IEEE ICBC'24 1570977337

Blockchain-Based Multirole Authentication and Authorization in Smart Contracts with a Hierarchical Factory Pattern

***Abstract*—The recent spread of smart contract usage in several different application domains has posed additional challenges related to both their scalability, effective management and security. At this regard some design patterns have been proposed to both factorize common parts, preventing their replicated deployment by promoting reusability, and to apply some role-based access control (RBAC) technique during smart contract execution. In particular, the factory pattern has been widely recognized as a common practice to solve the scalability issue, reducing also the increased gas costs related to both smart contract deployment and its subsequent use. However, in real world applications the factory pattern can be considered too limited, since it allows one to instantiate only a single type or family of contracts. From the other side, no solutions have been proposed which tightly integrate a factory pattern with RBAC functionalities. For these reasons, this paper proposes an extension of the factory pattern, called hierarchical factory pattern, which is integrated with a multirole authentication and authorization mechanism specifically tailored to this structure, since it allows a secured and role-specific interaction between the various layers of the hierarchy. The paper ends with some considerations about performance and scalability.**

***Index Terms*—smart contract, design pattern, factory pattern, multi-role authentication and autorization**

I. Introduction

A smart contract is a piece of code deployed on the blockchain to execute predefined actions, enabling automated execution, by ensuring immutability, and transparency of agreements in a trustless environment [1]. Smart contracts greatly contribute to the spread of the blockchain technology in several different application domains, such as healthcare [2], tourism [3], energy and water management [4], identity management [5], cultural heritage preservation [6], and so on. Moreover, the introduction of decentralized finance (DeFi) and Non Fungible Tokens (NFT) has further extended their scope and applicability. At the same time, these diverse applications have raised critical challenges in smart contract development, as any other software project, such as scalability, modularity and security vulnerabilities [7].

In its general meaning, scalability is the ability of the blockchain to accommodate a higher volume of transactions. With reference to smart contracts, it is related to the ability of efficiently handling an increased workload, namely an increased demand towards their functionalities. It mainly refers to run-time aspects and is also highly related to the transaction costs required to invoke a smart contract function. Conversely, modularity is mostly related to the design and development of smart contracts which often have to deal with an increasing complexity and intricate correlations. Programming with modularity in mind means separate program functions into independent pieces of code. This can help increasing not only reusability, but also readability, and in turn prevents vulnerabilities. Moreover, as smart contracts become more complex and interactive, the need for secure, controlled, and authorized access increases. Indeed, it is important to ensure that only authorized entities can interact with a contract to perform specific actions.

With the aim to face all these challenges, in the last years several efforts have been made in literature to start proposing new design patterns tailored for smart contracts development. The significance of design patterns is widely recognized in general purpose programming and, in this context, well consolidated solutions have been defined. However, with reference to the development of smart contracts and decentralized applications (DApps), innovative solutions are necessary to face their specific challenges, related to the decentralized nature of blockchain platforms. In this context, one of the most important design patterns is represented by the *factory pattern* [8] whose main idea is to a have a contract (the factory) that is responsible for the creation of other contracts. In this way, you can create multiple instances of the same contract skeleton by easily keeping track of them and simplifying their managements. Moreover, the factory pattern allows one to reduce the deployment costs, since the child contract is deployed only once when you deploy the factory contract, while subsequent instances are created by the factory without the need to redeploy the bytecode, and it increases smart contract security by reducing the risk of vulnerabilities [9].

In real world applications, the use of the factory pattern can be considered too limited, since it allows one to instantiate only a single type or family of contracts. Therefore, this paper proposes an extension of the factory pattern, called *hierarchical factory pattern*, which introduces a multi-layered structure for the creation and management of objects. More specifically, it allows for the creation of objects that, in turn, can act as factories for other objects. This results in a tree-like structure of object creations, where each node can potentially be a factory for the next level. However, even though the hierarchical factory pattern provides a structured and scalable approach to smart contract deployment, it also poses security and access control challenges. Indeed, since factories and contracts are linked by multiple layers, it is essential to ensure

that each interaction is secured and role-specific. Therefore, we enrich this hierarchical design pattern with a Multirole Authentication and Authorization (MAC) mechanism specifically tailored to this structure. The proposed system not only defines roles at each level but also 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 solution not only provides an effective solution for real-world applications, but it also efficiently reduces the bytecode size and data storage of the smart contract, contributing also to the scalability issue.

