Smart Contract-based Access Control through Off-chain Signature and On-chain Evaluation

Jialu Hao, Cheng Huang, *Member, IEEE,* Wenjuan Tang, *Member, IEEE,* Yang Zhang and Shuai Yuan

***Abstract*—Access control is essential in computer security systems to regulate the access to critical or valuable resources. Conventional access control models mainly rely on a centralized and trusted server to mediate each attempted access from client to resources, which face serious challenges of single point of failure and lack of transparency. In this brief, we propose a smart contract-based access control framework, which enables the owner to achieve resource access control in a reliable, auditable and scalable way. An access control contract is deployed on blockchain to manage attribute-based access policies of resources flexibly and make access decisions for clients credibly. A set of attributes is distributed to the clients through off-chain signatures signed by the owner to determine their privileges, without consuming the expensive on-chain storage space. Finally, we implement an experimental prototype on Ethereum test network and perform extensive experimental and theoretical analysis to evaluate its scalability and efficiency.**

***Index Terms*—Access control, Blockchain, Smart contract, Ethereum, System security**

I. Introduction

Access control is a security technique that regulates who or what can view or use resources (*e.g.*, data, services, computational units and storage space) in a computing system. It is a fundamental concept in security that minimizes risk to the business or organization. Traditional access control models mainly rely on a centralized and trusted server to mediate each attempted access from client to resources in the system [1]. Specifically, in the access control list (ACL) based model, each object (resource) is associated with an access control list that saves the users and their access rights for the objects. In the role-based access control model (RBAC) [2], permissions are assigned to roles within the organization, and each user belongs to a role. In the capability-based access control model (CapBAC) [3], the user is empowered with capabilities, and granted with a communicable, unforgeable token from authority, to demonstrate his access rights. In the attribute-based access control model (ABAC) [4], each entity is described with a set of attributes, and special policies combining attributes of user, resource and environment are defined to regulate each access.

Although the effectiveness and efficiency of access control can be guaranteed in the above centralized models, there exists the issue of single point of failure [5]. That is, once the centralized server is compromised or controlled maliciously, owners of the system resources would suffer huge losses [6]. In addition, considering that the access control procedure is primarily enforced on the server side and lack of transparency, clients may also have the demand of confirming the access decisions.

With the advantages of decentralization, transparency, immutability and consensus, the blockchain technology has attracted considerable attentions from researchers focusing on trustworthy and auditable access control [7], [8]. Ouaddah et al. [9] first proposed FairAccess as a fully decentralized pseudonymous and privacy preserving authorization management framework that enables users to own and control their data. They utilized the blockchain into a decentralized access control manager and introduced new types of Bitcoin transactions that are used to grant, get, delegate, and revoke access. Maesa et al. [10] proposed a new approach based on blockchain technology to publish the policies expressing the right to access a resource and to allow the distributed transfer of such right among users. In their proposed scheme, the policies and the rights exchanges are publicly visible on the blockchain, consequently any user can know at any time the policy paired with a resource and the subjects who currently have the rights to access the resource. However, the above preliminary works mainly abstract blockchain as distributed and non-tampering storage to store both resource access rules and client information, and only limited computing capability of locking scripts has been exploited.

As a collection of code (functions) and data (state) that resides at a specific address on blockchain, smart contracts enable users to define rules like a regular contract and automatically enforce them via the code [11], which point a new direction for achieving more effective and reliable access control. Cruz et al. [12] proposed a role-based access control scheme using smart contract (RBAC-SC). A smart contract is deployed to maintain the roles assigned to each user, such that any service provider can read and verify the role of users when providing services. Xu et al. [13] proposed a blockchain-enabled decentralized capability-based access control scheme (BlendCAC) for the security of IoTs. A smart contract is deployed on blockchain to store and manage the capability tokens (*i.e.*, special data structures that maintain the allowed actions of a client on a certain resource), which will be used by the local service provider to make access decisions. To provide more fine-grained access control and more flexible token management, Nakamura et al. [14] improved [13] by defining capability tokens in units of actions, *i.e.*, dividing

J. Hao, Y. Zhang and S. Yuan are with the Xi'an Satellite Control Center, Xi'an, China, e-mail: haojialu.xscc@outlook.com.

