Journal of Network and Systems Management (2026) 34:84
https://doi.org/10.1007/s10922-026-10065-2

A Smart Contract Architecture Using Hierarchical Factory Pattern and Multirole Access Control

Muhammad Bin Saif[1] · **Sara Migliorini**[1] · **Fausto Spoto**[1]

Received: 28 April 2025 / Revised: 1 March 2026 / Accepted: 22 April 2026

Abstract
The use of blockchain technology and smart contracts is progressively spreading to real-world applications, especially to those that deal with critical services, such as water management systems. In fact, these applications can exploit the immutability, enforceability, and trustworthiness of such technology while promoting a more reliable, tamper-resistant way to collect and store data. At the same time, the development of decentralized applications poses challenges with respect to scalability, efficiency, management, and security. To address these problems, some design patterns, such as the factory pattern, have been proposed in the literature, to deal with modularity and scalability; at the same time, these proposed solutions apply some common concepts of role-based access control (RBAC), by adapting them to the smart contract context. This paper extends prior work by extending the definition of a hierarchical factory pattern, enhanced with multirole authentication and authorization capabilities, and applying it in the context of a water management system. It provides an extensive description of both advantages and disadvantages of this solution, discussing why the ability to instantiate a hierarchical family of contracts is essential in some application domains, and how a finer management of dynamic roles and permissions can be achieved in this kind of design. This paper also performs an extensive analysis of the performance and scalability capabilities of the proposed solution, and discusses some security aspects by considering its ability to overcome certain security attacks.

Keywords Hierarchical factory pattern · RBAC · Smart contract · Decentralized application · Water management system

Extended author information available on the last page of the article

Published online: 05 May 2026

1 Introduction

The term decentralized application (or dApp) is commonly used nowadays to refer to a new kind of software application rooted in blockchain and smart contracts technology. In particular, this kind of software is characterized by a business logic implemented through smart contracts and a storage layer represented by a distributed ledger, namely a blockchain. A smart contract is essentially a piece of code that performs predefined actions on a blockchain, enabling automated execution and ensuring the immutability and transparency of agreements in a trustless environment [1]. The development of dApps is spreading to various application domains, including healthcare [2], tourism [3], energy and water management [4], identity management [5], cultural heritage preservation and restoration [6, 7].

As dApps begin to enhance traditional information systems and even become a valid substitute for them, the need to provide traditional functionalities and to apply consolidated design methodologies is becoming increasingly important to ensure their real success. In this regard, three pivotal aspects in software design and development are needed: scalability, modularity, and security. In the blockchain domain, *scalability* is strictly related to the ability to process an increasingly large amount of transactions in a reasonable amount of time and with limited costs [8]. Scalability is related to the cost and time required to invoke a smart contract function to obtain a change in the global state. It is typically a run-time aspect dealing with the execution stage, and it is tackled from three points of view: (a) the optimization of the smart contracts structure, in terms of their bytecode dimension, which affects the deployment phase, (b) the optimization of function implementations, which is related to the execution costs, and (c) the characteristics of the underlying blockchain, which determines the transaction speed and capacity. Relative to the last aspect, the recent development of Layer 2 solutions provides an unprecedented benefit in this direction [9]. Conversely, *modularity* is a property related to the design and development of smart contracts, enabling the management of their increased inherent complexity and correlations with other smart contracts [10]. Specific design patterns have been studied and proposed in the literature to promote modularity in smart contract development. Indeed, even if a wide range of design patterns have been proposed in the field of software engineering in the past, the decentralized nature of smart contracts and dApp leads to the need for new design patterns explicitly tailored for this kind of blockchain-based software solutions. Finally, with respect to the *security* aspects, in the development of real-world applications, multirole authentication and authorization are becoming increasingly important to ensure the real adaptability of dApps [11]. Namely, the need for secure, controlled, and authorized access increases as smart contracts become more complex and interactive, making it essential to ensure that only authorized entities can interact with a contract and perform specific actions.

Among all the patterns specifically tailored for smart contract development, the most critical one, with respect to all three mentioned properties, is the *factory pattern* [12]. This pattern has been developed with the idea of having a contract (the factory) responsible for creating other contracts. In this way, creating multiple instances of the same contract skeleton is possible by easily tracking them and simplifying their management. Besides modularity, the factory pattern affects scalability since it

reduces the deployment costs by only deploying the factory contract once. The latter will create subsequent children without redeploying its bytecode. Moreover, this pattern also increases smart contract security by reducing the risk of vulnerabilities [13]. However, as highlighted in [14], the traditional factory pattern could be too limited in real-world scenarios where different families or variants of the same contract need to be deployed. For this reason, in that work, the authors propose developing a hierarchical factory pattern in which, at each level, a different type of smart contract object can be created and managed, acting as a factory or a simple contract based on specific needs.

Regarding security aspects, a factory deployed at its specific level of the hierarchy cannot be considered as an island, with no connection to the factories deployed at the parent or sibling levels of the hierarchy. In particular, dependencies related to the different authorizations and roles govern the deployment and management of the smart contracts, which could need to be modified or revoked by the corresponding parent level. To address this situation, in [14], this pattern is also integrated with a Multirole Authentication and Authorization (MAC) mechanism explicitly tailored to work with this structure. The proposed system defines roles at each level and ensures that permissions are consistently checked across the hierarchy. The system dynamically checks and enforces role-specific permissions as contracts are instantiated and interacted with at various levels.

This paper is an extension of our previous work in [14]. It is different from and extends it in several aspects: (i) it contextualizes the proposed solution to a real-world case scenario represented by a water management system, discussing the importance and advantages of the proposed solution in critical infrastructures that need a hierarchical organization of users and functionalities, and limited access control to them. (ii) It provides additional details regarding both the management of the hierarchical contract structure and multirole authentication and authorization. (iii) It extends the evaluation section by also considering the use of Layer 2 solutions to reduce costs and improve performance, as well as by discussing the ability of the proposed solution to overcome some security attacks such as the signature replay attack and the need to dynamically revoke roles or transfer ownership.

The remainder of this paper is organized as follows: Sect. 2 presents the motivating example used throughout the paper to describe and discuss the proposed solution, which is introduced in Sect. 3. Section 4 details its implementation and evaluation with respect to both performance and security. Finally, Sect. 5 summarizes the related work, and Sect. 6 concludes by outlining future research directions.

2 Motivating Example

This section presents a motivating example derived from the water management domain to illustrate the practical relevance of the proposed approach. Water management is a critical aspect of modern infrastructure, requiring continuous monitoring and control of water distribution and quality at multiple administrative levels. A water management system could be organized in a hierarchical structure, as shown in Fig. 1. At the highest level, a water distribution authority acts as the super admin-

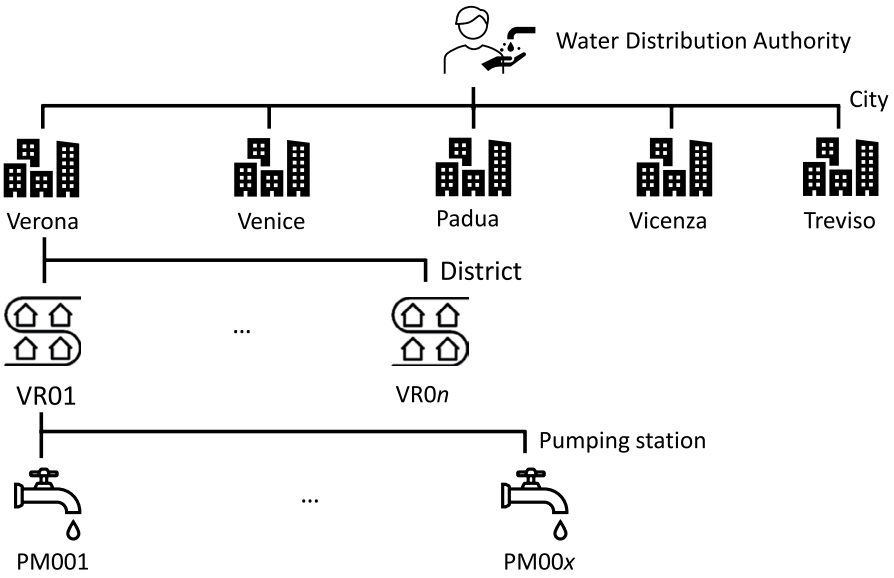

Fig. 1 Hierarchy of actors inside a water management system

istrator responsible for managing permissions and configurations across multiple cities, ensuring that water is extracted, processed, and distributed efficiently while adhering to regulatory policies on conservation and quality standards. This authority establishes baseline permissions and oversees the creation of subordinate entities represented by the set of supervised cities. Each city is responsible for managing its respective water distribution network, which includes multiple districts. At the same time, each district oversees a set of physical water treatment or pumping facilities. At the lowest level, each station comprises multiple water pumps and sensors that monitor essential parameters such as flow rate, pressure, pH levels, and turbidity.

Blockchain technology and smart contracts could successfully implement such a water management system. By combining blockchain technology with IoT sensor data, the system creates an immutable, transparent audit trail that supports operational efficiency and regulatory compliance, ultimately establishing a resilient framework for modern water management. In particular, while IoT sensors can collect data continuously and store it off-chain on distributed storage systems such as IPFS (InterPlanetary File System), secure records of the hashes of this data are stored in the blockchain to ensure transparency and regulatory compliance.

At the same time, when implementing a water management system through a dApp, it becomes clear that each level of the hierarchy is responsible for deploying and managing its own smart contracts while remaining dependent on the parent level. Moreover, the super administrator, also known as the Water Management Authority, grants and revokes permissions to ensure that only authorized roles manage water distribution and access water records. The city water departments are key client factories, coordinating their activities with the central authority and local subdivisions. They handle local water delivery and quality monitoring while implementing addi-

tional smart contracts to manage different districts. Finally, the district stations at the lowest level directly interact with water treatment facilities and sensor networks.

