, and the records stored on the blockchain. The full access information is stored on the storage nodes, and users synchronize their own access trees with the ones located on the storage nodes. Policy Enforcement Point (PEP) which is also represented by a smart contract, sends to PDP the Merkle tree proof from the user request along with the according expected Merkle tree root hash.

Thus, PIP is represented by three entities:

- The distributed network of storage nodes, which have the full policies, and are used only to allow users to sync their policies.
- Users side, where each user stores his policies whether it is DAC or RBAC.
- Merkle tree root hashes for both users and roles, which are stored on the blockchain.

Users’ trees consist of three types of nodes:

Root–it represents the Merkle tree total hash.

Leaves: represent the files (or directories, in case the policy allows the user to access all files in a directory instead of selecting files manually). They can have an optional value “compressed.” We will discuss it in the tree compression algorithm section.

Intermediate nodes: they can be directories or combining nodes. A combining node is the node which combines two nodes of various types, where any of those two nodes can be a leaf, a directory or a combining node. This type of nodes will be discussed further in “Building binary Merkle tree” section.

Each tree node contains two associated data structures: Hash and value. Hash represents the hash of the value. Table 3 represents the values of each node type.

Figure 2 represents the updated data storage architecture of DecStore. In the updated architecture, it is clear that role trees are distributed across storage nodes, where a storage node does not store a role and its copy. Also, inside VDs, access trees are added.

The main outlines of the method can be described as follows:

1. An access control tree for each user is created in each storage node. It has the set of files in this node the user has access to, which exist in VDs of a specific preselected index. For each user, all trees from VDs in a node are combined to a single tree,

TABLE 3 Nodes' types and structure description.

Node type	Node structure	Example	Hash
File	{Name = "File name," Type = File, Access = "Access type," Compressed = "compressed paths," Path = "Path"}	{Name = "document.pdf,"Type = File, Access = R, Compressed = Path = "/documents/"}	aa72770681fa71958b02cf5511182f1fbae25efc9b361d0ebc0e533ed6842e64
Directory	{Name = "Directory name," Type = Directory, Access = "Access type," Compressed = "compressed paths," Path = "Path"}	{Name = "documents," Type = Directory, Path = "/"}	c5f503118d94f8e81085329abca5b4a21b64e2981548b13308f0a8282ba82cb8
Combining node	{Type = Combiner, Paths = ["paths array"]}	{Type = Combiner, Paths = []}	4c8d6d1c41e8740161f9353aa6d70a577e7d25f98ba9de1002dd851a8d016365

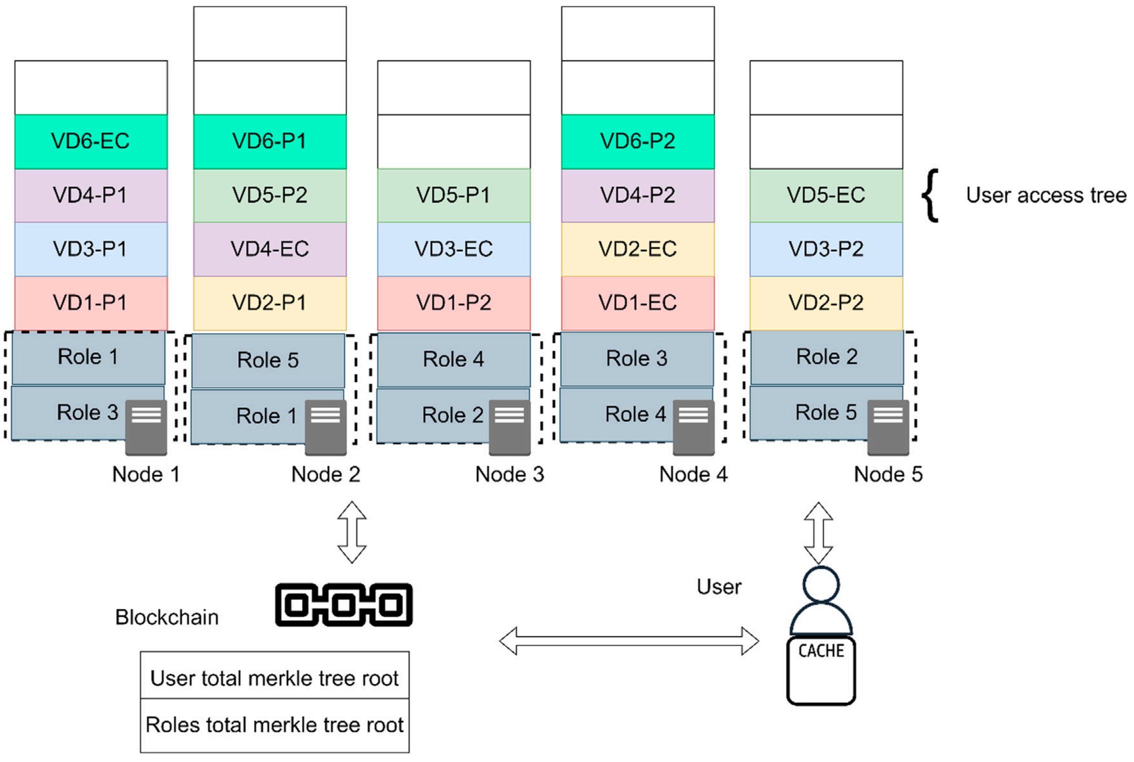

FIGURE 2
DecStore architecture with proposed access control model.

compressed and converted to a binary tree. These trees are used for DAC (Section 4.2).