C. Huang is with the Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, ON N2L 3G1, Canada.

W. Tang is with the College of Computer Science and Electronic Engineering, Hunan University, Changsha, China.

This work was supported by a grant from the National Natural Science Foundation of China (Grant No. 61801489).

a conventional capability token containing multiple actions into multiple ones with each being associated with a certain action. Note that, in these schemes, the main purpose of smart contracts is still to manage the data records, which will be read by the owner or service provider for making access decisions off-chain.

To further eliminate unreliability caused by off-chain factors and exploit the computing capability of blockchain, Zhang et al. [15] proposed an ACL-based access control framework using Ethereum smart contracts, where one smart contract is deployed for each subject-object pair to store the ACL and implement the related access control. When a subject wants to access an object, it sends a transaction including the required access information to the corresponding smart contract. Once the smart contract is executed, the access control results will be returned to both the subject and object. Nevertheless, the ACL needs to be specified for each client in the system, such that the on-chain storage cost will increase linearly with the number of clients. Therefore, the scalability and flexibility of their proposed framework are seriously limited, especially in a large-scale IoT system. Different from [15], Yutaka et al. [16] combined ABAC model with smart contracts, which consists of one Policy Management Contract (PMC), one Subject Attribute Management Contract (SAMC), one Object Attribute Management Contract (OAMC) and one Access Control Contract (ACC). The PMC, SAMC and OAMC are responsible for storing and managing the ABAC policies, the attributes of subjects (i.e., entities accessing resources) and the attributes of objects (i.e., resources being accessed), respectively. When receiving access requests, the ACC retrieves the subject attributes and object attributes as well as the corresponding policy from the SAMC, OAMC and PMC, to perform the access control and output the access dicisions. Considering that the attributes of each client also need to be stored on blockchain in [16], it suffers from the scalability issue similar to [15]. Table I gives a brief comparison of the existing smart contract-based access control schemes.

TABLE I
COMPARISON OF SMART CONTRACT-BASED ACCESS CONTROL

Schemes	Basic model	Decision maker	Blockchain platform
[12]	RBAC	Service provider	Ethereum(testnet)
[13]	CapBAC	Service provider	Ethereum(private)
[14]	CapBAC	Owner	Ethereum(private)
[15]	ACL	Smart contract	Ethereum(private)
[16]	ABAC	Smart contract	Ethereum(private)
Ours	ABAC	Smart contract	Ethereum(Ropsten)

In this brief, taking advantage of attribute-based access control and blockchain, we propose a scalable and reliable smart contract-based access control framework with the following three contributions:

- We utilize a set of attributes but not a unique identity to describe the clients flexibly and distribute attribute tokens based on the idea of off-chain signature delivery. Consequently, the burden of on-chain identity storage and management is significantly diminished, and the scalability is effectively improved. Meanwhile, the reliability of the off-chain attribute information is guaranteed through efficient on-chain signature verification.
- We design an access control contract to manage the expressive attribute-based access policies of resources and enforce access evaluation procedure on blockchain. The access decision could be monitored by the requester and verified by any party. In addition, the on-chain resource access policies could be updated on demand, and client revocation is achieved only by maintaining an incremental nonce for each client in the contract state.
- We implement a prototype of the proposed framework on the Ethereum test network (Ropsten) and perform extensive experimental and theoretical analysis to demonstrate its scalability and efficiency.

II. THE PROPOSED FRAMEWORK

A. System Model

As shown in Fig. 1, the proposed smart contract-based access control framework includes four parts: owner, client, blockchain and resource pool. The owner is responsible for managing the resources in the pool and sharing them with the authorized clients. A smart contract is deployed on blockchain by the owner to store attribute-based access policies of the resources and make access decisions for the clients. Each client is assigned with a set of attributes by the owner, which would be used by the smart contract to evaluate whether he/she is authorized to access the specified resource. If the attributes of the client satisfy the access policy related to the target resource, the smart contract will trigger an "allow" event. Once catching this event, the resource pool is assumed to faithfully grant the corresponding rights to the client. Note that, the detailed response process is beyond the scope of this brief.