The main contributions of this paper can be summarized as follows: (i) we propose a novel hierarchical factory design pattern for smart contracts, (ii) we enrich it with a blockchain based multi-role authentication and authorization access control, and (iii) we demonstrate how the proposed hierarchical factory design pattern efficiently manages the bytecode size and contributes to the reduction of the data storage of smart contracts. To the best of our knowledge, there are no solutions that combine role-based access control with a factory pattern, or even better with a hierarchical factory pattern.

The remainder of the paper is organized as follows: Sect. II discusses some previous work in the field of design patterns as well as authentication and authorization in smart contract development. Sect. III illustrates the proposed hierarchical factory design pattern enriched with multi-role authentication and authorization. Sect. IV evaluates the implementation of the proposed solution and discusses some obtained results. Finally, Sect. V concludes the work.

Motivating Example

Let us consider a company divided into several departments, each of which is responsible for a particular aspect of the organization activities (e.g., purchasing, sales, marketing, production, technology and so on), as depicted in Fig. 1. In this case the main company manages the rights of all its subsidiaries, from departments to single offices, by granting or revoking their permissions. At the same time, each department is responsible for the activities under its control which can be operationally carried out through several offices, eventually geographically distributed in different sites. All these levels can benefit from the use of one or more factory patterns, because the company should deploy similar smart contracts for each of its departments, and at the same time each department could have the need to coordinate several offices which can in turn deploy some similar contracts for their operating activities.

However, each of these factories cannot be considered an island alone without any connection with the other ones, since they could have some functional dependencies with the one used by the parent or sibling levels in the hierarchy. In particular, one of these dependencies regards the different authorizations and roles that govern the deployment and management of smart contracts, which could need to be modified or revoked by the corresponding parent level. In order to address this typical real-world situation, it is necessary to extend the classical factory pattern as proposed in this paper, namely through the definition of a hierarchical factory pattern supporting multirole authentication and authorization.

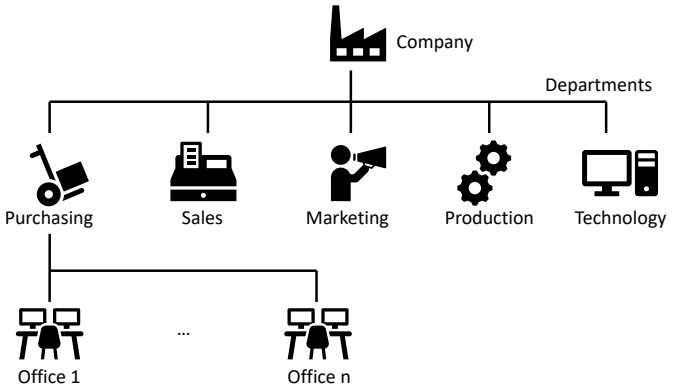

Fig. 1. Motivating example: typical structure of a company.

II. Related Work

This section summarizes some work related to the definition of design patters for smart contract development and to the development of solutions for managing authentication and authorization in this field.

Smart contract design patterns – In [8] Rajasekar et al. explore various design patterns developed for blockchain applications. Among these, the *factory contract* pattern is the most important one for our proposal. It stores on blockchain a template contract, called factory, from which it is possible to instantiate similar child contracts, with the aim to improve modularity and consistency. Other proposed patters are: *checks-effects-interactions*, *oracle*, *off-chain data storage*, *state channel*, *contract registry*, and the *emergency stop* pattern. All these patterns demonstrate the evolving best practices in designing blockchain applications, emphasizing the importance of security, efficiency, and adaptability. Relatively to the factory pattern, also Wu et al. in [9] emphasize the significance in smart contract design. In particular, they demonstrate its usefulness for applications that require consistent behaviours to be replicated across multiple smart contracts, such as gambling Decentralized Applications (DApps).

With reference to the definition of design patterns for specific application domains, in [10] 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 create user accounts. It allows the instantiation of objects without specifying exact classes and simplifying the creation of new related contracts for departments and subdivisions.

Compared to the traditional factory pattern already present in literature, the hierarchical factory pattern proposed in this paper offers an improved structured and scalable approach.

Indeed, while traditional factory or abstract factory patterns can deploy only similar instances of smart contracts, the hierarchical factory pattern can deploy both diverse factories and contracts. This makes it more versatile and adaptable for diverse applications, as well as more compatible with complex hierarchical organizational structures.

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. Therefore, several design patterns have been proposed to address the access control challenge in smart contract. The *role-based access control* (RBAC) pattern is one of the access control design patterns that assigns the roles to different users and grant permissions to the roles. In [11], 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. Another access control design pattern is the *token-based access control* (TBAC), which employs tokens to represent user access rights. The TBAC enables users to transfer or delegate access rights by transferring tokens. In [12], Liu et al. introduce the SMACS framework, which enables low-cost implementation of updatable and sophisticated access control rules (ACRs) for smart contracts.