2. Also, each storage node has several role trees. There is an extra copy of each role tree, which is located on a different node. All storage nodes have the same number of trees. Blockchain distributes the trees and their copies on the storage nodes.
3. Users synchronize their trees with the ones existing on the storage nodes. This includes their personal trees (DAC) and the roles trees (RBAC).
4. Users use their copies of trees to access files by generating Merkle proof. See Section 4.5.
5. Blockchain handles processes related with distributing and the recovery of trees in different situations, in order to grant that the system stays balanced, and that files are recoverable, or recovered automatically to another node when a storage node is lost.

4.2 Creating Merkle tree algorithm in case of DAC

This algorithm has three main phases:

6. In each VD which is selected by blockchain, a tree for each user is created. This tree shows which files a user has access to in this VD.

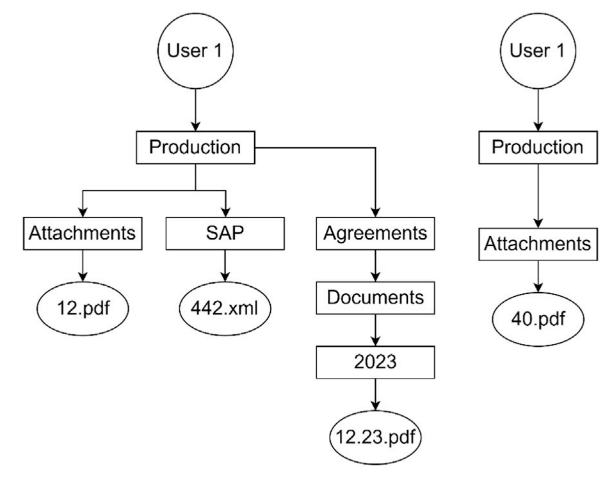

FIGURE 3
Separate VDs trees example.

7. Each node combines these trees into a single tree, which represents which files the user has access to in this node. In the combination process, files from different trees that are located under the same folder get combined under a single tree node.
8. The tree gets compressed after that, by combining tree nodes which have a single child into one tree node, in order to minimize the size of the tree (Section 4.2).
9. The resulting tree is converted into a binary tree, in order to minimize the size of Merkle proof (Section 4.4).

Users' access information is distributed across storage nodes using VDs. This means that data is replicated in VDs that belong to the same VC. When creating the combined Merkle tree, only one of the VDs of each VC participates in this process to avoid redundancy, which means that it is required to choose which VD will participate in this process. Selecting a fixed VD for all users is not practical, because it means more load on specific VDs, while it is preferred to distribute the load over VDs. Assigning a set of VDs for each user is not the best practice, as this means extra data storage in blockchain. The selection of VD in this algorithm is dynamic, based on the formula:

$$selectedVdIndex\,(user) = userId\,(user) mod\, 3$$

If the resulting number is 1, it means that only the first VD in all VCs is selected for building the tree for this user. If it is 2, it means that the second VD is used. If it is 0, it means that the VD which holds the Erasure Coding is used. The initial treeNode in the algorithm is the tree root node.

Trees are represented using sets of sets, where the main set is the tree root, and the elements of this set represent child nodes of the root (files and directories), and they are also sets that represent the same concept. These sets (tree nodes) have attributes (name, type, etc.). Files are represented by empty sets with attributes. We will use the words «tree nodes» and sets interchangeably. In each storage node, trees are selected based on the selected VDs (based on the selected VD index),

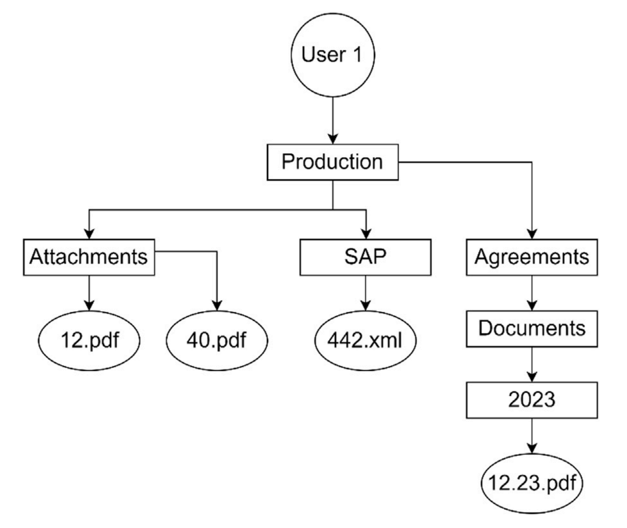

FIGURE 4
User 1 combined tree of VD3-P2 and VD6-P2.