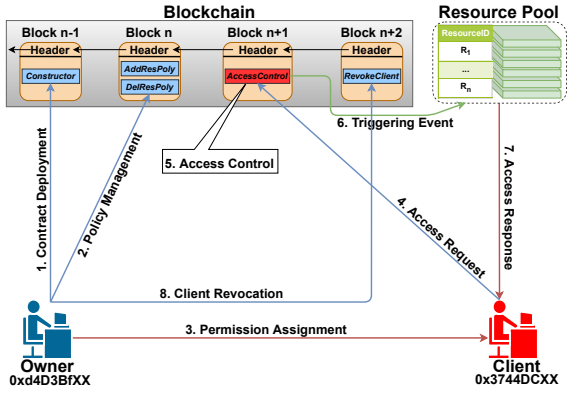

Fig. 1. System model of the smart contract-based access control framework

B. Concrete Design

The concrete design of our proposed smart contract-based access control framework is as follows. Some related notations are defined in Table II.

*1) **Contract Deployment**:* The owner O first creates an access control contract, and sends a transaction to deploy the contract on blockchain. During the deployment phase, as shown in Algorithm 1, several state variables whose values

TABLE II
NOTATIONS

Notations	Descriptions
O, OA	the resource owner and corresponding account address
C, CA	the client and corresponding account address
R, P	the specified resource and its related policy
S	the attribute set of client
$nonce$	a label bound with each client for revocation
$Sig_o(\cdot)$	the signature generated by the owner

are permanently stored in the contract storage are defined, including the address of the owner, a mapping between the resource and policy, and a mapping between the client address and nonce. In addition, an event interface to log the evaluation result is defined. In the *Constructor* function, which is only called once during deployment, the variable *owner* is initialized with the address of the owner, *i.e.*, the sender of the deployment transaction. Note that, the contract address and corresponding Application Binary Interface (ABI) are publicly released for subsequent invocations.

Algorithm 1 Contract Deployment

```
address owner
mapping(resourceID ⇒ policy) policies
mapping(address ⇒ nonce) clientNonce
event ResultEvent(address, resouceID, "allow")
function CONSTRUCTOR()                    ▷ Only invoked once
    owner = msg.sender
end function
```

*2) **Policy Management**:* After the contract is deployed, the owner is able to create transactions to call the $AddResPoly$ and $DelResPoly$ methods provided in the contract, with which the access policy bound with the resource could be added, modified and deleted from blockchain. As shown in Algorithm 2, both $AddResPoly$ and $DelResPoly$ will call the $require$ function to ensure that the resource policies can only be operated by the owner. Policy addition and modification can be achieved with $AddResPoly$, and deletion with $DelResPoly$. Note that, the *delete* operation in $DelResPoly$ will save some on-chain storage space of the contract.

Algorithm 2 Policy Management

```
Input: the target resource R and access policy P
function ADDRESPOLY(R, P)   ▷ Only invoked by owner
    require(msg.sender == owner)
    policies[R] = P
end function
function DELRESPOLY(R)      ▷ Only invoked by owner
    require(msg.sender == owner)
    delete policies[R]
end function
```

*3) **Permission Assignment**:* To add a client C into the system, the owner first defines a set of attributes S based on his/her system role or credentials. Then, it sends a message consisting of $\langle S, nonce, Sig_o(CA||S||nonce)\rangle$ to the client offline, where CA is the client's blockchain account address and $Sig_o(\cdot)$ means the signature of O on the specified message. Here, $nonce$ is initialized with 0 for each new client, and will be used in the client revocation phase.

*4) **Access Control**:* To initiate an **Access Request** on the resource R in the pool, the client C sends a transaction to call the $AccessControl$ function in the contract with the input of $\langle R, S, Sig_o(CA||S||nonce)\rangle$. As shown in Algorithm 3, in the $AccessControl$ procedure, it first calls the $SigVerify$ function to verify whether the submitted signature matches with the owner's account address OA. If it does, *i.e.*, the attribute set S and $nonce$ of C is valid, it then calls the $PolicyCheck$ function to check whether the access policy P related to R is satisfied by the client attribute set S, and triggers an "allow" event if it is (**Triggering Event**). Once catching the event, the resource pool will grant C the privilege to access R (**Access Response**). Note that, the built-in $assert$ function is used in $AccessControl$ for internal error checking, and $ecrecover$ is used to recover the address associated with the public key from the off-chain signature.