In [13], Zhou et al. proposes a smart contract-based ownership access control framework for the Internet of Things (IoTs) in smart home systems. The proposed framework employs the ownership design pattern to define a hierarchy of owners, such as device, home, and system owner.

To the best of our knowledge, there are no solutions that combine role-based access control with a factory pattern. This paper proposes a hierarchical factory pattern for managing and deploying smart contracts, along with a multi-role authentication and authorization mechanism for access control. This novel approach potentially offers more effective security and access control solutions for smart contracts.

III. Proposed Solution

The proposed hierarchical factory pattern introduces a multi-layered structure for the creation and management of objects. Unlike traditional factory patterns that focus on instantiating a single type or family of objects, the hierarchical factory pattern allows for the creation of objects that, in turn, can act as factories for other objects. This results in a tree-like structure of object creation, where each node can potentially be a factory for the next level. Moreover, the proposed hierarchical factory pattern is enhanced and integrated with an authentication and authorization mechanism which is able to properly treat roles and permissions along the tree.

A. Hierarchical Factory Pattern

The structure of the hierarchical factory pattern supports the recursive creation of sub-factories, so that each level consists of several distinct factories and/or distinct smart contracts. Fig. 2 illustrates an example of such hierarchy, where the yellow boxes represent the mandatory components of the tree, while the 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. For instance, in the figure, one additional intermediary level of client factories has been added, besides to the mandatory one, with the following convention: the first number in the subscript represents the level number, while the second number denotes the sibling inside the same level. This number of levels can accommodate the situation introduced in the example of Sect. I: indeed, while the Super Factory is associated with the company owner, there is a level of client factories managed by the departments and a subsequent one under the control of each single office.

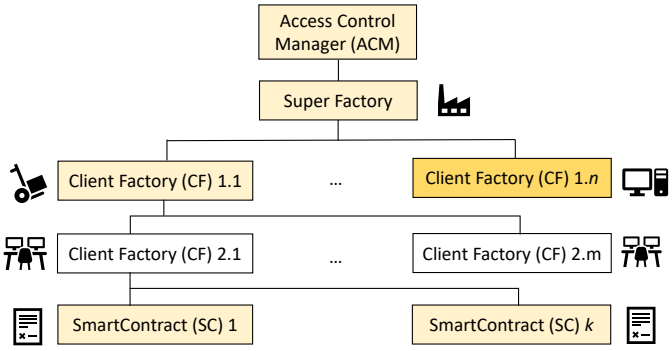

Fig. 2. Hierarchical Factory Pattern

The root level of the tree hierarchy is represented by the *Access Control Manager* (ACM) smart contract which plays a crucial role in the management of permissions, roles, and access levels for the various children. It acts as the manager to ensure that only authorized entities interact with factories and smart contracts. The second level of hierarchy is represented by the *Super Factory* (SF). This entity serves as a central hub to deploy client-specific sub-factories and handles the connection with the ACM to acquire permissions and roles for these sub-factories. The ACM and the SF can be considered a unique component in the hierarchy, since they manage together the hierarchical generation of factories and smart contracts enhanced with authentication and authorization capabilities. They have been split into two smart contracts in order to both increase modularity and also to reduce their size.

Alg. 1 reports a sketch of the ACM structure. 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). Clearly, depending on the specific requirements of the use-case application, the number and kinds of both roles and contract types can be properly changed in a straightforward manner. In this case, in order not to clutter the notation, we have defined only the three mandatory roles and the four mandatory layers. The ADMIN role has access to any element in the hierarchy in any way

Algorithm 1: AccessControlManager

```
1  Contract AccessControlManager :
2      public constant ADMIN;
3      public constant CLIENT;
4      public constant USER;
5      enum ContractType {acm, superFactory,
         clientFactory, smartContract};
6      struct ContractData :
7          address _contractAddress;
8          address _ownerAddress;
9          address _parentContractAddress;
10         ContractType _contractType;
11     address private _adminOwner;
12     mapping private _accessControl;
13     constructor :
14         createNewContract(ADMIN, msg.sender,
             address(this), msg.sender, address(this), acm);
15         createNewContract(ADMIN, msg.sender,
             address(new SuperFactory(address(this))),
             msg.sender, address(this), superFactory);
16         adminOwner = msg.sender;
17     function createNewContract(role, user, contrAddr,
         ownerAddr, parAddr, contrTy) :
18         ContractData c = new ContractData
             ({contrAddr, ownerAddr, parAddr, contrTy});
19         accessControl[role][user].push(c);
20     function authnz(role, data, sign) returns address :
21         // Authentication and authorization
22     function addContractOnBehalfOf(contractAddr,
         owner, parent, data, sign, type) :
23         address signer = authnz(ADMIN, data, sign);
24         createNewContract(CLIENT, owner,
             contractAddr, owner, parent, type);
25         createNewContract(ADMIN, signer
             contractAddr, owner, parent, type);
26     function addContract(contractAddr, parent, data,
         sign, type) :
27         address signer = authnz(CLIENT, data, sign);
28         createNewContract(CLIENT, signer,
             contractAddr, owner, parent, type);
29         createNewContract(ADMIN, _adminOwner,
             contractAddr, owner, parent, type);
```