which will be combined to build a single tree on the level of the storage node. The tree node is expressed by the set $TreeNode$, and the root of the tree is expressed as $TreeRootNode$. The initial selected $TreeNode$ in the algorithm is $TreeRootNode$. Attribute x of a tree root node can be expressed as $TreeRootNode^x$, and the first element of the set is $TreeNode_1$. This does not necessarily mean that this is the tree of VC1. It means that it is the tree of the first VD that the storage node holds, which is in the list of selected VDs. $CombinedTreeRootNode$ can be calculated by selecting the union (∪) of each $TreeRootNode$ that participates in this process. Starting from that, elements of a tree node $Node$ get combined as follows:

$$\begin{aligned} TreeNode = \{C = A \cup B \vee A \in Node \wedge \exists B \in Node,\\ B \neq A \wedge A^{Path} = B^{Path}\} \cup \{A \vee A \in Node \wedge \nexists B \in Node,\\ B \neq A \wedge A^{Path} = B^{Path}\} \end{aligned}$$

Which means that it can have items from different trees under the same directories, or it can be a single item if the directory has one file only.

Let's assume that in storage node 1 there are three VDs: VD3-P2, VD5P1 and VD6-P2. Let's assume also that the selected VD index is 2. In that case, only VD3-P2 and VD6-P2 are chosen. Now, if the according trees are shown as in Figure 3, the resulting combined tree will look like the tree in Figure 4.

The combination process repeats for every tree node in the list, and the join function is performed on all the resulting tree nodes as well until there are no more child nodes. The used notation shows that tree nodes that have the same name attribute are combined with each other and added to the TreeNode, as well as tree nodes that have unique names.

As the current study case considers DecStore, in other systems term VD can be replaced by "virtual machine," or any unit that allows logically separating files on one server (ex. folders). If files are directly stored on the server, this means that the algorithm of combining trees is not required.

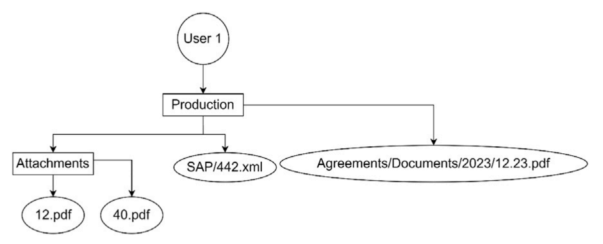

FIGURE 5
Compressed tree example.

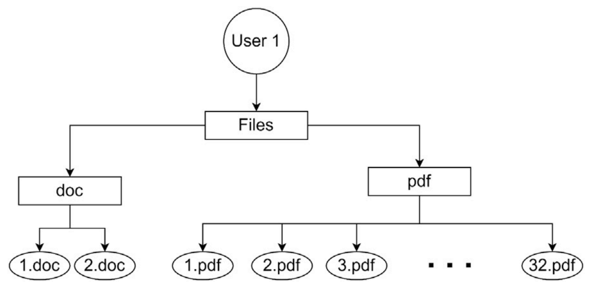

FIGURE 7
Example of a tree that does not use combining nodes.

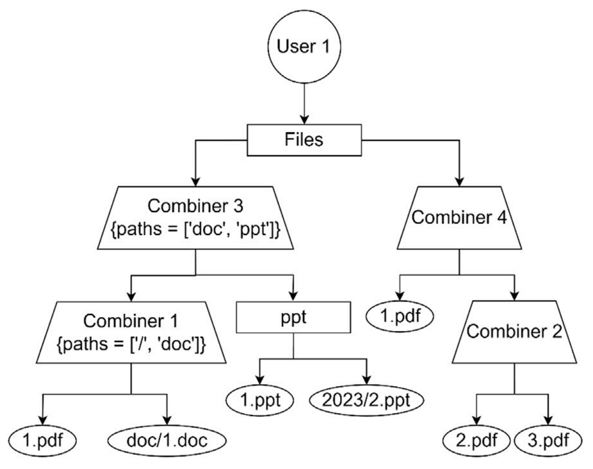

FIGURE 6
An example of the usage of paths attribute.

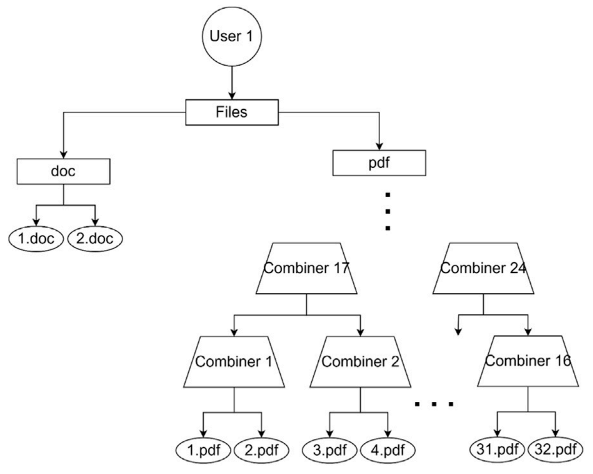

FIGURE 8
Example of a tree that uses combining nodes.