Algorithm 3 Access Control

```
Input: the target resource R, client attribute set S and
attribute signature Sig
function ACCESSCONTROL(R, S, Sig)          ▷ Invoked by client
    assert(SIGVERIFY(S, clientNonce[msg.sender], Sig))
    assert(POLICYCHECK(R, S))
    emit ResultEvent(msg.sender, R, "allow")
end function
function SIGVERIFY(S, nonce, Sig)          ▷ Invoked internally
    MessageSigned = msg.sender||S||nonce,
    if ecrecover(MessageSigned, Sig) == owner then
        return true
    else
        return false
    end if
end function
function POLICYCHECK(R, S)                 ▷ Invoked internally
    P = policies[R]
    if P is satisfied by S then
        return true
    else
        return false
    end if
end function
```

*5) **Client Revocation**:* To revoke the client C, the owner sends a transaction to call the $RevokeClient$ function in the contract, with the input of the account address of C. Specifically, as shown in Algorithm 4, the value of $nonce$ associated with C will be updated as $nonce + 1$, such that the previous message signed for C including the old value can not pass validation in the access control phase (in line 2 of Algorithm 3). Similarly, the $RevokeClient$ function can only be invoked by the owner itself. Note that, the privileges of a revoke client can be recovered or updated by the owner through distributing new signatures including the current nonce.

Algorithm 4 Client Revocation

Input: account address of the revoked client

1: **function** REVOKECLIENT(CA) ▷ Only invoked by owner
2: $require(msg.sender == owner)$
3: $clientNonce[CA]$++
4: **end function**

C. Discussion

We give a brief discussion on the proposed smart contract-based access control framework as follows.

1) ***Reliability****:* The reliability of the proposed framework is guaranteed by the unforgeability of off-chain signature and the credibility of on-chain operations. For one thing, the client address is contained in the off-chain signature signed by the owner and the smart contract is able to identify clients from the submitted transactions, such that only the authorized clients can be successfully validated in the $SigVerify$ procedure of the $AccessControl$ method. For another thing, based on the trustworthiness and consensus provided by blockchain, all the operations defined in the contract will be performed accordingly and automatically, and the state of contract cannot be tampered illegally.

2) ***Auditability****:* Traditional access control is enforced by a centralized and trusted server, resulting in the lack of transparency. In our proposed framework, particularly with a public blockchain, not only the related entities (the owner, clients and resource pool) in the system, but aslo any third party is able to keep listening to the contract event to verify the access decisions. In addition, blockchain has the potential to impact all recordkeeping processes, including the way transactions are initiated, processed, authorized, recorded, and reported. In this way, all the unauthorized and illegal operations could be traced. Hence, the auditability of the proposed framework could been guaranteed.

3) ***Scalability****:* The access rules and client identity are decoupled in our proposed framework. That is, the resources are bound with attribute-based access policies, and the client is described with a set of attributes but not a unique identity. To grant access rights to a new client, neither additional operations on access policies nor storage space for client permission information is required on blockchain, such that the number of clients will not affect the on-chain cost. As a result, the scalability of the proposed framework is significantly enhanced.

4) ***Revocation****:* The value of *nonce* bound with each client is initialized with 0 by default. By increasing *nonce* with 1, the previous signed message including the old value would fail validation in the access control phase, such that the revoked clients are not able to access any resource. Moreover, even a client is re-authorized, his previous privileges cannot be abused anymore. Therefore, revocation can be achieved effectively.

III. PERFORMANCE EVALUATION

We implement a prototype to evaluate the feasibility and performance of the proposed framework. Both the owner and client are instantiated on a notebook with an Intel Core i7-7600U CPU at 2.80GHz and 16GB RAM, running Windows 10 system. Ropsten is selected as the Ethereum test network, and the smart contract is written in Solidity (v0.5.13), and compiled, deployed and invoked in the Remix environment with the help of MetaMask. The off-chain signatures are implemented with the web3.js library in Nodejs environment, in which the signature algorithm (*i.e.*, ECDSA-secp256k1) for signing a transaction in Ethereum is used. The access policy is described with a threshold value k and a list L of m attributes. In this case, only if the number of the overlapped attributes between L and the client's attribute set S is no less than k, the access policy is satisfied. The contract and user account addresses used in our experiments are listed in Table III, and the related transactions are publicly visible on *https://ropsten.etherscan.io*.