This real-world scenario could be successfully implemented through the hierarchical factory pattern proposed in [14]. In particular, its application facilitates efficient water management across different administrative levels and ensures that each system component can be independently updated and secured. The multirole authentication and authorization mechanism enforces strict access control, ensuring that operations are conducted only by entities with appropriate permissions.

The following sections discuss in detail how this real-world scenario can be implemented using a hierarchical factory pattern enriched with multirole authentication and authorization mechanisms. In particular, they will describe the benefits and characteristics of such solutions and discuss eventual envisioned future extensions.

3 The Proposed Solution

This section introduces the hierarchical factory pattern, incorporating multirole authentication and authorization mechanisms, using the water management system as an application scenario. The hierarchical factory pattern introduces a multi-layered structure for creating and managing smart contracts. Unlike the traditional factory pattern, which focuses on instantiating a single type or family of objects, the hierarchical factory pattern allows the creation of objects that, in turn, can act as factories for other objects, mirroring the organizational structure of a water management system. This approach results in a tree-like structure, where each node, such as a water distribution authority or a city manager, can serve as a factory for the subsequent level, for example, districts and pumping stations. Furthermore, the hierarchical factory pattern is integrated with a secure authentication and authorization mechanism to manage roles and permissions along the tree. This ensures that the top-level authority and the individual sensor data collectors consistently enforce access control. To mitigate the potential increase in complexity due to this hierarchy, shared libraries have been implemented for common functionalities, minimizing redundancy and improving adaptability for real-world applications in water management systems.

3.1 The Hierarchical Factory Pattern

The hierarchical factory pattern extends the traditional factory pattern by allowing the recursive creation of sub-factories at each level of the hierarchy so that each level consists of several distinct factories and/or distinct smart contracts.

Inside this hierarchical structure, there are mandatory and optional levels that can be added to accommodate an increased level of nesting and control. Figure 2 shows the general structure of the hierarchy: the mandatory parts of the tree have been highlighted in yellow, while white boxes correspond to optional levels that can appear in any number greater than or equal to zero, depending on the specific characteristics of the application domain.

The root level of the tree hierarchy is represented by the *access control manager* ACM smart contract, which plays a crucial role in managing permissions, roles,

and access levels for the various children. It is responsible for ensuring authorized interaction with factories and smart contracts. The second level of the hierarchy is represented by the SuperFactory contract. This entity serves as a central hub for deploying client-specific sub-factories in the hierarchy and handles the connection with the ACM to acquire permissions and roles for these sub-factories. The ACM and the SuperFactory can be considered together as a unique component in the hierarchy, since they manage the hierarchical generation of factories and smart contracts enhanced with authentication and authorization capabilities. They have been split into two smart contracts here to increase modularity, as the SuperFactory can be customized based on the application domain, and also to reduce the bytecode size of the deployed contracts.

More specifically, with reference to the motivating example introduced in Sect. 2, besides the ACM contract that will be described in more detail in the following, the water distribution authority deploys the central WMSFactory smart contract, which customizes the SuperFactory and establishes baseline permissions and oversees the creation of subordinate entities by using a hierarchical factory pattern. At the city level, a CityFactory is used to deploy and track smart contracts that govern district-level operations. Meanwhile, each district oversees a set of physical water treatment or pumping facilities through a DistrictFactory. Finally, the last level involves smart contracts that are responsible for making the information collected through IoT devices immutable. The contextualization of the hierarchy in Fig. 2 to this use case is depicted in Fig. 3.

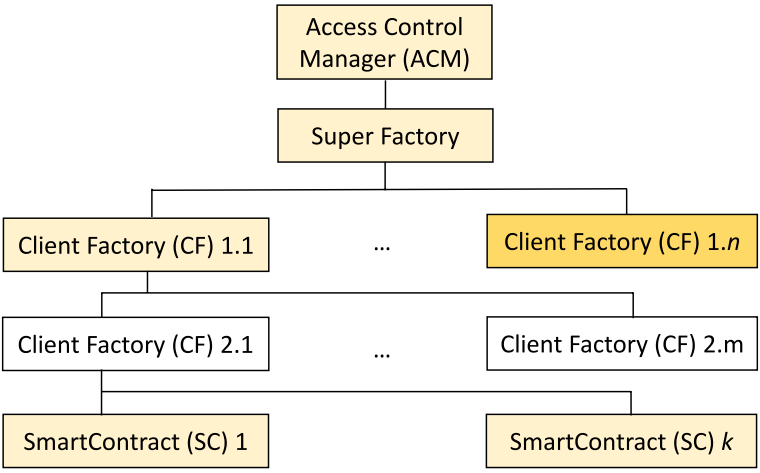

Fig. 2 The hierarchical factory pattern

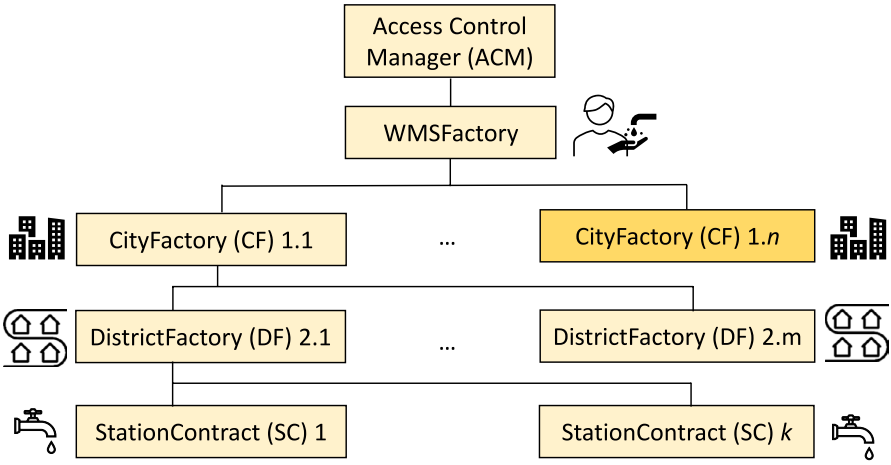

Fig. 3 Hierarchical factory pattern contextualized to the water management scenario

```
1  contract ACM :
2      public constant ADMIN;
3      public constant WATER_MGR;
4      public constant VERIFIER;
5      enum ContractType {ACM, WMSF, CF, DF, SC};
6      struct ContractData :
7          ContractType _contractType;
8          address _contractAddress;
9          address _ownerAddress;
10         address _parentContractAddress;
11     private address _adminOwner;
12     private mapping _accessControl;
13     private mapping _executedNonce;
14     constructor :
15         s ← msg.sender;
16         a ← address(this);
17         createNewContract(ADMIN, s, a, s, a, ACM);
18         createNewContract(ADMIN, s, address(new WMSF(a)), s, a, WMSF);
19         _adminOwner ← s;
20     function createNewContract(role, user, contr, owner, par, type) :
21         ContractData c ← new ContractData(type, contr, owner, par);
22         _accessControl[role][user].add(c);
23     function authnz(role, data, sign, nonce, contract) returns address :
24         // authentication and authorization
25     function addContractOnBehalfOf(contr, own, par, data, sign, nonce,
         type) :
26         address signer ← authnz(ADMIN, data, sign, nonce, type);
27         createNewContract(WATER_MGR, own, contr, own, par, type);
28         createNewContract(ADMIN, signer, contr, own, par, type);
29     function addContract(contr, parent, data, sign, nonce, type) :
30         address signer ← authnz(WATER_MGR, data, sign, nonce, type);
31         createNewContract(WATER_MGR, signer, contr, signer, parent, type);
32         createNewContract(ADMIN, _adminOwner, contr, signer, parent, type);
```

Fig. 4 Structure of the access control manager (ACM) contract

Algorithm 1 reports the structure of the ACM contract. At the beginning of this contract, it is necessary to specify the list of permission roles (lines 2-4) and the set of contract types composing the hierarchy in Fig. 2 (line 5). Depending on the specific application requirements, the number and types of both roles and contract types can be adjusted in a straightforward manner. However, the definition of the ADMIN role is mandatory and automatically assigned to the deployer of the ACM smart contract. This role assignment is a foundational security measure to ensure that the primary administrative control remains with the entity responsible for contract creation. In the case of the water management system, we define three roles: the ADMIN, the WATER_MGR, and the VERIFIER. The ADMIN has complete access to every function across all smart contracts, including creation, modification, deletion, and viewing of data at every level of the hierarchy. This

role also possesses the authority to create any new sub-factory or smart contract in the hierarchy by using the functions in the WMSFactory. This role resembles the administration user within a traditional information system, which has access to all the functionalities of the system and can also manage the roles of other users by revoking and assigning grants to them. The WATER_MGR role identifies users who are responsible for overseeing water distribution at each specific level of the hierarchy. In particular, a user assigned to this role for a CityFactory contract can add a new district factory for that city. Similarly, a user assigned to this role inside a DistrictFactory contract can create new station contracts to monitor the activities of that specific district. A WATER_MGR role also gives assigned users the right to access reading and writing functions inside the corresponding smart contract. On the other hand, the VERIFIER role has been introduced to grant read-only access to data secured in blockchain. A user assigned to this role can audit IoT sensor data for verification purposes, without altering it in any way, by calling the appropriate read-only function. Regarding the layers, the system is organized into four mandatory layers, as depicted in Fig. 3, each corresponding to a different element in the ContractType enumeration, thereby reinforcing a clear separation of responsibilities and access privileges throughout the hierarchy.