4.3 Compressing algorithm

After that, compression algorithm is called. This algorithm is required to minimize the size of the tree, by compressing tree nodes which have a single direct child into one, resulting in a smaller tree in total. This means that directories which have only one object (a file or a directory) are compressed into one. Joining two tree nodes into one is described by the formula:

$$JoinTreeNode\,(TreeNode) = \{TreeNode_1^{Compress} = TreeNode^{Name} + TreeNode^{Compress},\; TreeNode = TreeNode_1\}$$

"Compressed" attribute in the tree leaf represents the path of compressed tree nodes if they exist. "Access type" and "compressed" attributes in the case of directories are used only when the policy allows the user to access all files in a folder, so in this case, the directory is a leaf. Starting with the root node, the compress algorithm runs in a top-bottom model.

$$ParentTreeNode = TreeRootNode$$

The compress function for a parent tree node is defined by the formula:

$$Compress\,(ParentTreeNode) = \forall TreeNode \in ParentTreeNode,$$

$$\begin{cases} JoinTreeNode\,(ParentTreeNode), compress\,(TreeNode)\colon\ |ParentTreeNode| = 1 \\ compress\,(TreeNode)\colon\ |ParentTreeNode| > 1 \end{cases}$$

The compressed tree version of the tree presented in Figure 4 is shown in Figure 5. It is clear that "compress" attribute for 12.23.pdf is equal to Agreements/Documents/2023.

4.4 Building binary Merkle tree

The next step is converting the compressed tree into a binary tree. In order to perform this step, a new type of Intermediate nodes is added (combiner). Combiners are used to combine child nodes in such a way that only 2 tree nodes are considered as direct child nodes. This type of nodes has "paths" attribute. This attribute is set if the combining node has a directory in one of its direct descendants which are not combining nodes. Figure 6 represents how paths are created in binary trees.

In combiner 3, "paths" value is doc and ppt. That is because it has two direct folders: doc which is a "compress" value for the nodes doc/1.doc and ppt. Folder 2023 is not added to paths, because it is a

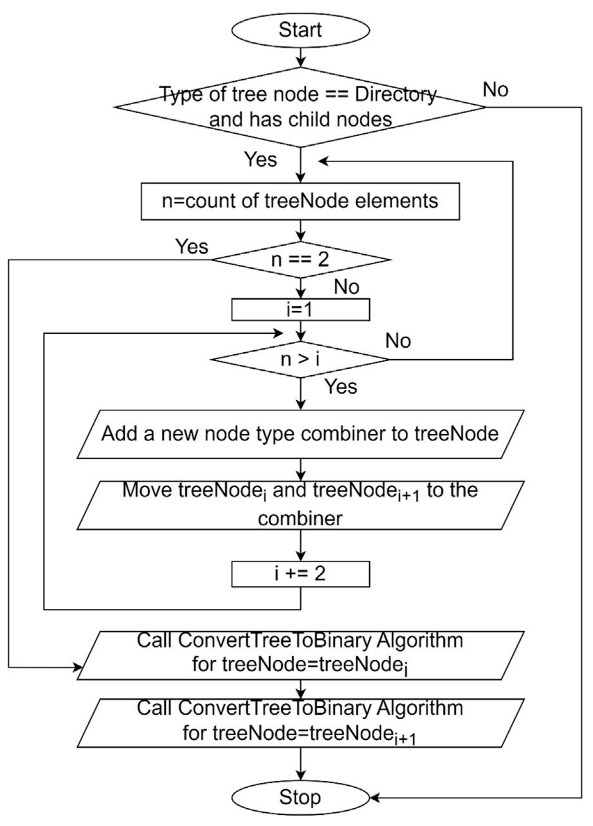
Start
Type of tree node == Directory and has child nodes
No
Yes
n=count of treeNode elements
Yes
n == 2
No
i=1
No
n > i
Yes
Add a new node type combiner to treeNode
Move treeNode$_i$ and treeNode$_{i+1}$ to the combiner
i += 2
Call ConvertTreeToBinary Algorithm for treeNode=treeNode$_i$
Call ConvertTreeToBinary Algorithm for treeNode=treeNode$_{i+1}$
Stop

FIGURE 9
Converting tree to binary algorithm.

child node of ppt, which is already added in paths. When accessing 2.ppt, given that the full path is/files/ppt/2023/2.ppt, Combiner 3 is directly accessed because in paths it has ppt. And after that, 2023/2.pdf is accessed directly.

Converting the tree into a binary tree is important in order to decrease the size of Merkle tree proof. Let's consider the tree example in Figure 7 which does not use combining nodes:

Merkle Proof of the file 1.pdf requires checking 33 tree nodes: 1.pdf + hashes of all pdf files + hash of doc directory. If we convert the tree to a binary one, it will require checking 7 nodes only: 1.pdf + hash of 2.pdf + hash of 4 combiners (combiners 2 + 18+26 + 30) + hash of doc directory as in Figure 8. The number of hashes of required tree nodes in a single directory for Merkle proof is equal to $log_2\ numberoftreenodes$.

Figure 9 represents the algorithm. It is clear that each 2 tree child nodes of any tree node get combined into one combiner, and the process is recursive until in total there are only 2 child nodes. The algorithm then is called for each child. In this algorithm, $treeNode_1$ is the first child of node treeNode. The algorithm is first called for the root node of the tree.