TABLE III
CONTRACT AND USER ACCOUNT ADDRESSES

Type	Accont Address
Contract	0x467ad70a20813F59ef04869892A9eb458645E707
Owner	0xd4D3Bfc73812fa05AffCB0BCda855fB75B3C1b20
Client	0x3744DC77Ff25305Caf824acA61430DAF19AFe8b6

Ethereum uses a unit called gas to measure the amount of operations needed to perform a task, *e.g.*, deploying a smart contract or calling a function. In general, the more complex a task is, the more gas it consumes. Table IV gives a detailed comparison of the gas consumption of the main operations in the proposed framework with that in [15] and [16], in which the number of involved attributes is set as 5. Since only one contract needs to be deployed, the gas cost of contract deployment in our framework is the lowest. The gas cost of adding a policy mainly depends on the complexity of the access policy. Note that, extral gas cost of adding attributes for the resource and client is required in [16]. The calling between different contracts during access control in [15] and [16] makes them consume more gas than that in our framework. In total, the proposed framework effectively minimizes the on-chain gas consumption.

TABLE IV
GAS CONSUMPTION OF THE MAIN OPERATIONS

Operations	[15]	[16]	Ours
Deploying contracts	1,706,290	1,301,972	836,943
Adding a policy	128,777	363,964	165,582
Adding attributes	-	308,109	-
Access control	75,771	264,635	46,825
Total	1,910,838	2,238,680	1,049,350

Table V analyzes the relationship between the on-chain storage cost and the number of clients n. Considering that the ACL needs to be specified for each client in [15] and the attributes of each client need to be maintained in [16], the on-chain storage cost will increase linearly with the number of clients in both of them. While in our scheme, the client attributes are distributed off-chain and the number of clients does not affect the on-chain storage cost, thus the scalability is effectively improved.

Fig. 2 further shows the relationship between the gas cost of $AddResPoly$, $DelResPoly$ and $AccessControl$ with

TABLE V
THE RELATIONSHIP BETWEEN THE ON-CHAIN STORAGE COST AND THE NUMBER OF CLIENTS n

[15]	[16]	Ours
$O(n)$	$O(n)$	$O(1)$

the number of involved attributes in our framework. Since more storage and computation resources on blockchain are required for more attributes, all of them increase with the number of related attributes. The reason why the gas cost for $DelResPoly$ is far less than that for $AddResPoly$ is because the former one will release some storage space on blockchain and lead to a gas refund. For the $AccessControl$ method, the cost of signature verification and event triggering is fixed, thus the result is mainly determined by the procedure of policy checking. Note that, even for 10 attributes, only 64307 gas is consumed, demonstrating that the process of access control for client is quite efficient and economic.

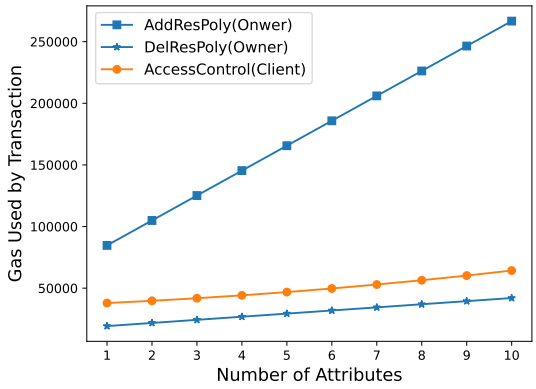

Fig. 2. The relationship between the gas cost of $AddResPoly$, $DelResPoly$ and $AccessControl$ with the number of involved attributes

IV. CONCLUSION

In this brief, we have proposed a smart contract-based access control framework. An access control contract has been designed and deployed on blockchain to manage attribute-based access policies of resources flexibly and make access decisions for clients reliably. A set of attributes is distributed to the clients through off-chain signatures signed by the owner to determine their privileges, without consuming the expensive on-chain storage space. The on-chain evaluation, mainly including signature verification and policy check, has been done to make trustworthy access decisions. The properties of decentralization, transparency, immutability and consensus provided by blockchain effectively guarantee the reliability and auditability of the access control functions. Finally, we have also conducted extensive experiments on Ethereum test network to demonstrate the scalability and efficiency of the proposed framework.