(creation, modification, deletion or invocation); the CLIENT role has access to any element inside its sub-tree only for the creation and invocation operations; finally, the USER role can only be used for the invocation of functionalities provided by the smart contracts in the leaf level only.

Given that, the ACM contract maintains two main properties: a reference *_adminOwner* to the owner of this contract, which acts as 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. More specifically, each contract is represented through a data structure *ContractData* (lines 6-10) which maintains the useful information, such as 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 to. Other additional information can be added depending on the specific application requirements.

The ACM constructor (lines 13-16) is responsible for three important things: (i) it registers the current ACM contract inside the *_accessControl* map and associates it with the ADMIN role for the `msg.sender` address. (ii) It creates a new instance of the SuperFactory and registers also it inside the *_accessControl* map with the same role and owner. (iii) Finally, it stores the address of the transaction sender inside the *_adminOwner* variable. The updates of the *_accessControl* map are performed with function *createNewContract*() (lines 17-19), which essentially creates a new instance of the data structure and stores it inside the map.

Function *authnz*() will be described in more detail in Sect. III-B. It essentially authenticates the user which has signed *data* with the signature *sign* and checks if this user has the required role *role*. In case of success, it returns the address of *user*.

Finally, the last two functions are responsible for the assignment of role permissions to contracts deployed through the factories inside the hierarchy. In particular, function *addContractOnBehalfOf*() could be invoked by the *Super Factory* to deploy an element anywhere in the hierarchy (both subfactories and smart contracts) on behalf of other factories. Indeed, after authenticating the signer and checking that it has an ADMIN role, the function registers the current contract inside the *_accessControl* map, by assigning administrative permissions to the signer (i.e., the SuperFactory contract) and CLIENT role to the specified contract owner. Conversely, function *addContract*() could be invoked by a Client Factory during the deployment of one of its child contracts or subfactories. In this case, the signer is checked against the CLIENT 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.

The skeleton of the SF contract is illustrated in Alg. 2. It maintains a reference to the connected ACM contract and it defines a modifier *checkAdmin()* which invokes the ACM method for checking the ADMIN role for the message sender (lines 5-7). This modifier is used to ensure that the function *createClientContract*() could be called only by an ADMIN user. This function (lines 8-16) could in turn create any kind of factory or smart contract in the hierarchy, on the basis of the type passed as parameter (see the enumeration in Alg. 1, line 5), receiving as a result the contract address. Finally, the contract is registered in the ACM map on behalf of the corresponding factory inside the hierarchy (line 16). Referring to the situation in Fig. 2, through the SF, the company owner

Algorithm 2: Super Factory Smart Contract

```
1  Contract SuperFactory :
2      ACM private _acm;
3      constructor (address acmAddress) :
4          _acm = ACM(acmAddress);
5      modifier checkAdmin() :
6          require(_acm.hasRole(ADMIN, msg.sender));
7          _;
8      function createClientContract(ownerAddr,
        parentAddr, data, sign, type) checkAdmin() :
9          address cAddr;
10         if type == 1 then
11             cAddr = address(new CF_1(address(_acm)));
12         else if ... then
13             ... ;
14         else
15             cAddr = address(new CF_n(address(_acm)));
16         _acm.addContractOnBehalfOf(cAddr,
            ownerAddr, parentAddr, data, sign, type);
```

could create an instance of any client factory or smart contract in the tree on behalf of any department or office in the hierarchy.

The subsequent levels in the hierarchy could include one or more Client Factories (CFs) which can be domain-specific factories able to deploy other factories or smart contracts, based on the organization requirements. For instance, in Fig. 2 there is a first level of CFs associated to the company departments, each of which could deploy a set of factories for its offices representing a second level of CF which is responsible for deploying the final smart contract instances. In other words, each low level CF together with its child leaf nodes act as a traditional factory pattern.