Once all the storage nodes build the user's tree, the root nodes are combined the same way using combining nodes to build a single Merkle tree. The hash of the resulting root node of this tree is stored for each. Let's consider building a Merkle tree for RBAC. For each role, a tree gets built. A role tree contains the files to which the policy associated with this role allows access. The same type of tree nodes and attributes proposed for DAC are used in RBAC. Roles get duplicated, and distributed across storage nodes, where a storage node cannot hold a role and its duplication. Each storage node holds $(|Roles|*2)/|N|$ of roles, where N is the set of storage nodes.

4.5 Users' access verification algorithm

This algorithm is used to verify whether a user has access to a file or not. When a user sends a request to access a file, whether the request is for reading or writing, he sends a Merkle proof along with the request. The body of the request depends on the access method used for this file. Table 4 shows the message header in the case of RBAC and DAC.

Each user has a copy of his own personal tree + a copy of the role(s) tree(s). The first step is to locate the file. This process can be done using a top-bottom method without searching for all possible tree nodes as in Depth First Search (DFS) (Tarjan, 1971), because the Merkle tree is built in a structure which keeps the files organized in its original path hierarchy. Files can be reached quickly by selecting the correct intermediate nodes. Combining nodes have "paths" attribute, which contains the names of direct directories if they exist. Compressed attribute is also used in this process, in case there are compressed nodes.

Once the user selects the file from the according tree, he sends the file info along with Merkle proof to the load balancer, represented by a blockchain smart contract. Blockchain smart contract first checks the access type, and hashes the file info, and then calculates the hash of the sum of the file info hash and the first hash in Merkle proof array and repeats the process for all hashes in Merkle proof array. After that, it compares the resulting hash with the stored one in Table 1 in case of DAC, or with the stored hash in Table 2 in the case of RBAC. Blockchain can confirm the user's identity based on his wallet authorization.

The user sends the file value without hashing it, and the Merkle proof array (required tree nodes) P = {P1,P2,P3, . . .Pn}, where P1 is the required file info value, and P2,P3,. . .Pn are the hashes of the required tree nodes for Merkle proof.

Assuming that hash(x) is the hash function, Merkle tree hash from the Merkle proof can be calculated using the recursive function $verifyMerkleProof(P, n)$.

$$verifyMerkleProof(P, i) = \begin{cases} 0, i = 0 \\ hash(P_i + verifyMerkleProof(P, i-1)) otherwise \end{cases}$$

When the user sends a request to access a file, The type of access sends (RBAC or DAC) is also sent along with Merkle proof. Thus, the system understands which entry is required to be compared with, as seen in Figure 10.

4.6 Synchronizing local access tree algorithm

4.6.1 SIn case of DAC

When a user gets granted or revoked to access a file f, or if the access type changes (read/write), this file gets added/removed/

TABLE 4 The header of an access file request.

DAC	RBAC
{Access = "DAC," MerkleProof = [. . .], FileInfo = {. . .}}	{Access = "RBAC," Role = "Role type," MerkleProof = [. . .], FilleInfo = {. . .}}

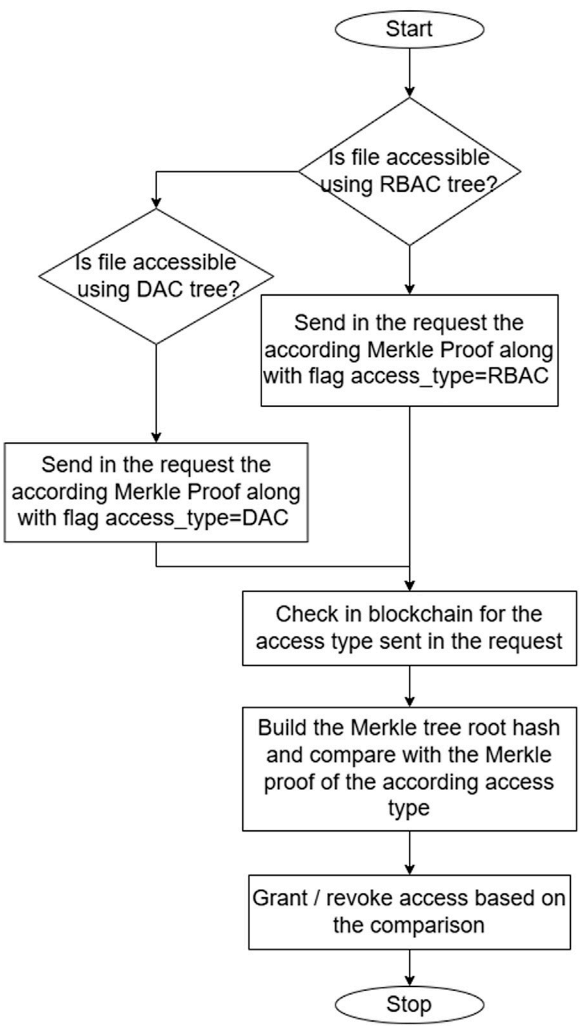
Start
Is file accessible using RBAC tree?
Is file accessible using DAC tree?
Send in the request the according Merkle Proof along with flag access_type=RBAC
Send in the request the according Merkle Proof along with flag access_type=DAC
Check in blockchain for the access type sent in the request
Build the Merkle tree root hash and compare with the Merkle proof of the according access type
Grant / revoke access based on the comparison
Stop

FIGURE 10
Algorithm of resolving type of access.

updated in the access tree in the according VD. Changes get propagated to the combined tree and to the total Merkle root hash. The user can detect changes in his tree by comparing his total Merkle root hash with the one stored in the blockchain.