It is essential to note that both roles and layers must be predefined at design time in an ACM contract and cannot be modified after its deployment. This aligns with conventional role-based access control systems commonly found in traditional information systems, where roles and their associated permissions are identified during the requirements analysis. Moreover, this static definition of roles and layers is inherently linked to the immutable nature of smart contracts, which offer a higher level of trust and security to users, since the underlying logic and assigned roles remain immutable after deployment. At the same time, even if roles and levels in the hierarchy are predefined, the system allows dynamic reassignment of individual user addresses to the identified roles, maintaining flexibility in user management while preserving the structural integrity and security guarantees inherent to the contract design. Section 3.2 will explain authentication and authorization functions.

Given that, the ACM contract maintains two main properties: a reference *_adminOwner* to the owner of the contract, which is a general administrator for the entire hierarchy (line 11), and a structured mapping called *_accessControl* (line 12). This map maintains, for each role r and user address u, the list of contracts that u can access with the role r. Each contract is represented through a data structure *ContractData* (lines 6-10), to maintain the information related to the contract address, the address of the contract owner, the address of the parent contract, i.e. the factory that generates it, and the type of the contract, i.e. the level of the hierarchy to which it belongs. Additional information can be added, depending on the specific application requirements. The use of this data structure is essential to implement a dynamic assignment between users and roles, as well as to allow the revocation of rules to compromised or outdated users or grant roles to new users. The last maintained variable is *_executedNonce*, whose use

and importance will be described in more detail in Sect. 4.3 since it is related to the prevention of signature replay attacks [15].

The ACM constructor (lines 14-19) is responsible for three important aspects: (i) it registers the current ACM contract inside the *_accessControl* map and associates it with the ADMIN role for the *msg.sender* address. This means that the owner of the ACM contract, namely the address that performed its deployment, will act as an ADMIN user for the entire system, managing and revoking rights and roles when necessary. (ii) It creates a new instance of the WMSFactory and registers it inside the *_accessControl* map with the same role and owner, making the same address also the owner of the WMSFactory contract with ADMIN grants. Therefore, the message sender will be able to manage rights and roles and deploy any factory or contract across the entire hierarchy. (iii) Finally, it stores the address of the message sender in the *_adminOwner* variable. The updates to the *_accessControl* map are performed by the function *createNewContract*() (lines 20-22), which essentially creates a new instance of the data structure and stores it inside the map.

The function *authnz*() will be described in more detail in Sect. 3.2. It authenticates the user who has signed *data* with the signature *sign* and checks if this user has the required *role* for the *contract*. In the event of success, it returns the address of the *user*. This function is used in the last two functions, which assign role permissions to contracts deployed through the factories inside the hierarchy. In particular, the function addContractOnBehalfOf() could be invoked by the WMSFactory to deploy any contract instance anywhere in the hierarchy, both sub-factories and smart contracts, on behalf of the WATER_MGR role. Indeed, after authenticating the current transaction signer and checking that it has an ADMIN role, the function registers the given contract inside the *_accessControl* mapping by assigning administrative permissions to the signer (i.e., the WMSFactory calling the function) and WATER_MGR role to the contract owner specified as the parameter. This function enables administrators to deploy and manage contracts at any level of the hierarchy, assigning ownership and correct rights to the appropriate role. Conversely, each subfactory inside the hierarchy could invoke the function *addContract*() to deploy a DistrictFactory or StationContract, respectively. In this case, the signer is checked against the required role, and if both authentication and authorization succeed, it will be registered as the contract owner, while the ADMIN role is assigned to the ACM owner. This function enables members of the hierarchy to deploy contracts under their responsibility, while assigning the administrative role to the ACM owner.

```
1  contract WMSFactory :
2      private ACM _acm;
3      constructor (acm) :
4          _acm ← ACM(acm);
5      modifier checkAdmin() :
6          if !(_acm.hasRole(ADMIN, msg.sender, this)) then
7              revert NotAdmin();
8              _;
9      function createChildren(owner, par, data, sign, nonce, ty) checkAdmin() :
10         address c;
11         if ty = ACM.CF then
12             c ← address(new CityFactory(address(_acm)));
13         else if ty = ACM.DF then
14             c ← address(new DistrictFactory(address(_acm)));
15         else if ty = ACM.SC then
16             c ← address(new StationContract(address(_acm)));
17         else
18             revert IncorrectType();
19         _acm.addContractOnBehalfOf(c, owner, par, data, sign, nonce, ty);
```

Fig. 5 Structure of the WMSFactory smart contract

The skeleton of the WMSFactory contract is illustrated in Alg. 2. In this implementation, the contract maintains a reference to the connected ACM contract and defines a modifier, checkAdmin() (lines 5-8) which invokes the ACM verification routine to ensure that only an entity with the ADMIN role can call sensitive functions. This security measure guarantees, in particular, that only an ADMIN user can invoke the *createChildren*() function (lines 9-19). Based on the parameter *ty* given as input and referring to the *ContractType* enumeration in line 5 of Alg. 1, the function can create any factory or smart contract within the hierarchy, returning the corresponding contract address. Finally, the new contract is registered in the ACM *_accessControl* mapping on behalf of the instantiated factory (line 19) by calling the function addContractOnBehalfOf() described previously. In the context of the water management system illustrated in Fig. 3, this design enables the ADMIN to deploy subordinate factories, such as CityFactory and DistrictFactory, as well as StationContract reflecting the organizational structure of a water distribution system in which the distribution authority can manage all subordinate organization levels. The WMSFactory not only streamlines contract creation but also ensures that the hierarchical deployment of smart contracts follows the access control criteria outlined in earlier sections. Notice that the reference to the ACM contract is set by passing its address in the WMSFactory constructor and performing a cast. This operation is secure and does not introduce vulnerabilities in the contract since the constructor is invoked only once during the factory deployment by the ACM itself. Furthermore, the immutability of this reference is guaranteed by the *private immutable* modifier, which prevents external contracts from altering it after deployment.

Further levels in the hierarchy are implemented through domain-specific factories tailored to deploy and manage smart contracts following the organizational structure of the water management system. At the city level, the CityFactory (see Alg. 3) is responsible for the implementation and tracking of smart contracts for district-level operations and the management of city data. In particular, each *CityFactory* defines a mapping (line 5) that records aggregated data collected from all city districts under its responsibility each year, using such information as the key. While the function addCityData() could be invoked only by the user having WATER_MGR role for this specific contract, the corresponding reading function *getCityData*() could also be invoked by users registered with a VERIFIER role for it. This is obtained by the definition of two modifiers: checkManager() to restrict function calls to users with WATER_MGR role for this contract and checkManagerOrVerifier(), which allows users with the VERIFIER or WATER_MGR role for the contract to invoke its read-only functions. To simplify the notation, the aggregated data stored at the city level have been synthetically described by *cityData*. However, it can be extended and detailed based on the specific needs. Finally, the CityFactory can instantiate subordinate *DistrictFactories* through the function createContract(). This function deploys a new DistrictFactory and registers it in the ACM mapping through the function addContract().

```
1  Contract CityFactory :
2      private ACM _acm;
3      constructor (acm) :
4          _acm ← ACM(acm);
5      private mapping _aggregatedData;
6      modifier checkManager() :
7          if !(_acm.hasRole(WATER_MGR, msg.sender, this)) then
8              revert NotAuthorized();
9              _;
10     modifier checkManagerOrVerifier() :
11         if !(_acm.hasRole(WATER_MGR, msg.sender, this) ∨
               _acm.hasRole(VERIFIER, msg.sender, this)) then
12             revert NotAuthorized();
13             _;
14     function createContract(parentAddr, data, sign, nonce) checkManager() :
15         _acm.addContract(address(new DistrictFactory(address(_acm))),
             parentAddr, data, sign, nonce, DF);
16     function addCityData(year, cityData) checkManager() :
17         _aggregatedData[year] ← cityData;
18     function getCityData(year) checkManagerOrVerifier() :
19         return _aggregatedData[year];
```

Fig. 6 Structure of the CityFactory contract

The DistrictFactory described in Alg. 4 represents the subsequent layer in the hierarchy, and follows a similar design. In particular, the differences reside in the createContract() function, which this time creates an instance of a StationContract, as well as in the type and frequency of data collection. In particular, the *districtData* mapping collects aggregated monthly data from the water stations under the control of the specific district. DistrictFactory employs the checkManager() modifier to restrict contract creation to users with the WATER_MGR role for the contract, while functions to retrieve the stored data are protected by the checkManagerOrVerifier() modifier. Overall, CityFactory and DistrictFactory extend the hierarchical factory pattern by ensuring that smart contracts at each level comply with the overall access control policy and manage water data at the appropriate level of granularity. Therefore, the proposed hierarchical factory pattern offers much more flexibility than traditional smart contract factories proposed in the literature, which can only create new instances of the same smart contract type.

```
1  Contract DistrictFactory :
2      private ACM _acm;
3      private mapping _districtData;
4      constructor (acm) :
5          _acm = ACM(acm);
6      modifier checkManager() :
7          if !(_acm.hasRole(WATER_MGR, msg.sender, this)) then
8              revert NotAuthorized();
9              _;
10     modifier checkManagerOrVerifier() :
11         if !(_acm.hasRole(WATER_MGR, msg.sender, this) ∨
               _acm.hasRole(VERIFIER, msg.sender, this)) then
12             revert NotAuthorized();
13             _;
14     function createContract(parentAddr, data, sign, nonce) checkManager() :
15         _acm.addContract(address(new StationContract(address(_acm))),
             parentAddr, data, sign, nonce, SC);
16     function addDistrictData(date, districtData) checkManager() :
17         _districtData[date] ← districtData;
18     function getDistrictData(date) checkManagerOrVerifier() :
19         return _districtData[date];
```

Fig. 7 Structure of the DistrictFactory contract

```
Contract StationContract :
    private ACM _acm;
    private mapping _stationData;
    constructor (acm) :
        _acm = ACM(acm);
    modifier checkManager() :
        if !(_acm.hasRole(WATER_MGR, msg.sender, this)) then
            revert NotAuthorized();
            _;
    modifier checkManagerOrVerifier() :
        if !(_acm.hasRole(WATER_MGR, msg.sender, this) ∨
         _acm.hasRole(VERIFIER, msg.sender, this)) then
            revert NotAuthorized();
            _;
    function addStationData(date, dataCID, dataHash) checkManager() :
        _stationData[date] ← dataCID, dataHash⟩;
    function getStationData(date) checkManagerOrVerifier() :
        return _stationData[date];
```

Fig. 8 Structure of the StationContract

To conclude, a prototypical smart contract belonging to the leaf level is presented in Alg. 5. StationContract is responsible for interacting with sensor networks and physical water treatment facilities. Since it represents the last level of the hierarchy, it does not present any function for creating child contracts or factories. However, it only includes functions for collecting and retrieving data. In line with the previous contracts, access to these functions is controlled by role-specific modifiers. Due to the high volume of data produced by each station, whose frequency could be very fine-grained, we decided to store the raw data within IPFS and make it immutable by storing its content identifier (CID) and hash in the blockchain. Further details about this are out of the scope of this paper, and so are omitted.