Algorithm 3: Client Factory

```
1  Contract ClientFactory_i :
2      ACM private acm;
3      constructor (address acmAddress) :
4          acm = ACM(_acmAddress);
5      modifier checkClient() :
6          require(acm.hasRole(CLIENT, msg.sender));
7          _;
8      function createContract(parentAddr, data, sign)
        checkClient() :
9          acm.addContract(
10             address(new ⟨SC_i⟩(address(acm))),
11             parentAddr, data, sign, ⟨type⟩);
12     function f(...) checkClient() :
13         ...
```

The structure of a CF is reported in Alg. 3. Similarly to the SF, a CF maintains a reference to the ACM, but it defines a modifier that restricts the access only to users with CLIENT role. This role is necessary to call the *createContract*() method which creates a new child contract (or factory in the intermediary levels) and registers it in the ACM through the *addContract*() function described in Alg. 1. At each level of the hierarchy, the body of the *createContract*() function could differ for the created contract (generally denoted as $\langle \mathbf{SC_i} \rangle$ in line 11) and the passed enumerated contract type (generally denoted as $\langle \mathbf{type} \rangle$ in line 11).

Algorithm 4: Smart Contract

```
1  Contract SmartContract_i :
2      ACM private manager;
3      constructor (address acmAddress) :
4          acm = ACM(acmAddress);
5      modifier checkRole() :
6          require(acm.hasRole(CLIENT, msg.sender) ||
            acm.hasRole(ADMIN, msg.sender) ||
            acm.hasRole(USER, msg.sender));
7          _;
8      function g(...) checkRole() :
9          ...;
```

Let us notice that each CF could contain additional specific functions, for instance $f()$ in lines 12-13, which provides to the factory additional capabilities, besides to the creation of new contracts. This makes our factories much more flexible than traditional smart contract factories proposed in literature, that can only create new instances of the same smart contract type.

To conclude, a prototypical smart contract belonging to the leaf level is presented in Alg. 4. In this case, there is a modifier that checks if the message sender has a USER role or any upper level role. Each function (such as g in lines 8-9) is then guarded by such modifier.

The proposed design pattern is scalable, since deploying a new contract type requires adding a new factory in the appropriate parent factory contract. This ensures that the system can scale without altering the existing smart contracts. Each contract in the proposed system manages a specific set of tasks. For instance, ACM manages access controls, SF handles the creation of certain types of smart contracts, and GF further deploys client-specific contracts. This separation ensures that smart contracts can be modular and that they can be managed, tested, and updated separately: the hierarchical factory pattern proposed in this research presents a flexible, managable, scalable and adaptive design.

B. Multirole Authentication and Authorization

The emergence of blockchain technology has brought a paradigm shift in the digital security domain, from traditional

centralized authentication and authorization systems to decentralized mechanisms. Indeed, traditional Role Based Access Control (RBAC) systems employ centralized authorities to verify the user identity, exposing the system to a single point of failure, risks of data breaches and unauthorized access. The fundamental characteristics of blockchain, such as immutability, transparency, and resistance to tampering, provide a solid foundation for multi-role authentication and authorization processes. In particular, blockchain-based systems could distribute the authentication information across a network of nodes, making it difficult, for malicious actors, to undermine the integrity of the system.

This paper employs Ethereum smart contracts and the Metamask wallet for implementing blockchain-based multi-role authentication and authorization. This approach requires users to actively sign a message with their private key in Metamask, generating a cryptographic proofs of their identity. Using message signature for retrieving public addresses enhances the security and privacy for user authentication, since it provides a more significant indication of user intent and ownership, rather than the address retrieved from the `msg.sender` variable. The signed message is then verified with the *verify*() function illustrated in Alg. 5 (lines 2-3). This function takes two inputs: a hashed message *data* and a signature *sign*, from which it is able to retrieve the user address through the *recover*() function of the Elliptic Curve Digital Signature Algorithm (ESDSA) library.

The *verify*() function is used by *authnz*(), which performs both authentication and authorization (lines 7-10). Indeed, after retrieving the signer address in a safe way, it checks if the signer has the required role through the *hasRole*() function (lines 4-6), by accessing the content of the `accessControl` map. As deeply discussed in the previous section, instead of relying on a flat permission model, our solution uses a hierarchical access control mechanism that maps user addresses and roles to specific contract addresses. This defines clear access boundaries for roles at each level in the hierarchy. When a user attempts to access a contract, the system not only verifies the user role, but also if the user is authorised to access the specific contract within the hierarchy.