4.6.2 In the case of RBAC

The same algorithm is used. If the user role gets changed, the tree gets updated, and the user detects that by comparing the role Merkle tree root hash stored in blockchain with his stored version of roles. If the user gets a new role added, the user requests a copy of the role Merkle tree from one of the storage nodes that host it, and the tree checks from the blockchain if the user has this role or not before sending it back. If the user loses a role, the role gets removed from his entry in blockchain, and thus access gets revoked.

5 Experiments results

In order to estimate the performance of the algorithms, an application which implements the proposed method for DecStore was developed. The application was written in C++ with the help of QT open-source platform, and the trees were represented in two different ways: SQLite files and JSON files, as the implementation technologies affect the performance results. The application was tested using a laptop running Ubuntu 22.04 LTS on core i7-8575u CPU and 16 GB of RAM. The experiment included creating two user DAC trees from scratch consisting of 500 new files each, in order to create the combined tree. SHA-256 hashing function was used to create nodes hashes. The test was performed automatically 100 times. The mean time for this process is 9.48 s in case of using SQLite, and 0.55 s in case of JSON files. Total tree size in case of SQLite is 249.9 KB, while it is 552 KB when using JSON. The implementation included combining 2 trees into one, compressing it and converting it to binary, and it is available here (Obadah Hammoud, n.d.).

In real life applications, DAC is probably much smaller than 1,000 files per node, as access is usually granted based on roles, and also by applying policies to directories and their subdirectories, instead of selecting individual files one by one, which means that the size of the resulting trees is supposed to be smaller.

In order to evaluate the performance of the system compared to other possible implementations, 3 systems were considered for testing:

1- Centralized system: Laravel PHP framework was used to implement it, as it is considered as a popular framework for implementing APIs. MySQL was used for storing access policies.
2- Blockchain Hyperledger: Where data is stored directly in blockchain instead of using an external resource to store policies. Docker was used for Hyperledger nodes.
3- DecStore: which is an implementation of the access control model proposed in this paper.

1,000 files entries were generated, as well as 1,000 users. All users were given access to all files (1,000 × 1,000). Three test cases were performed:

1- Granting a new permission to a file: Which expresses the speed of adding a new access rule for a user x to a file y (1 × 1).
2- Revoking a user's access completely: Which means revoking all types of access given to a user x to any file in the system (1 × 1000).
3- Validating access: It is the timer required by the system to check if a user x has access to a file y (1 × 1).

Obtained experiment results are shown in Table 5.

TABLE 5 Performance tests results.

Operation	DecStore		Blockchain (ms)	Centralized system (ms)
	JSON (ms)	SQLite (ms)		
Granting a new permission to a file	157	190	100	60
Revoking a user's access completely	400	300	300	60
Validating access	130	140	100	10

Another experiment was conducted in order to understand how much storage can be saved when applying the proposed method (DecStore) against storing data directly on blockchain. For this experiment, we considered the case of RBAC. 100 roles are created, along with 1,000 files. Each file can be accessed using one of 10 roles, from these 100 roles. Hyperledger Fabric (Blockchain) is used in this experiment, as in the previous one. The size of data is obtained from the peer nodes, be checking the size of the stored blockchain blocks:

– In case of using DecStore: 592 KB
– In case of using Hyperledger Fabric directly: 5.4 MB

It is important to mention that the roles were assigned directly. It is clear that the storage size in DecStore is significantly smaller than storing on Blockchain directly. Consecutive updating roles and adding more files will result in much bigger size when using Blockchain compared with Decstore. For example, adding or revoking a role from the whole system will require updating all affected files entries on blockchain to add/revoke the required role in the case of Blockchain, which is time consuming and results of having many transactions (storage size), while in DecStore, one transaction is required, which adds–remove the role hash.

Based on the performance tests results, we can conclude that the speed of the proposed system in various operations is not by far behind other systems, while it supports scalability unlike other systems. Also, it is clear that JSON files can be a good way to represent the trees. However, the performance might differ when implementing using other stack of technologies.

6 Discussions and limitations

The presented model has a number of advantages over the solutions discussed earlier in Section 3. It minimizes the size of the DAC and RBAC access rights data stored in the blockchain and is capable of scaling.

When adding new access to the user's combined tree, running the compressing algorithm and converting the tree to a binary can be performed faster than when creating from scratch, as the tree might not require compressing at all, and the result of running the algorithm of converting the tree to binary can end up with adding few nodes without changing most of the nodes as it is already in binary form.