In the future work, we will mainly focus on the issues related to fine-grained privileges (read, write, execute, delegate, *etc.*), privacy preserving (zero knowledge proof and attribute-based signature), and the intergration of on-chain and off-chain resources (such as intel SGX).

REFERENCES

[1] R. S. Sandhu and P. Samarati, "Access control: principle and practice," *IEEE Communications Magazine*, vol. 32, no. 9, pp. 40–48, 1994.
[2] R. S. Sandhu, "Role-based access control," *Advances in Computers*, vol. 46, no. 1, pp. 237–286, 1998.
[3] S. Gusmeroli, S. Piccione, and D. Rotondi, "A capability-based security approach to manage access control in the internet of things," *Mathematical and Computer Modelling*, vol. 58, no. 5-6, pp. 1189–1205, 2013.
[4] V. C. Hu, D. R. Kuhn, D. F. Ferraiolo, and J. Voas, "Attribute-based access control," *Computer*, vol. 48, no. 2, pp. 85–88, 2015.
[5] J. Hao, W. Tang, C. Huang, J. Liu, H. Wang, and M. Xian, "Secure data sharing with flexible user access privilege update in cloud-assisted iomt," *IEEE Transactions on Emerging Topics in Computing*, vol. 10, no. 2, pp. 1–14, 2021.
[6] J. Hao, C. Huang, J. Ni, H. Rong, M. Xian, and X. S. Shen, "Fine-grained data access control with attribute-hiding policy for cloud-based iot," *Computer Networks*, vol. 153, no. 1, pp. 1–10, 2019.
[7] S. Rouhani and R. Deters, "Blockchain based access control systems: State of the art and challenges," in *ACM International Conference on Web Intelligence*. ACM, 2019, pp. 423–428.
[8] Z. Li, J. Hao, J. Liu, H. Wang, and M. Xian, "An iot-applicable access control model under double-layer blockchain," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 68, no. 6, pp. 2102–2106, 2020.
[9] A. Ouaddah, A. Abou Elkalam, and A. Ait Ouahman, "Fairaccess: a new blockchain-based access control framework for the internet of things," *Security and Communication Networks*, vol. 9, no. 18, pp. 5943–5964, 2016.
[10] D. D. F. Maesa, P. Mori, and L. Ricci, "Blockchain based access control," in *IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer, 2017, pp. 206–220.
[11] D. Lin, J. Wu, Q. Yuan, and Z. Zheng, "Modeling and understanding ethereum transaction records via a complex network approach," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, no. 11, pp. 2737–2741, 2020.
[12] J. P. Cruz, Y. Kaji, and N. Yanai, "Rbac-sc: Role-based access control using smart contract," *IEEE Access*, vol. 6, no. 1, pp. 12 240–12 251, 2018.
[13] R. Xu, Y. Chen, E. Blasch, and G. Chen, "Blendcac: A smart contract enabled decentralized capability-based access control mechanism for the iot," *Computers*, vol. 7, no. 3, pp. 39–46, 2018.
[14] Y. Nakamura, Y. Zhang, M. Sasabe, and S. Kasahara, "Capability-based access control for the internet of things: an ethereum blockchain-based scheme," in *IEEE Global Communications Conference*. IEEE, 2019, pp. 1–6.
[15] Y. Zhang, S. Kasahara, Y. Shen, X. Jiang, and J. Wan, "Smart contract-based access control for the internet of things," *IEEE Internet of Things Journal*, vol. 6, no. 2, pp. 1594–1605, 2018.
[16] M. Yutaka, Y. Zhang, M. Sasabe, and S. Kasahara, "Using ethereum blockchain for distributed attribute-based access control in the internet of things," in *IEEE Global Communications Conference*. IEEE, 2019, pp. 7–12.