Several roles can be defined inside the ACM, based on the specific application requirements (see lines 2-4 in Alg. 1). Anyway, the definition of the ADMIN role is mandatory. It is automatically assigned to the deployer of the ACM smart contract and this role assignment is a foundational security measure to ensure that the primary administrative control remains with the entity responsible for the contract creation. As a result, the contract deployer is responsible to manage the contract access in the hierarchy and enforce role-based permissions.

The ACM smart contract also includes a function to revoke roles from users. This function is called *revokeRole*() (lines 11-14) and could be invoked only by the ADMIN to remove any role different from itself. This limitation is necessary in order to not lose the control over the hierarchy. It accepts two parameters: the first one is a role r and the second one is a user address u to which we want to revoke the role privilege r towards the previously assigned smart contracts. The function initially verifies that the user address is a valid one (i.e., different from zero) and is not the administrator address (lines 12-13), then it removes from the roles r the user u in the access control mapping. This function provides an additional layer of control to the ADMIN, allowing the effective role management and the maintenance of the integrity of ACM in a dynamic environment where user permissions may need to be updated frequently.

Finally, the ACM incorporates a function to transfer the ownership of the ACM, namely the transfer of the ADMIN role. The *transferOwnership*() function can only be invoked by the current ADMIN to transfer the ownership to a new ADMIN (lines 15-23). The function initially verifies that the new ADMIN address is not a zero address, to ensure that ownership is transferred to a valid address, and that the new address is different from the previous one. Then, all the contracts associated with the old ADMIN role are transferred to the new one in the `accessControl` mapping.

The *transferOwnership*() function is critical for ensuring the security of ACM, especially if the private key of the ADMIN

Algorithm 5: AccessControlManager

```
1  Contract AccessControlManager :
2      function verify(data, sign) returns address :
3          return data.toEthSignMessageHash().
             recover(sign);
4      function hasRole(role, user) returns bool :
5          ContractData[][] cs = accessControl[role][user];
6          return cs.length > 0;
7      function authnz(role, data, sign) returns address :
8          address signer = verify(data, sign);
9          require(hasRole(role, signer);
10         return signer;
11     function revokeRole(role, user) onlyAdmin() :
12         require(user != address(0));
13         require(user != adminOwner);
14         delete accesscontrol[role][user];
15     function transferOwnership() onlyAdmin :
16         require(newAdmin != address(0), "Invalid
             address");
17         require(newAdmin != AdminOwner, "New
             admin must be different");
18         ContractData[] storage oldAdminConts =
             accessControl[ADMIN][adminOwner];
19         ContractData[] storage newAdminConts =
             accessControl[ADMIN][newAdmin];
20         for i = 0; i < oldAdminConts.length; i++ do
21             newAdminConts.push(oldAdminConts[i]);
22         delete accessControl[ADMIN][adminOwner];
23         adminOwner = newAdmin;
```

has been compromised or exposed. 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.

C. Optimization of smart contracts

The optimization of smart contracts in terms of both gas cost for the deployment and bytecode size is an important and pivotal task during development. 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 has a great impact on G_i and smart contracts have to be properly designed in order to reduce this amount.

The use of hierarchical factory pattern is certainly one of the methods for reducing the bytecode size and increasing scalability, since some common parts could be isolated inside the factory and called by all the other factories or smart contracts in the hierarchy. Beside this, another useful approach is the use of libraries for isolating some reusable behaviours. In particular, the SF in Alg. 2 could exploit methods contained in some libraries to create the various sub-factories inside the *createClientContract*() function. Finally, in smart contract optimization, a clever ordering of state variables could have a great impact in gas cost reduction, since the EVM saves state variables in storage slot by slot and each slot has a space occupation of 32 bytes. Let us consider the *ContractData* structure presented in Alg. 1, and extended for our application purposes as in Alg. 6. By using the proposed organization of variables, the structure occupies a total of 160 bytes, since each address occupies 20 bytes, but its real consumption is extended to 32 bytes, each string occupies an entire slot (32 bytes), and the enum value, which theoretically occupies 1 byte, needs an entire slot in this case. Conversely, if we put the *_contractType* variable between the addresses and the bytes32 variables, the structure requires now $32 \times 4 = 128$ bytes to be stored, since it can be placed inside the same storage slot of the last address.