Threats were analyzed using STRIDE modeling framework (Shostack, 2014):

- Spoofing: If the attacker were able to spoof the Merkle proof, he would not be able to use it to access data (as in replay attack), because the Merkle proof's hash is verified against the users' identity value, which is tied to their wallet. Additionally, manipulating nodes data by colluding nodes is not possible, as the network is permissioned, which means that outsider nodes cannot join and provide fake data, which prevents against attacks such as Sybil (Iqbal and Matulevicius, 2021).
- Tampering: If a user attempts to change his local tree copy by adding access to a file or changing the type of access from read to write, then the resulting Merkle proof will differ from the one stored on the blockchain, and access will be denied.
- Repudiation: If a user's access to a file was revoked and he does not allow updating his local tree and sends the Merkle proof he had before the revocation, he would not get access, as the Merkle proof does not match
- Information Disclosure: If the attacker were able to copy the Merkle tree cache from another user, this would not be enough to access data, as he has a different identity. The attack can only succeed when he also steals the identity (wallet) of a user.
- Denial of Service (DoS): Such a problem is considered when the attack is performed against a storage node rather than a blockchain node, as blockchain nodes are replicated. However, it is required to attack two nodes which have VDs that belongs to VCs that are shared among those 2 nodes to stop data availability.
- Elevation of Privilege (EoP): Users cannot self-assign data access. Permissions must be granted by data owners. Additionally, it is possible to track who has access to a specific file using trees built in storage nodes, and access of unwanted users can be revoked.

However, the proposed system has some limitations:

- Users are required to synchronize their trees regularly, in order to access the system. However, synchronization does not mean copying all the users' trees. Changes can be detected by comparing the total hash with the one stored in blockchain. In case the hash is different, the node which has the modified tree can be detected by its hash.
- Frequent changes to access rules means frequent synchronization of changes and running the associated algorithms. However, access can be assigned by directories instead of selecting individual files, and RBAC can be used in some cases instead of DAC, which can effectively minimize the number of required processes.
- Time of revoking a user's access operation is relatively long, which means that the proposed model is not suitable for systems where frequent deletion of the access rights is necessary.

- Deployment: Results and performance depend on the system settings, like the chosen blockchain network, which directly affects various parameters, such as transaction speed, gas cost, smart contracts, etc. Additionally, the system has several components, which may necessitate additional deployment time to ensure proper configuration and integration.

7 Conclusion and future work

In this research, we proposed a new access control system that can work with blockchain in combination with off-chain storage with caching on the users' side. In this system, a part of the PIP data is stored on the users themselves who request authorization, yet the model uses Merkle tree root hashes stored on the blockchain to prevent data manipulation. The proposed model uses a distributed system for access control management and provides the mechanism for maintaining data fairly distributed in several cases, including adding or removing storage nodes. The access control model consists of several algorithms which allow managing access control in different scenarios. The caching mechanism reduces the number of required requests, which means less pressure on storage nodes. Also, it means less response time, as the decision is made in blockchain itself, without requesting more data from the storage node.

As experiments have shown, the proposed model is scalable and minimizes the size of the rights storage in the blockchain several times. However, the provided tests are implemented on a local machine, and we plan to perform stress tests on the cloud in the future.

This study considered a distributed file storage system (DecStore) as a study case. However, this work can be adapted to fit other types of systems that use blockchain and off-chain storage to store any type of data. The approach proposed here does not depend on the method of encoding or restoring data, but considers the issue of reducing the dimension of the DAC matrix and the distribution of this data in blockchain-based systems.

In future, we plan to work on integrating this access control model with other technological platforms and study the outcome of such integration.

Data availability statement

Publicly available datasets were analyzed in this study. This data can be found here: https://github.com/Obadah-H/DistributedFileSystem.

Author contributions

OH: Validation, Conceptualization, Writing – original draft, Visualization, Software. IT: Supervision, Writing – review and editing, Conceptualization.

Funding

The author(s) declare that no financial support was received for the research and/or publication of this article.

Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Generative AI statement

The author(s) declare that no Generative AI was used in the creation of this manuscript.

Publisher's note

References

Adams, C. (2005). "Authorization architecture," in *Encyclopedia of cryptography and security*. Editor H. C. A. van Tilborg (Boston, MA: Springer US), 23–27. doi:10.1007/0-387-23483-7_18

Balaji, S. B., Nikhil Krishnan, M., Vajha, M., Ramkumar, V., Sasidharan, B., and Vijay Kumar, P. (2018). Erasure coding for distributed storage: an overview. *Sci. China Inf. Sci.* 61 (10), 100301. doi:10.1007/s11432-018-9482-6

Benantar, M. (2006). "Discretionary-access control and the access-matrix model," *Access control systems: security, identity management and trust models* (Boston, MA: Springer US), 147–167. doi:10.1007/0-387-27716-1_5

Butincu, C. N., and Alexandrescu, A. (2024). Design aspects of decentralized identifiers and self-sovereign identity systems. *IEEE Access* 12, 60928–60942. doi:10.1109/ACCESS.2024.3394537

Chen, T., Lu, H., Kunpittaya, T., and Luo, A. (2022). A review of Zk-SNARKs. *arXiv*. doi:10.48550/ARXIV.2202.06877