The proposed design pattern offers a more flexible structure than the traditional factory pattern, as it can accommodate multiple layers, each handling specific functions and tasks. The proposed design pattern is scalable, since deploying a new contract instance requires adding a new reference only in the appropriate parent factory contract, without altering the rest of the smart contract hierarchy. In other words, newly created instances are distributed across parent factories without increasing the complexity of unrelated contracts. The other important aspect, which will be discussed in more detail in the following section, is related to the separation of roles and grants, that mimic the physical and administrative structure of a water management system: the ADMIN can do anything that includes the deployment of factories; the WATER_MGR can both create subordinate contract instances and update critical off-chain references; and the VERIFIER role allows secure, read-only access to ensure data integrity. This approach enhances modular deployment and granular access con-

trol while also establishing a verifiable relation between on-chain contracts and off-chain sensor data.

3.2 Multirole Authentication and Authorization

This section addresses the issue of authenticating and authorizing users within the hierarchical structure outlined in the previous section. In particular, regarding authentication, it is worth noting that the emergence of blockchain technology has induced a paradigm shift in the digital security domain, from traditional centralized authentication and authorization systems to decentralized mechanisms. In fact, traditional information systems typically rely on a Role-Based Access Control (RBAC) mechanism, which is often based on centralized authorities or servers to verify user identity, thereby exposing the system to a single point of failure, data breaches, and unauthorized access. Conversely, in the blockchain domain, no central authority is present, and there is a need to implement robust decentralized multirole authentication and authorization mechanisms that distribute authentication information across a network of nodes, thereby making it difficult for malicious actors to undermine system integrity.

```
contract ACM :
    function verify(data, sign) returns address :
        return data.toEthSignMessageHash().recover(sign);
    function hasRole(role, user, contract) returns bool :
        ContractData[] cs ← _accessControl[role][user];
        return cs.contains(contract);
    function authnz(role, data, sign, nonce, contract) returns address :
        byte32 hash ← keccak256(data, nonce);
        if (_executedNonce[hash]) then
            revert AlreadyExecuted();
        address signer ← verify(hash, sign);
        if (!hasRole(role, signer, contract)) then
            revert NotAuthorized();
        _executedNonce[hash] ← true;
        return signer;
    function revokeRole(role, user) onlyAdmin() :
        if (user = address(0)) then
            revert InvalidAddress();
        if (user = _adminOwner) then
            revert NotAuthorized();
        _accessControl.delete(⟨role, user⟩);
    function transferOwnership(address newAdmin) onlyAdmin() :
        if (newAdmin = address(0)) then
            revert InvalidAddress();
        if (newAdmin = _adminOwner) then
            revert NewAdminSameAsCurrent();
        ContractData[] prev ← _accessControl[ADMIN][_adminOwner];
        ContractData[] curr ← _accessControl[ADMIN][newAdmin];
        for (i = 0; i < prev.length; i++) do
            curr.push(prev[i]);
        _accessControl.delete(⟨ADMIN, _adminOwner⟩);
        _adminOwner ← curr;
    function addVerifier(verifier, contract) onlyAdmin() :
        if (verifier = address(0)) then
            revert InvalidAddress();
        _accessControl[VERIFIER][verifier].add(contract);
```

Fig. 9 Structure of the access control manager ACM contract

As already mentioned in Sect. 3.1, the proposed solution implements the multi-role authentication and authorization mechanism in the ACM contract. In particular, ACM uses the *authnz*() function to authenticate users and check their roles through the *_accessControl* private mapping. The details of this mechanism and functionalities are reported in Alg. 6.

Before proceeding with the description of *authnz*(), it is necessary to note that, in the absence of a central server, a blockchain-based system can exploit the transaction signature mechanism to provide authentication. A simple authentication inside the wallet application is not enough to provide strong and secure authentication, since a wallet is a client tool and can be compromised by a malicious user. Therefore, it is necessary to implement a mechanism that combines smart contract functionalities and wallet signature capabilities to establish a robust decentralized authentication mechanism, which also involves the network nodes. This approach requires users to actively sign a message with their private key in the wallet, generating cryptographic proof of their identity. The signed message is then verified with the *verify*() function illustrated in lines 2-3 of Alg. 6. This function takes two inputs: a hashed message *data* and a signature *sign*, from which it can retrieve the address of the user signing the message *data*. In order to do that, we exploit the *recover*() function provided by the OpenZeppelin's Elliptic Curve Digital Signature Algorithm (ECDSA) library.[1] This method allows the retrieval of the address that signs a message and then confirms that the message comes from an expected address. Indeed, without this authentication procedure, the signature can be forged by a malicious user, producing a mismatch between the *msg.sender* and the address that signs the message. In other words, the message was breached and not sent directly from the person from whom it was expected to be sent. Conversely, the success of this check requires that the transaction's sender is indeed the individual who signed the message hash. Although the current design leverages OpenZeppelin's ECDSA-based recovery due to its optimized implementation via the use of EVM precompiled *ecrecover*, future extensions of the proposed solution may consider alternative signature schemes, such as the use of the EIP-2098 short signature format or other cost-effective cryptographic methods, to further reduce gas consumption without compromising security.

The *verify*() function is used by *authnz*(), which performs both authentication and authorization (lines 7-15). Before safely retrieving the signer address (line 11), the function uses a nonce to overcome a potential signature replay attack (line 8-10). This mechanism will be explained in more detail in Sect. 4.3. After that, the *authnz*() function checks if the signer has the required role for a contract through the *hasRole*() function (lines 4-6), by accessing the content of the *_accessControl* mapping. As discussed in the previous section, the proposed solution employs a hierarchical access control mechanism, rather than relying on a flat permission model that maps user addresses and roles to specific contract addresses. This defines clear access boundaries for roles at each level of the hierarchy. When a user attempts to access a contract, the system verifies the user's role and determines whether the user is authorized to access the specific contract within the hierarchy.

The *revokeRole*() function (lines 16-21) can dynamically revoke roles from users and could be invoked only by the ADMIN to remove any role different from itself. This limitation is necessary to avoid losing control over the hierarchy. The function accepts two parameters: a role *role* and a user address *user*, to which we want to revoke the role privilege *role* for the previously assigned smart contracts. The func-

[1] https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/cryptography/ECDSA.sol.

tion initially verifies that the user address is valid (i.e., different from zero) and is not the administrator address (lines 18-20), then it removes role *role* from the user *user* in the access control mapping. This function provides an additional layer of control to the ADMIN, allowing effective role management and maintaining the integrity of ACM in a dynamic environment where user permissions may need to be updated frequently.

The ACM incorporates also a function to transfer its ownership, namely the transfer of the ADMIN role. The *transferOwnership*() function can only be invoked by the current ADMIN and transfers the ownership to a new ADMIN (lines 22-32). That function initially verifies if the new ADMIN address is valid (nonzero) to ensure that ownership is transferred to a valid address and that the new address differs from the previous one. Then, all the contracts associated with the old ADMIN role are transferred to the new ADMIN in the *_accessControl* mapping. The *transferOwnership*() function is critical for ensuring the security of ACM, particularly in the case of a compromised or exposed ADMIN private key. Indeed, in this case, an attacker could take control of the overall hierarchy and act on any contract and factory inside the tree. Therefore, its usage could be carefully evaluated and eventually excluded from the ACM implementation.

Finally, the *addVerifier*() function allows the ACM to add read-only permissions to external contracts by managing the VERIFIER roles associated to a specific contract.

The multirole authentication and authorization framework not only ensures secure access to smart contracts in the hierarchy and role-specific permissions but also preserves the decentralized and trustless nature of the system, which is crucial for sensitive infrastructure such as water management applications.

The next section will discuss the implementation details of the proposed solution and will evaluate it from both a performance and a security perspective.

4 Implementation Performance and Security Analysis

In the experimental setup of our study, we use the Ethereum Sepolia test network to evaluate the proposed Hierarchical Factory Pattern in a more realistic environment, using testnet Ethers. We developed smart contracts in the Solidity language, version 0.8.40, in the Hardhat deployment environment. We employed the Metamask wallet for blockchain interaction, smart contract deployment, and transactions. Finally, the dApp layer has been developed using the React.js and Ether.js libraries. Table 1

Table 1 Tools and technologies used in the experimental setup

Technology	Version/network
Ethereum test network	Sepolia
Solidity	0.8.40
Contract deployment	Hardhat
Blockchain interaction	MetaMask Wallet
DApp development	Ethers.js
Frontend framework	React+TypeScript
Bytecode optimizer	true

summarizes all the tools and technologies used, source code is available in a GitHub repository.[2]

The following section discusses the implementation of the proposed hierarchical factory pattern from different perspective: the optimization of the developed smart contracts (Sect. 4.1), the evaluation of their performance with respect to bytecode size and Gas costs (Sect. 4.2), and the ability of the solution to overcome two well-known security attacks: the signature replay attack, and the dynamic revocation and transfer of ownership (Sect. 4.3).