Algorithm 6: Contract Data structure

```
1 struct ContractData :
2     address _contractAddress;
3     address _ownerAddress;
4     address _parentContractAddress;
5     byte32 _ownerName;
6     ContractType _contractType;
```

IV. Implementation and Performance Analysis

In the experimental setup of our study, we exploit the Ethereum Sepolia test network to test the proposed Hierarchical Factory Pattern in a realistic way but without spending real Ethers. We developed smart contracts in the Solidity language, version 0.8.20, in the Hardhat deployment environment. We employed the Metamask wallet for blockchain interaction, smart contract deployment and transactions. Finally, the application DApp layer has been developed by using the React library and Ether.js. Tab. I summarizes all the tools and technologies used in our experiments, while the source code has been published on a GitHub repository[1].

TABLE I
Tools and Technologies Used in the Experimental Setup

Technology	Version/Network
Ethereum Test Network	Sepolia
Solidity	0.8.20
Contract Deployment	Hardhat
Blockchain Interaction	MetaMask Wallet
DApp Development	Ether.js
Frontend Framework	React TypeScript

We evaluated the performance of the proposed solution with a particular focus on two critical aspects: the gas cost required for contract execution and deployment and the bytecode size of the deployed contracts. These two metrics are essential for determining the efficiency and resource allocation of the proposed pattern in an Ethereum network. In particular, bytecode size measures the complexity of the contract and the amount of code processed by the EVM, while the gas consumption represents the computational resources required for the deployment and the execution of contracts.

For better studying the effect of the hierarchical structure, in our evaluation we assume to implement three intermediary levels of Client Factories, denoted as $\text{CF}_{1,*}$, $\text{CF}_{2,*}$ and $\text{CF}_{3,*}$, respectively. Tab. II illustrates the gas consumed for the deployment and the bytecode size of each factory in the hierarchy, such as the Access Control Manager, the Super Factory, and the three levels of Client Factories. We can observe that the bytecode size of all proposed smart contracts is below the EVM limit, which has been fixed to 24Kbytes to mitigate security attacks, and the size of each child client factory decreases as the depth in the hierarchy increases.

TABLE II
Bytecode Size and Gas Cost of Smart Contract

Smart Contract	Bytecode size (bytes)	Gas used (gwei)
ACM w/o SF Deployment	5,883	1,392,499
Super Factory	1,532	376,733
AM with SF Deployment	7,566	1,906,434
$\text{CF}_{1,*}$	7,460	1,650,649
$\text{CF}_{2,*}$	5,795	1,291,439
$\text{CF}_{3,*}$	4,130	932,240

Fig. 3 compares the gas consumption (in gwei) of the proposed solution with the one of a traditional factory pattern as the number of intermediary client level increases. The results show that, with only a single intermediary, a traditional factory

[1]https://anonymous.4open.science/r/Hierarchical-Factory-Pattern-in-Smart-Contracts-C258

pattern performs better that the proposed solution. However, when the number of levels increases, the benefits of the proposed approach becomes evident. Indeed, to manage this case, the traditional factory pattern requires the deployment of several independent factories, with duplicated functionalities. Similarly, Fig. 4 shows a comparison between the bytecode size of the proposed approach and the one of a traditional factory pattern. According to the previous results, the proposed solution generates an overall bytecode size which is smaller than that of the traditional solution. These results indicate that the proposed solution is more efficient, cost-effective and scalable.

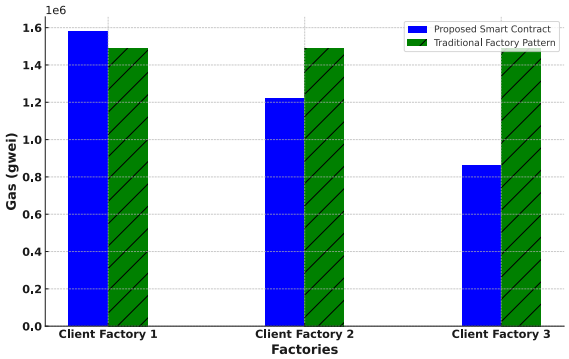

Fig. 3. Gas cost comparison of Proposed vs Traditional Factory Pattern

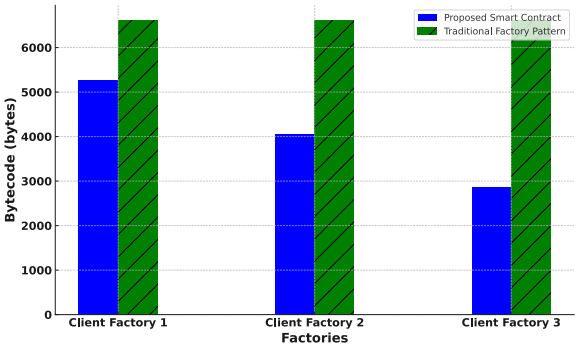

Fig. 4. Bytecode comparison of Proposed vs Traditional Factory Pattern

Conversely, Fig. 5 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 [14] and [11]. In this case, two kinds of operations have been considered: the addition of a new user and the removal of an existing one. For the first operation, different amounts of informative data are passed to the method. The results show that, even if the gas cost increases with the amount of data processed by the *AddUser* method, the proposed approach outperforms both baselines. Also for the *removeUser* method, the obtained results confirm the good behaviour of the approach.