Cheng, C., Yan, B., and Wang, G. (2022). The blockchain based access control scheme for the internet of things. *Procedia Comput. Sci. Int. Conf. Identif. Inf. Knowl. internet Things* 202 (January), 342–347. doi:10.1016/j.procs.2022.04.046

Dai, Yi, Lu, G., and Huang, Y. (2024). A blockchain-based access control system for secure and efficient hazardous material supply chains. *Mathematics* 12 (17), 2702. doi:10.3390/math12172702

De Oliveira, T., Reis, L. H. A., Verginadis, Y., Mattos, D. M. F., and Olabarriaga, S. D. (2022). SmartAccess: attribute-based access control system for medical records based on smart contracts. *IEEE Access* 10, 117836–117854. doi:10.1109/ACCESS.2022.3217201

Dong, C., Wang, Z., Chen, S., and Xiang, Y. (2020). "BBM: a blockchain-based model for open banking via self-sovereign identity," in *Blockchain – ICBC 2020. Lecture notes in computer science*. Editors Z. Chen, L. Cui, B. Palanisamy, and L.-J. Zhang (Cham: Springer International Publishing), 12404, 61–75. doi:10.1007/978-3-030-59638-5_5

Facebook (2023). Facebook. Available online at: https://www.facebook.com (Accessed August 1, 2023).

Gupta, P., Stoller, S. D., and Xu, Z. (2014). Abductive analysis of administrative policies in rule-based access control. *IEEE Trans. Dependable Secure Comput.* 11 (5), 412–424. doi:10.1109/TDSC.2013.42

Hammoud, O., and Tarkhanov, I. (2022). A novel blockchain-integrated distributed data storage model with Built-in load balancing. doi:10.1109/AICT55583.2022.10013548

Hammoud, O., Tarkhanov, I., and Kosmarski, A. (2021). An architecture for distributed electronic documents storage in decentralized blockchain B2B applications. *Computers* 10 (11), 142. doi:10.3390/computers10110142

Han, P., Sui, A., and Wu, J. (2025). A secure and efficient access-control scheme based on blockchain and CP-ABE for UAV swarm. *Drones* 9 (2), 148. doi:10.3390/drones9020148

HYPERLEDGER (2019). Hyperledger architecture, volume 1 introduction to hyperledger business blockchain design philosophy and consensus. *Hyperledger Archit.* 1. Available online at: https://8112310.fs1.hubspotusercontent-na1.net/hubfs/8112310/Hyperledger/Offers/Hyperledger_Arch_WG_Paper_1_Consensus.pdf.

Iqbal, M., and Matulevicius, R. (2021). Exploring sybil and double-spending risks in blockchain systems. *IEEE Access* 9, 76153–76177. doi:10.1109/access.2021.3081998

Maesa, D., Mori, P., and Ricci, L. (2017). Blockchain based access control. doi:10.1007/978-3-319-59665-5_15

Masi, M., Pugliese, R., and Tiezzi, F. (2012). "Formalisation and implementation of the XACML access control mechanism," in *Engineering secure software and systems. Lecture notes in computer science*. Editors G. Barthe, B. Livshits, and R. Scandariato (Berlin, Heidelberg: Springer), 60–74. doi:10.1007/978-3-642-28166-2_7

Merkle Tree: A Fundamental Component of Blockchains|IEEE Conference Publication|IEEE Xplore (2023). Available online at: https://ieeexplore.ieee.org/document/9588047 (Accessed September 30, 2023).

Mudarri, T., Abdo, S., and Al-Rabeei, S. (2015). Security fundamentals: access control models. *Interdiscip. Theory Pract.*

Obadah Hammoud (n.d.). accessControl_Github. *GitHub*. Available online at: https://github.com/Obadah-H/DistributedFileSystem/tree/main/access_control (Accessed July 3, 2025).

Paillisse, J., Subira, J., Lopez, A., Rodriguez-Natal, A., Ermagan, V., Maino, F., et al. (2019). "Distributed access control with blockchain," in *ICC 2019 - 2019 IEEE international conference on communications (ICC)*, 1–6. doi:10.1109/ICC.2019.8761995

Shostack, A. (2014). *Threat modeling: designing for security*. Indianapolis: Wiley.

Sun, S., Du, R., Chen, S., and Li, W. (2021). Blockchain-based IoT access control system: towards security, lightweight, and cross-domain. *IEEE Access* 9, 36868–36878. doi:10.1109/ACCESS.2021.3059863

Tarjan, R. (1971). "Depth-first search and linear graph algorithms," in *12th annual symposium on switching and automata theory (swat 1971)*, 114–121. doi:10.1109/SWAT.1971.10

Tarkhanov, I. (2016). Extension of access control policy in secure role-based workflow model. doi:10.1109/ICAICT.2016.7991691

Wang, S., Zhang, Y., and Zhang, Y. (2018). A blockchain-based framework for data sharing with fine-grained access control in decentralized storage systems. *IEEE Access* 6, 38437–38450. doi:10.1109/ACCESS.2018.2851611

Zhang, Y., Kasahara, S., Shen, Y., Jiang, X., and Wan, J. (2019). Smart contract-based access control for the internet of things. *IEEE Internet Things J.* 6 (2), 1594–1605. doi:10.1109/JIOT.2018.2847705