4.1 Optimization of Smart Contracts

The optimization of smart contract code is a critical aspect of any real-world decentralized application since it has effects on both the Gas cost during deployment and the overall bytecode size, which has to comply with predefined limits. Indeed, the Gas cost G_i for the deployment of a contract C_i could be computed as $G_i = G_{\text{base}} + n_i \times G_{\text{byte}}$, where G_{base} is the base transaction cost, n_i is the bytecode size of C_i, and G_{byte} is the cost per byte. Therefore, the bytecode size n_i greatly impacts G_i, and smart contracts must be properly designed to reduce this amount.

The presented solution addressed this problem from several perspectives: first of all, the proposed hierarchical factory pattern itself represents a way to reduce the bytecode size and increase scalability, since some common parts could be isolated inside the factory and called by all the other factories or smart contracts in the hierarchy. Section 4.2 will experimentally analyze this aspect by comparing the Gas costs induced by the proposed solution with respect to a flat solution that does not use factories, as the total number of deployed contracts increases.

In addition to this, another efficient approach is the use of libraries to isolate some reusable behaviors. In particular, the WMSFactory in Alg. 2 could exploit methods included in some libraries to create the various subfactories within the *createNewContract*() function. Specifically, when compiled monolithically, the WMSFactory bytecode exceeds the contract size limit of 24 KB (24576 bytes) imposed by Ethereum to protect against DoS risks. Since the library code is stored in a separate contract, the size of the WMSFactory is significantly reduced below the 24 KB limit, decreasing the cost of deploying and interacting with the contract.

Additionally, we optimized the code by replacing standard *require* statements with custom error handling introduced in Solidity version 0.8.4. The *require* statement, which stores the entire revert reason as a string, increases bytecode size and Gas consumption. Custom errors allow one to define typed error codes that consume significantly less Gas when reverting. This minimizes the size of the deployed bytecode and the cost of transaction reverts, particularly in functions secured by role checks, which are frequently triggered in hierarchical contracts. In our implementation, we replaced *require* with *custom errors* in all access checks, resulting in a reduction of approximately 134-446 bytes per contract in the deployed bytecode. This corresponds to a decrease of 4.3% to 6.3%, resulting in direct Gas savings during both deployment and revert operations. These reductions are particularly important for

[2] https://github.com/MuhammadBinSaif/Extension-Hierarchical-Factory-Pattern-in-Smart-Contracts.

role-protected modifiers and other security-critical checks, which are frequently used in our hierarchical contracts.

Finally, in smart contract optimization, a strategic ordering of state variables can significantly reduce the Gas cost. The EVM allocates state variables in storage slots, where each slot occupies a 32-byte. Moreover, it sometimes inserts padding after large types to maintain consistent alignment, especially when a small type (such as an enumeration, which is 1 byte) appears after multiple large types (like strings, which occupy 20 bytes each). It follows that with the organization of the *ContractData* structure as presented in Alg. 7 for our specific application, the structure will occupy 32 bytes $\times$ 3 for storing the first three addresses and an additional slot of 32 bytes is needed to store the last enumeration attribute, even if it requires only 1 byte, for a total of 128 bytes. Conversely, if we place the *_contractType* variable above the addresses, the enumeration can be placed within the same storage slot as the first address, and the structure now requires other $32 \times 3 = 96$ bytes.

```
1 struct ContractData :
2     address _contractAddress;
3     address _ownerAddress;
4     address _parentContractAddress;
5     ContractType _contractType;
```

Fig. 10 Contract data structure

4.2 Performance Evaluation

We evaluated the performance of the proposed solution with a particular focus on two critical aspects: the bytecode size of the deployed contracts and the Gas cost required for contract execution and deployment. These two metrics are crucial for evaluating the efficiency and resource allocation of the proposed pattern within an Ethereum network. In particular, bytecode size measures the complexity of the contract and the amount of code processed by the EVM, while Gas consumption represents the computational resources required for contract deployment and execution. This evaluation compares the proposed hierarchical solution with a traditional factory pattern. Moreover, we will consider both the traditional Layer 1 Ethereum EVM and a Layer 2 solution, represented by the Polygon zkEVM, to assess the solution's feasibility in real-world application domains.

To better study the effect of the hierarchical structure, we reproduced the organization of a water management system serving the municipality of Verona, a city in northern Italy. In our hierarchy, the top-level CityFactory oversees 12 DistrictFactories, with each district handling from 7 to 15 StationContracts, for a total of 124 stations. We developed a three-tier structure (i.e., city, district, station) based on the roles described in Sect. 2, focusing on the key operations of role checks, contract deployment, and data aggregation. We deployed the full hierarchy on the Sepolia testnet and measured bytecode size and Gas for deployment and typical management operations (role assignment, station registration, yearly and monthly data uploads). Table 2 illustrates the Gas consumed for the deployment and the bytecode size of

Table 2 Bytecode size and estimated gas cost of smart contract

Smart contract	Bytecode size (bytes)	Estimated Gas Cost (gwei)
ACM	7,452	1,619,208
WMSFactory	1,607	374,400
CityFactory	7,373	1,530,644
DistrictFactory	4,789	1,291,439
StationContract	2,003	932,240

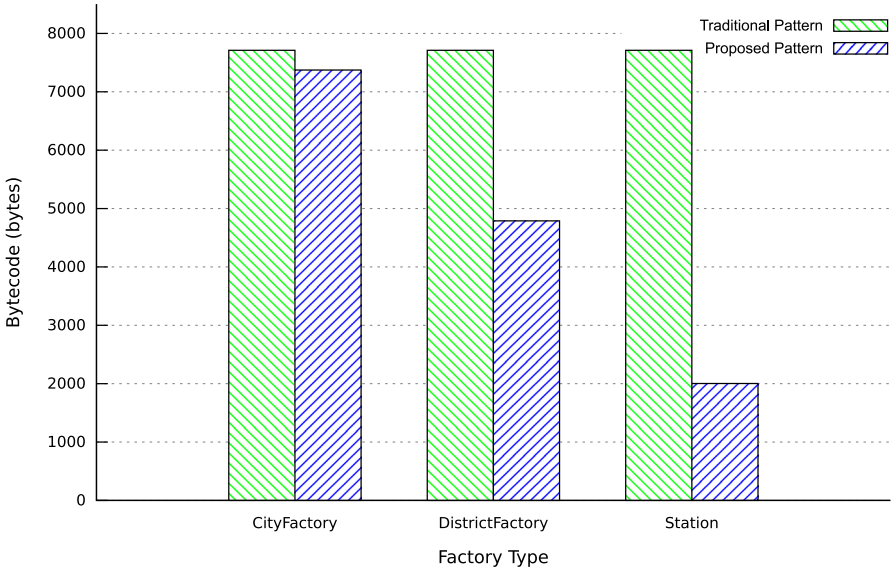

Fig. 11 Bytecode comparison of the proposed vs the traditional factory pattern

each factory in the hierarchy, such as the ACM, the WMSFactory, and the three levels, namely CityFactory, DistrictFactory, and StationContract. We can observe that the bytecode size of all proposed smart contracts is below the EVM limit, which has been fixed to 24 Kbytes to mitigate security attacks, and the size of each factory decreases as the depth in the hierarchy increases.

As shown in Fig. 11, the traditional factory pattern generates an identical bytecode size of 7,711 bytes for each level CityFactory, DistrictFactory, and StationContract. This is because each contract independently includes the full logic required for contract creation and access control. In contrast, the hierarchical factory pattern concentrates the main logic at the highest level, resulting in smaller bytecode sizes at lower levels. Particularly, CityFactory remains relatively large at 7,373 bytes since it contains references to all subordinate contracts, while DistrictFactory and StationContract sizes reduce to 4,789 bytes and 2,003 bytes, respectively. Therefore, by centralizing common functionality at the root and minimizing redundancy, the hierarchical approach reduces the overall bytecode size compared to the traditional factory.

The cost in currency for different operations is measured by converting the amount of *GasUsed* into USD by considering the unit Gas price of two networks: the Ethe-

reum mainnet, and the Polygon zkEVM, which is a Layer 2 solution. At the time of the experiment, the median base-fee for Ethereum mainnet was 1.382 gwei at an exchange rate of $1\ \mathrm{ETH} = 1733.68$ USD, and for Polygon zkEVM, the base-fee was 513 gwei and $1\ \mathrm{POL} = 0.2257$ USD. The conversion is performed with the following formula:

$$\mathrm{Cost}_{\mathrm{USD}} = \mathrm{GasUsed} \times \mathrm{GasPrice}_{\mathrm{gwei}} \times 10^{-9} \times \mathrm{tokenUSD}$$

Figure 12 shows the deployment cost of our three-layer hierarchy (i.e., CityFactory, DistrictFactory, and StationContract), and compares it with the deployment cost induced by a conventional single-factory pattern, on both the Ethereum net and the Polygon zkEVM network. As can be observed, with the proposed solution, the CityFactory deployment is approximately 8% more expensive than a traditional factory, since the contract includes both access control logic and shared libraries. However, this extra cost decreases at the DistrictFactory level, where our approach is 10% cheaper, and at the StationFactory level, which accounts for most of our contracts, it is 34% cheaper. Indeed, the traditional pattern requires the redeployment of a whole factory (about 3.0$ on the Ethereum mainnet) for each new district or station, repeating bytecode and role checks. In contrast, the hierarchical factory pattern reuses parent logic and deploys only a light child contract. Since the DistrictFactory and the StationContracts are the most frequently added or interacted with, a one-third cost reduction at this level directly reduces operational costs for water network operators. Regarding the comparison between the costs on the Ethereum net and the Polygon one, we can observe that a significant cost reduction can be achieved in the former, where general functioning costs are higher, and obtaining a reduction is particularly important.