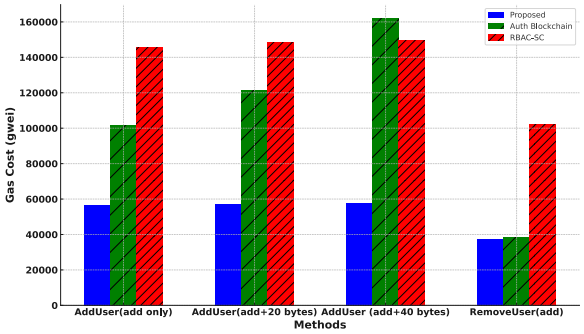

Fig. 5. Gas cost comparison of Proposed vs [14] and [11]

V. Conclusion

This paper proposes a hierarchical factory pattern tightly coupled with an RBAC mechanism for managing authentication and authorization. This design pattern allows a more flexible and scalable approach for the deployment of smart contracts in several different application domains and also in presence of complex organizational structures. Moreover, the strict integration of a multi-role authentication and authorization mechanism allows a finer and more precise management of roles and permissions inside the entire hierarchy. Some experiments performed also with respect to the optimization issues confirm the ability of the pattern to improve scalability and reduce the overall deployment gas costs.

References

[1] Bhabendu Kumar Mohanta, Soumyashree S Panda, and Debasish Jena. An overview of smart contract and use cases in blockchain technology. In *2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, pages 1–4, 2018.

[2] Rui P. Pinto, Bruno M. C. Silva, and Pedro R. M. Inácio. A system for the promotion of traceability and ownership of health data using blockchain. *IEEE Access*, 10:92760–92773, 2022.

[3] Roberto Leonardo Rana, Nino Adamashvili, and Caterina Tricase. The impact of blockchain technology adoption on tourism industry: A systematic literature review. *Sustainability*, 14(12), 2022.

[4] Wenjun Xia, Xiaohong Chen, and Chao Song. A framework of blockchain technology in intelligent water management. *Frontiers in Environmental Science*, 10, 2022.

[5] Lukas Stockburger, Georgios Kokosioulis, Alivelu Mukkamala, Raghava Rao Mukkamala, and Michel Avital. Blockchain-enabled decentralized identity management: The case of self-sovereign identity in public transportation. *Blockchain: Research and Applications*, 2(2):100014, 2021.

[6] Denis Trček. Cultural heritage preservation by using blockchain technologies. *Heritage Science*, 10(1):6, 2022.

[7] Carl R Worley and Anthony Skjellum. Opportunities, challenges, and future extensions for smart-contract design patterns. In *Business Information Systems Workshops: BIS 2018 International Workshops*, pages 264–276, 2019.

[8] Vijay Rajasekar, Shiv Sondhi, Sherif Saad, and Shady Mohammed. Emerging design patterns for blockchain applications. In *ICSOFT*, pages 242–249, 2020.

[9] Kaidong Wu, Yun Ma, Gang Huang, and Xuanzhe Liu. A first look at blockchain-based decentralized applications. *Software: Practice and Experience*, 51(10):2033–2050, 2021.

[10] Peng Zhang, Jules White, Douglas C Schmidt, and Gunther Lenz. Applying software patterns to address interoperability in blockchain-based healthcare apps. *arXiv preprint arXiv:1706.03700*, 2017.

[11] Jason Paul Cruz, Yuichi Kaji, and Naoto Yanai. Rbac-sc: Role-based access control using smart contract. *Ieee Access*, 6:12240–12251, 2018.

[12] Bowen Liu, Siwei Sun, and Pawel Szalachowski. SMACS: Smart Contract Access Control Service. In *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 221–232, 2020.

[13] Yiyun Zhou, Meng Han, Liyuan Liu, Yan Wang, Yi Liang, and Ling Tian. 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)*, pages 81–87, 2018.

[14] Priyanka Kamboj, Shivang Khare, and Sujata Pal. User authentication using blockchain based smart contract in role-based access control. *Peer-to-Peer Networking and Applications*, 14(5):2961–2976, 2021.