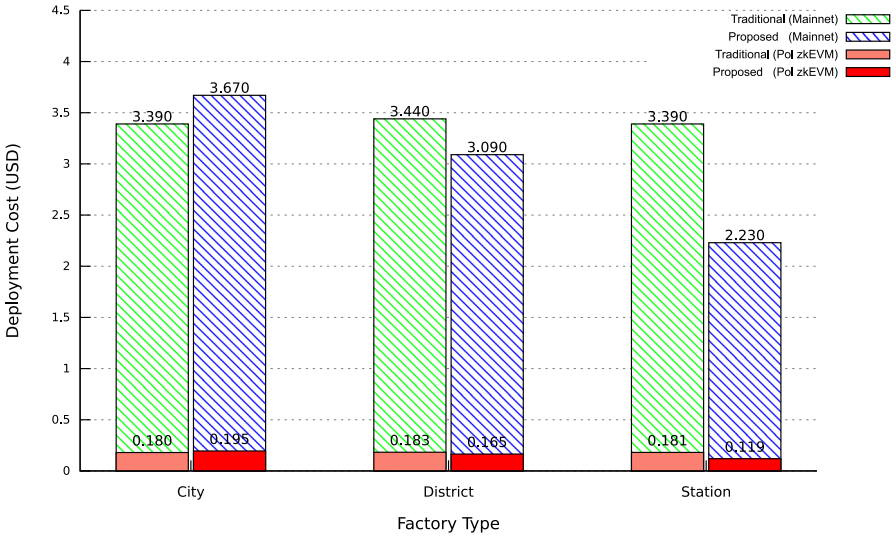

Fig. 12 Comparison of the USD costs needed for the contract deployment in the proposed hierarchical factory pattern approach vs the traditional factory pattern

Figure 13 compares the USD cost of four function calls addVerifier(), revokeUser(), transferOwnership(), and addData() on both the Ethereum EVM and the Polygon zkEVM. On Ethereum net, the transferOwnership() function is the most expensive operation, costing 1.33$, while the simple data upload (i.e., addData()) requires only 0.07$. The two remaining access-control calls cost 0.28$ for addVerifier() and 0.11$ for revokeUser(). Executing the same functions on Polygon zkEVM significantly reduces cost, such as transferOwnership() drops to 0.07$, addVerifier() to 0.02$, revokeUser() to 0.006$, and addData() to a negligible 0.004$. We can also observe that while the first three functions are rarely called, since they deal with granting and revoking of user roles and permissions, the last one (i.e., addData()) is the most frequently used in a real-world application, since it is used to store the water management data. The cost of its invocation is quite low compared to the other three and, when deployed in a Layer 2 solution, the fees are also economically feasible in case of frequent invocation.

Finally, Fig. 14 presents a detailed comparison of Gas costs for user management operations with the ACM smart contract with respect to the results obtained by applying the solutions in [16] and [17]. In this case, two types of operations have been considered: adding a new user and removing an existing one. For the first operation, different amounts of informative data were passed to the method. The results show that the proposed approach outperforms both baselines even if the Gas cost increases with the amount of data processed by *addUser*(). These results confirm the efficient behavior of the proposed approach for the *removeUser*() method as well.

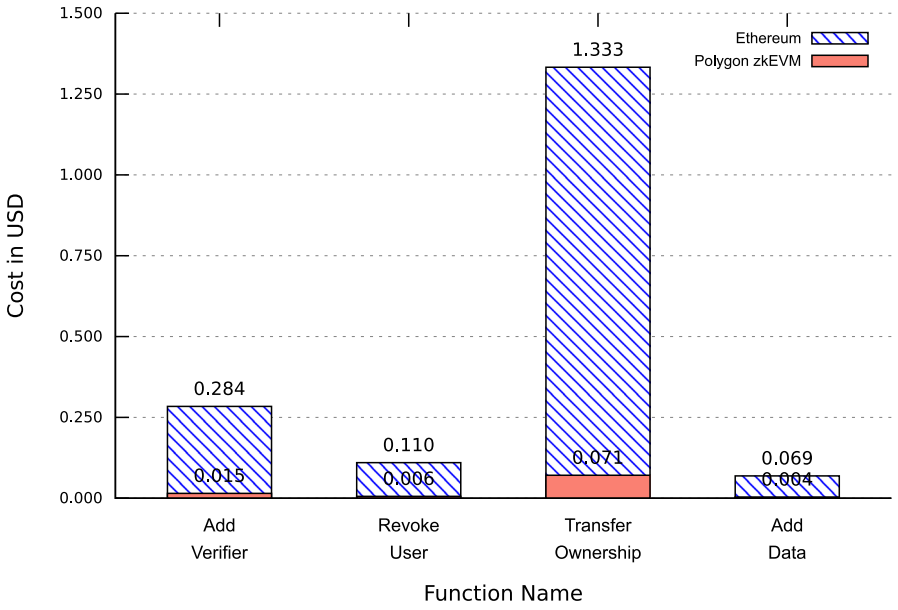

Fig. 13 USD cost of function calls

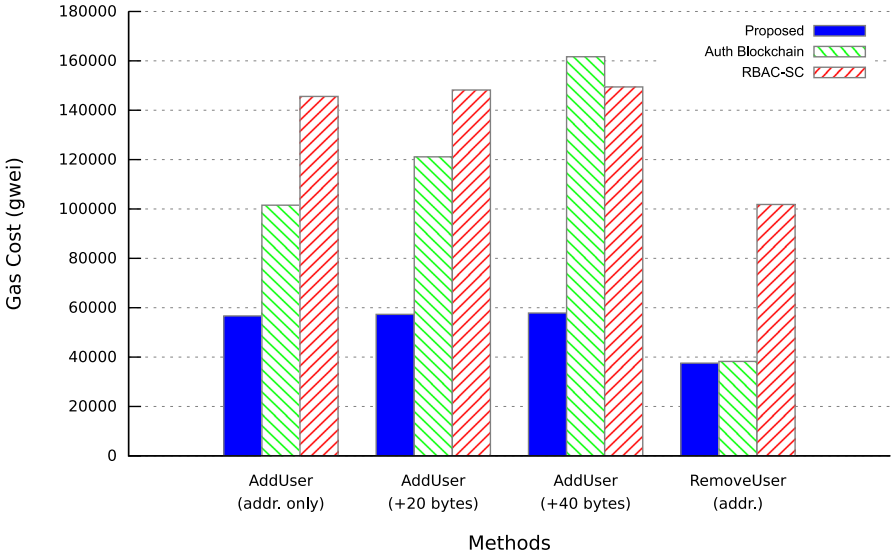

Fig. 14 Gas cost comparison of the proposed approach vs [16] and [17]

4.3 Security Evaluation

We finally evaluate how the proposed solution deals with two important security aspects, namely the prevention of signature replay attacks, and the permission of dynamic revocation and transfer of ownership in a secure way.

In the ***signature replay attack*** [15], a malicious user reuses a previously valid signature to illegitimately re-execute an operation. This kind of attack can affect the authnz() function of the *ACM* (see Alg. 6, lines 7-15) which is responsible for user authentication and authorization. To prevent this form of attack, our system incorporates a nonce-based security approach into the authentication process. In particular, each transaction invoking *authnz*() passes to it a nonce value, which is combined with the signed data to generate a unique hash (lines 9). The resulting hash is then checked against the *executedNonce* mapping, which records previously used signature hashes. The transaction immediately reverts if the hash already exists, preventing signature reuse. This approach ensures that every signed message is valid for a single execution, preventing signed messages from being subject to replay in function calls or contract contexts. Furthermore, the integration of the nonce argument in factory deployment and access control functions, particularly the *addContractOnBehalfOf*() and *addContract*() functions in Alg. 1, guarantees the constant mitigation of replay attacks in all tiers of the hierarchical factory pattern. This is crucial for the security of critical infrastructure applications, such as water management systems, since unauthorized access could compromise the integrity and confidence of the entire operating hierarchy.

Relative to the ***dynamic revocation and transfer of ownership*** attack, we can observe that the integrity and safety of the entire hierarchy strictly depend on the validity and safety of the accounts authorized to deal with the contracts inside the

hierarchy. However, in many real-world applications, some users can leave an organization or change their role inside it, or in some cases, their account could be compromised, and new authentication credentials need to be registered. To address both situations, the proposed solution provides a functionality, for the ADMIN, to dynamically revoke user roles, particularly in the case of a compromised account or wallet. In particular, the ADMIN can revoke any role assigned to a user in all previously authorized contracts by using the *revokeRole()* function. The system does not allow the revocation of the ADMIN role; however, if the private key associated with the ADMIN account is compromised, the owner can, as a last resort, invoke the *transferOwnership()* function to reassign administrative privileges to a new address securely. Notice that in this way, if the ADMIN private key has been stolen, a malicious user could potentially reassign all the hierarchy to him, before the administrative role has been properly reassigned. To further enhance security, the ADMIN role can also be connected to a multisignature (multisig) wallet, thereby requiring multiple user approvals to authorize sensitive actions such as the transfer of ownership, without changing the structure of the proposed smart contract, while reducing the risk associated with a single compromised key. However, this solution requires more Gas for performing each ADMIN operation and can be chosen or not, based on the application domain, transparently. By enabling dynamic role revocation and ownership transfer, the system can adapt quickly to potential security threats or structural adjustments without requiring the redeployment of the smart contract hierarchy. This ensures that the system remains robust and operational under various circumstances.

5 Related Work

This section summarizes recent work on the definition of both design patterns for smart contracts and solutions for managing authentication and authorization in this field, as well as the development of dApps dealing with critical systems, such as water management ones.

5.1 Smart Contract Design Patterns

In [12] Rajasekar et al. explore various design patterns developed for blockchain applications. Among these, the most important for our proposal is the *factory contract* pattern. Factory pattern stores a template contract on the blockchain, called the factory, from which it is possible to instantiate similar child contracts to improve modularity and consistency. The other patterns proposed in the paper are: the *checks-effects-interactions* pattern, which emphasizes secure transaction sequences, the *oracle* pattern focusing on integrating external data into the blockchain, the *off-chain data storage* and the *state channel* patterns, which together address the challenges of efficient data storage strategies, minimizing on-chain data while ensuring integrity. Finally, the authors propose the *contract registry* pattern, which allows flexibility in updating contract logic, and the *emergency stop* pattern, which introduces a safety mechanism against malicious activities. All these patterns demonstrate evolving best

practices in blockchain applications design, emphasizing the importance of security, efficiency, and adaptability.

With reference to the design patterns for specific application domains, in [18] Zhang et al. investigate their application to enhance the design and functionality of blockchain-based healthcare systems. They identify the following four main patterns: abstract factory pattern, flyweight pattern, proxy pattern, and publish-subscriber pattern. The *abstract factory* pattern provides a structured approach to creating user accounts. It allows the instantiation of objects without specifying exact classes and simplifies the creation of new related contracts for departments and subdivisions. The *flyweight pattern* addresses the critical challenge of large data storage for smart contracts by sharing data across similar objects. It is useful in scenarios where large amounts of patient data, such as insurance and billing information, need to be stored efficiently. The *proxy pattern* serves as a placeholder to ensure patient data privacy by giving access to metadata. This approach maintains an audit trail for data operations, enhancing transparency and data integrity on blockchain. The *publisher-subscriber* pattern improves information broadcasting by notifying relevant stakeholders, such as prescription requests.

Relatively to the factory pattern, Wu et al. in [13] emphasize its significance in smart contract design. In particular, they demonstrate its usefulness for applications that require consistent behaviors to be replicated across multiple smart contracts. Additionally, the implementation of a factory contract pattern in 28 dApps and 2671 smart contracts highlights its pivotal role in dApp development.

Compared to the traditional factory pattern already present in the literature, the hierarchical factory pattern proposed in [14] offers an improved, structured, and scalable approach. Namely, while the traditional factory or abstract factory patterns can only deploy similar instances of smart contracts, the hierarchical factory pattern can deploy diverse factories and contracts. This makes it more versatile and adaptable to diverse applications, as well as more compatible with complex hierarchical organizational structures.

5.2 Authentication and Authorization in Smart Contracts

An important aspect of smart contract design is the definition of who can access and execute functions and data. However, the decentralized and transparent nature of blockchain makes it challenging to implement access control. Several design patterns have been proposed to address the access control challenge in smart contracts. The *role-based access control* (RBAC) pattern is one of the access control design patterns that assign the roles to different users and grants permissions to the roles. In [17], Cruz et al. present RBAC-SC, which employs Ethereum smart contracts to model trust and endorsement relationships among roles, organizations, and services. The scheme also includes a challenge-response authentication protocol to confirm the ownership of user roles. This design is effective for flat hierarchies but does not support recursive factories or multi-layered governance. Another access control design pattern is the *token-based access control* (TBAC), that employs tokens to represent user access rights. The TBAC enables users to transfer or delegate access rights by transferring tokens. In [19], Liu et al. introduce the SMACS framework, which allows low-cost

implementation of updatable and sophisticated access control rules (ACRs) for smart contracts. It reduces Gas costs by 22% compared to RBAC-SC. However, SMACS focuses on static roles and cannot adapt permissions dynamically, based on real-time sensor data, a limitation in IoT-driven systems such as water management. In [20], Zhou et al. propose a smart contract-based ownership access control framework for the Internet of Things (IoTs) in smart home systems. The proposed framework utilizes the ownership design pattern to define a hierarchy of owners, including device, home, and system owners. Ding et al. in [21] propose Bloccess, a blockchain-based access control scheme for IoT systems. While this work effectively moves authentication to the chain, it relies on a flat access control structure, which is not suitable for multi-layered administrative domains such as water management. Similarly, Zhang et al. introduce in [22] a dynamic access control scheme for Industrial IoT (IIoT) that utilizes smart contracts to enforce privacy protection and adaptability, addressing the strict access control of traditional RBAC systems.

To address the limitations of flat permission models, recent research has shifted towards hierarchical access control structures that better reflect real-world administrative domains. For instance, Ma et al. in [23] propose a privacy-oriented distributed key management architecture (BDKMA) specifically designed for hierarchical access control in IoT. This approach utilizes a multi-blockchain structure spanning cloud and fog layers to reduce latency and enable cross-domain access. The system manages complex permission inheritance by using distinct authorization assignment modes (private, protected, and public), eliminating the need for a fully trusted third party. However, this architecture relies heavily on specific cryptographic key management transactions rather than the modular smart contract factories. Similarly, Abdi et al. in [24] address scalability in IoT systems by proposing a hierarchical blockchain architecture based on Hyperledger Fabric. This solution utilizes three distinct layers: Edge Blockchain Managers (EBCM) for local authentication, Aggregated Edge Managers (AEBCM) for cluster management, and a Cloud Consortium Manager (CCBCM) for external user access. This solution achieved lightweight security for constrained devices by employing a multi-chaincode structure and Attribute-Based Access Control (ABAC). However, while the multi-layer manager approach improves scalability, it requires a complex permissioned infrastructure with specific managers (i.e. the EBCM and the AEBCM).

To the best of our knowledge, while hierarchical access control has been explored in distributed key management and permissioned blockchain architectures, no existing solution combines a hierarchical factory pattern with a Role-Based Access Control (RBAC) mechanism on EVM-compatible chains. Unlike the complex permissioned infrastructures or key-management overlays proposed in recent literature, this paper introduces a hierarchical factory pattern for the direct management and deployment of smart contracts. This novel approach integrates Multirole Authentication and Authorization (MAC) directly into the factory lifecycle, offering a more modular, scalable, and secure solution for decentralized critical infrastructure.

5.3 Blockchain and Smart Contracts in Critical Systems

Blockchain technology has gained significant attention in critical infrastructure due to its ability to ensure transparency, immutability, and decentralized trust. In energy management, Xia et al. propose in [25] a blockchain framework for intelligent water systems, focusing on data integrity through hash storage on the chain. However, this work lacks granular access control, relying instead on a centralized administrator for permissions, which is a critical flaw in distributed systems. Biswas et al. propose in [26] the WaDA concept, in which the WAMO 300 IoT platform is combined with a proposed blockchain-based architecture to support water diplomacy and to secure real-time surface water observations against tampering. Their work addresses the integrity and transparency of hydrological data in politically sensitive water-sharing contexts; however, it does not model the multi-level administrative hierarchies required for effective facility management. Similarly, in [27] Alrammal et al. present a unified permissioned blockchain solution for managing both electricity and water services, utilizing smart contracts to automate key processes such as billing and payment, while enabling privacy-preserving online transactions for customers. However, their design focuses on interactions among a small set of entities (utility, vendor, loyalty programme, customer) and does not model the multi-layered administrative permission structure typical of municipal infrastructure (e.g., city, district, station levels). Addressing the need for hierarchical structures, Ferré-Queralt et al. recently proposed in [28] a hierarchical smart contract architecture for decentralized energy trading. While this work shares our motivation for a multi-level design, this hierarchy is designed to anonymize user identities and prevent profiling in energy markets, rather than to serve as a general-purpose, scalable framework for instantiating and managing the underlying infrastructure components.

Rana et al. in [29] explore blockchain applications for sustainable tourism, demonstrating how smart contracts automate service agreements, but overlook scalability challenges in hierarchical organizations. In healthcare, in [30], Pinto et al. design a blockchain-based system for the traceability of health data, leveraging RBAC to manage the roles of patients and providers. While their approach enforces strict access policies, it relies on a single factory contract for deployment, which limits scalability in multi-institutional ecosystems. Recent advancements in decentralized identity management, such as Stockburger et al. [31], employ self-sovereign identities (SSI) for public transportation systems. This framework eliminates centralized authority but struggles with dynamic role assignment during emergencies, a key requirement for critical infrastructure. Finally, Zhang et al. in [18] apply the abstract factory pattern to healthcare, inspiring similar modular designs in resource distribution networks, but without multirole authentication and authorization access control.

6 Conclusion

This paper takes a step forward and provides more details on the definition of a hierarchical factory pattern for smart contracts enhanced with multirole authentication and authorization. The proposed solution is applied to a real-world water manage-

ment system, allowing for a more detailed analysis of its advantages, limitations, and potential benefits. It demonstrates why the use of a hierarchical structure of smart contracts is necessary in some systems and how a fine-grained multirole authentication and authorization mechanism needs to be integrated at each level of the hierarchy to support secure and effective system operation. The proposed approach has also been evaluated from both performance and security perspectives. For the first aspect, this paper considers scalability and optimization concerns, while for the second aspect, it analyzes the ability of the approach to deal with two important security attacks.

Future work will consist of further experiments with the proposed approach in a company managing the water network of Verona, a city in northern Italy. This experimentation will provide further evidence of the scalability and potential of the approach. Moreover, it is planned to investigate in more detail the scalability and optimization aspects in relation to the storage of a huge amount of sensor data inside the smart contracts. This problem is out of the scope of this paper, and so it has only been mentioned when discussing the possibility of exploiting the Interplanetary File System (IPFS) for storing such data. However, a complete formalization of the problem will be needed in future work, together with a reliable solution.

Author Contributions The authors contribute equally to the work.

Funding Open access funding provided by Università degli Studi di Verona within the CRUI-CARE Agreement. This study was partially carried out within the Interconnected Nord-Est Innovation Ecosystem (iNEST) and the initiative "Innovative PhDs which respond to the companies demand of innovation". It received funding from the European Union Next-GenerationEU (PIANO NAZIONALE DI RIPRESA E RESILIENZA (PNRR) – MISSIONE 4 COMPONENTE 2, INVESTIMENTO 1.5 – D.D. 1058 23/06/2022 ECS00000043, and MISSIONE 4 COMPONENTE 2, INVESTIMENTO 3.3 - DM 352/2022 progetto M4C2 Investimento 3.3). This manuscript reflects only the authors' views and opinions, neither the European Union nor the European Commission can be considered responsible for them.

Data Availability The source code developed for this paper has been made available in the following GitHub repository https://github.com/MuhammadBinSaif/Extension-Hierarchical-Factory-Pattern-in-Smart-Contracts.

Declarations

Conflict of interest The authors have no Conflict of interest to declare that are relevant to the content of this article.

References

1. Mohanta, B.K., Panda, S.S., Jena, D.: An overview of smart contract and use cases in blockchain technology. In: 2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT), pp. 1–4 (2018). https://doi.org/10.1109/ICCCNT.2018.8494045
2. Pinto, R.P., Silva, B.M.C., Inácio, P.R.M.: A system for the promotion of traceability and ownership of health data using blockchain. IEEE Access **10**, 92760–92773 (2022). https://doi.org/10.1109/ACCESS.2022.3203193
3. Rana, R.L., Adamashvili, N., Tricase, C.: The impact of blockchain technology adoption on tourism industry: A systematic literature review. Sustainability (2022). https://doi.org/10.3390/su14127383
4. Xia, W., Chen, X., Song, C.: A framework of blockchain technology in intelligent water management. Front. Environ. Sci. (2022). https://doi.org/10.3389/fenvs.2022.909606
5. Stockburger, L., Kokosioulis, G., Mukkamala, A., Mukkamala, R.R., Avital, M.: Blockchain-enabled decentralized identity management: the case of self-sovereign identity in public transportation. Blockchain: Res. Appl. **2**(2), 100014 (2021). https://doi.org/10.1016/j.bcra.2021.100014
6. Trček, D.: Cultural heritage preservation by using blockchain technologies. Heritage Sci. **10**(1), 6 (2022). https://doi.org/10.1186/s40494-021-00643-9
7. Migliorini, S., Gambini, M., Belussi, A.: A blockchain-based platform for ensuring provenance and traceability of donations for cultural heritage. Blockchain: Res. Appl. (2025). https://doi.org/10.1016/j.bcra.2025.100278
8. Zhou, Q., Huang, H., Zheng, Z., Bian, J.: Solutions to scalability of blockchain: A survey. IEEE Access **8**, 16440–16455 (2020)
9. Saif, M.B., Migliorini, S., Spoto, F.: A survey on data availability in layer 2 blockchain rollups: Open challenges and future improvements. Future Internet (2024). https://doi.org/10.3390/fi16090315
10. Porru, S., Pinna, A., Marchesi, M., Tonelli, R.: Blockchain-oriented software engineering: challenges and new directions. In: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), pp. 169–171 (2017). IEEE
11. Merlec, M.M., In, H.P.: Sc-caac: A smart-contract-based context-aware access control scheme for blockchain-enabled iot systems. IEEE Internet Things J. **11**(11), 19866–19881 (2024)
12. Rajasekar, V., Sondhi, S., Saad, S., Mohammed, S.: Emerging design patterns for blockchain applications. In: ICSOFT, pp. 242–249 (2020)
13. Wu, K., Ma, Y., Huang, G., Liu, X.: A first look at blockchain-based decentralized applications. Softw.: Pract. Exp. **51**(10), 2033–2050 (2021)
14. Saif, M.B., Migliorini, S., Spoto, F.: Blockchain-based multirole authentication and authorization in smart contracts with a hierarchical factory pattern. In: 6th International Conference on Blockchain Computing and Applications (BCCA), pp. 22–29 (2024). https://doi.org/10.1109/BCCA62388.2024.10844391
15. Staderini, M., Palli, C., Bondavalli, A.: Classification of ethereum vulnerabilities and their propagations. In: 2020 Second International Conference on Blockchain Computing and Applications (BCCA), pp. 44–51 (2020). https://doi.org/10.1109/BCCA50787.2020.9274458
16. Kamboj, P., Khare, S., Pal, S.: User authentication using blockchain based smart contract in role-based access control. Peer-to-Peer Netw. Appl. **14**(5), 2961–2976 (2021)
17. Cruz, J.P., Kaji, Y., Yanai, N.: Rbac-sc: Role-based access control using smart contract. IEEE Access **6**, 12240–12251 (2018)
18. Zhang, P., White, J., Schmidt, D.C., Lenz, G.: Applying software patterns to address interoperability in blockchain-based healthcare apps. arXiv preprint arXiv:1706.03700 (2017)
19. Liu, B., Sun, S., Szalachowski, P.: SMACS: Smart contract access control service. In: 50th Annual IEEE/IFIP Inter. Conf. on Dependable Systems and Networks, pp. 221–232 (2020). https://doi.org/10.1109/DSN48063.2020.00039
20. Zhou, Y., Han, M., Liu, L., Wang, Y., Liang, Y., Tian, L.: Improving iot services in smart-home using blockchain smart contract. In: 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), pp. 81–87 (2018). https://doi.org/10.1109/Cybermatics_2018.2018.00047
21. Ding, Y., Sato, H.: Bloccess: enabling fine-grained access control based on blockchain. J. Netw. Syst. Manag. **31**(1), 6 (2023)

22. Feng, L., Lin, J., Qiu, F., Yu, B., JIn, Z., Wang, J., Cheng, J., Yao, S.: Sdac-bbpp: A secure dynamic access control scheme with blockchain-based privacy protection for iiot. IEEE Trans. Netw. Serv. Manag. **21**(3), 3179–3193 (2024)
23. Ma, M., Shi, G., Li, F.: Privacy-oriented blockchain-based distributed key management architecture for hierarchical access control in the iot scenario. IEEE Access **7**, 34045–34059 (2019)
24. Abdi, A.I., Eassa, F.E., Jambi, K., Almarhabi, K., Khemakhem, M., Basuhail, A., Yamin, M.: Hierarchical blockchain-based multi-chaincode access control for securing iot systems. Electronics **11**(5), 711 (2022)
25. Xia, W., Chen, X., Song, C.: A framework of blockchain technology in intelligent water management. Front. Environ. Sci. **10**, 909606 (2022)
26. Biswas, J., Haid, M., Bhalerao, A., Engelhardt, S., Lemke, S.: Wada-water diplomacy automation: using blockchain, ai, and environment iot for water management and climate action. J. Sens. Sensor Syst. **14**(2), 187–196 (2025)
27. Alrammal, M., Abu-Amara, F., Ismail, Z., Nadeem, M.: Blockchain technology for sustainable management of electricity and water consumption. Eng. Proc. **59**(1), 223 (2024)
28. Ferré-Queralt, J., Castellà-Roca, J., Viejo, A.: Blockchain-based hierarchical smart contracts to prevent user profiling in decentralized energy trading systems. Sustain. Energy, Grids Netw., 101762 (2025)
29. Rana, R.L., Adamashvili, N., Tricase, C.: The impact of blockchain technology adoption on tourism industry: a systematic literature review. Sustainability **14**(12), 7383 (2022)
30. Pinto, R.P., Silva, B.M., Inácio, P.R.: A system for the promotion of traceability and ownership of health data using blockchain. IEEE Access **10**, 92760–92773 (2022)
31. Stockburger, L., Kokosioulis, G., Mukkamala, A., Mukkamala, R.R., Avital, M.: Blockchain-enabled decentralized identity management: The case of self-sovereign identity in public transportation. Blockchain: Res. Appl. **2**(2), 100014 (2021)

Muhammad Bin Saif Muhammad Bin Saif is a PhD candidate at the University of Verona. He holds a master's degree in computer applications technology from the Shenyang Aerospace University, China. His research interests span blockchain technology, smart contracts, and agentic AI systems, with a particular focus on decentralized governance, zero-knowledge proofs, and compliance verification for multi-gent LLM architectures

Sara Migliorini Sara Migliorini is an Associate Professor of Computer Science at the University of Verona, Italy. She obtained both her Master's degree in Computer Science in 2007 and her Ph.D. in Computer Science in 2012 from the same institution. Her research interests include information systems, big data systems and analytics, recommender systems, machine learning, and blockchain technology. She has participated in and led research and development activities in several projects in the fields of information systems and blockchain technology

Fausto Spoto Fausto Spoto is associate professor at the University of Verona. He holds a PhD in computer science from the University of Pisa, where he started his research activity in the area of programming languages, semantics and software verification. More recently, he worked in the area of smart contracts, languages for smart contracts and software engineering for blockchain software.

Authors and Affiliations

Muhammad Bin Saif[1] · Sara Migliorini[1] · Fausto Spoto[1]

✉ Muhammad Bin Saif
muhammadbin.saif@univr.it

Sara Migliorini
sara.migliorini@univr.it

Fausto Spoto
fausto.spoto@univr.it

[1] Department of Computer Science, University of Verona, Strada Le Grazie, 15, Verona 37134, Italy