HAL
open science

# IoTChain: A Blockchain Security Architecture for the Internet of Things

Olivier Alphand, Michele Amoretti, Timothy Claeys, Simone Dall 'Asta, Andrzej Duda, Gianluigi Ferrari, Franck Rousseau, Bernard Tourancheau, Luca Veltri, Francesco Zanichelli

## To cite this version:

Olivier Alphand, Michele Amoretti, Timothy Claeys, Simone Dall 'Asta, Andrzej Duda, et al.. IoTChain: A Blockchain Security Architecture for the Internet of Things. IEEE Wireless Communications and Networking Conference, Apr 2018, Barcelona, Spain. hal-01705455

**HAL Id: hal-01705455**

**https://hal.science/hal-01705455v1**

Submitted on 9 Feb 2018

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# IoTChain: A Blockchain Security Architecture for the Internet of Things

Olivier Alphand†, Michele Amoretti*, Timothy Claeys†, Simone Dall'Asta*, Andrzej Duda†, Gianluigi Ferrari*, Franck Rousseau†, Bernard Tourancheau†, Luca Veltri*, Francesco Zanichelli*
*Department of Engineering and Architecture, University of Parma, Italy
†Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG, F-38000 Grenoble, France

***Abstract*—In this paper, we propose IoTChain, a combination of the OSCAR architecture [1] and the ACE authorization framework [2] to provide an E2E solution for the secure authorized access to IoT resources. IoTChain consists of two components, an authorization blockchain based on the ACE framework and the OSCAR object security model, extended with a group key scheme. The blockchain provides a flexible and trustless way to handle authorization while OSCAR uses the public ledger to set up multicast groups for authorized clients.**

**To evaluate the feasibility of our architecture, we have implemented the authorization blockchain on top of a private Ethereum network. We report on several experiments that assess the performance of different architecture components.**

***Index Terms*—Communication security, Internet of Things, Group key distribution, Authorization, Blockchain**

## I. Introduction

The Internet of Things (IoT) concerns the integration of IP-enabled constrained devices with the existing Internet infrastructure. IoT systems are increasingly deployed, but the design of widely accepted standards to regulate the IoT ecosystem is a tedious process. In particular, security issues, such as authorization, verification and access control are still far from being completely solved [3].

Recently, the Internet Engineering Task Force (IETF) ACE [2] working group has proposed a generic framework for authentication and authorization in constrained environments. ACE based on OAuth2.0 [4] enables a third party, defined as a *client*, to access protected resources from a *resource server*, e.g., a smart device tracking the temperature. Instead of requiring the owner of the device (the resource owner) to disclose his/her credentials, access is regulated by tokens provided by an *authorization server*. At the same time, Vučinić et al. [1] proposed a promising architecture to provide end-to-end (E2E) security for transporting IoT data. The Object Security Architecture for the Internet of Things (OSCAR) addresses the main limitations of the Datagram Transport Layer Security (DTLS) protocol [5] by protecting the payload at the application layer. Such an approach allows for efficient multicast, asynchronous traffic, and caching. Resource servers store their protected resources either locally or in an encrypted and signed format on a proxy server. Clients request the decryption keys from the responsible key server. Access control can be provided by encrypting different resources with different keys.

In this paper, we propose IoTChain, a scheme that combines OSCAR and the ACE authorization framework to provide an E2E solution for secure authorized access to IoT resources. In the ACE framework, clients must set up an encrypted and authenticated channel with a trusted authorization server to securely exchange owner permissions and access tokens, which requires the use of certificates or out-of-bound secret sharing. Additionally, rogue authorization servers can freely issue access tokens for every protected resource. We replace the single trusted authorization server in the ACE framework by a trustless authorization blockchain. The authorization blockchain improves on the ACE authorization model by making resource access control robust, flexible, and possibly privacy-preserving. The blockchain consensus protocol requires that an attacker controls at least $51\%$ of the blockchain before she can obtain illegitimate tokens. In IoTChain, the resource owner describes the access rights in a smart contract, which then automatically generates access tokens for the client if certain conditions are met. Unlike ACE, the access token is not transmitted to the client, but safely stored in the smart contract internal storage. The smart contract can later be queried by other entities to check the token validity.

Furthermore, we use OSCAR in combination with a self-healing group key distribution scheme to enable efficient multicasting of IoT resources. A client sends a request to the key server to join the key distribution groups associated with the desired resources. The key server checks the smart contract storage on the blockchain to verify if a client has been authorized. Once the client has obtained the group keys, it downloads and decrypts the protected resources.

The paper is organized as follows. Section II presents the IoTChain architecture. Section III analyzes and discusses the security of the IoTChain architecture. Section IV illustrates an IoTChain prototype together with performance considerations. Section V summarizes the related work. Section VI concludes the paper with an outline of open issues and the future work.

## II. IoTChain Architecture

In this section, we introduce the IoTChain architecture and present its main advantages. Fig. 1 illustrates the main elements of the architecture and presents the sequence of operations leading to authorized access to IoT resources. To avoid confusion concerning the nomenclature and the roles of

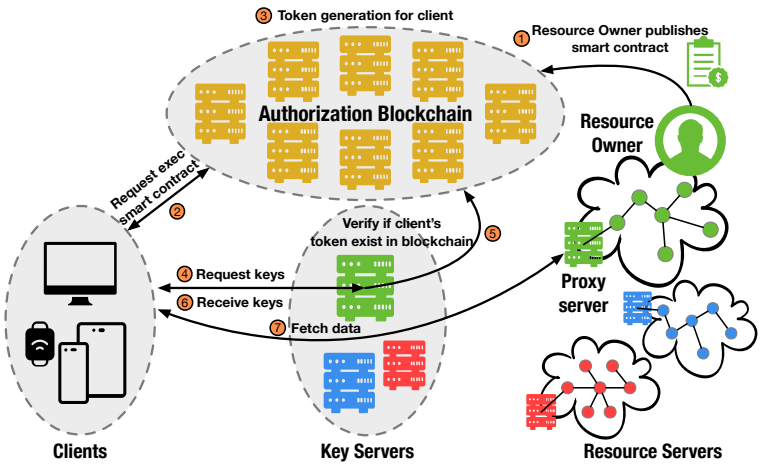


Fig. 1. The IoTChain blockchain-based security architecture (source of icons: [6]).

different entities, we follow the terminology specified by the IETF:

- **Resource Servers** generate and store protected resources.
- **Resource Owners** are the legal owners of the resource servers and their generated resources.
- **Clients** are third-parties that request access to the protected resources.
- **Proxy Servers** store the resources in an encrypted form when the resource servers are highly constrained.
- **Key Servers** generate the necessary keys to encrypt and decrypt the resources.
- **Access tokens** describe the access rights for a specific client and a specific resource.
- **Authorization Servers** generate access tokens.

### A. Authorization blockchain

In general, a blockchain can be considered as a persistent log whose records are stored in timestamped blocks. Each block contains transactions. A block is identified by its cryptographic hash and it references the hash of the preceding block. Anything that is stored in the blockchain is public.

The blockchain is maintained by nodes having a copy of at least the last $n$ blocks. In our architecture, the authorization servers, key servers, and clients act as nodes, although not all of them necessarily store the whole blockchain and participate in the consensus protocol. The authorization servers and key servers are full nodes, which means that they store the complete history of the blockchain. The authorization servers also act as miners. They verify the transactions on the blockchain and store them in blocks. The blocks are then sealed through a Proof-of-X consensus protocol, e.g., Proof-of-Stake or Proof-of-Work, and appended to the blockchain. The blockchain is a public network, anybody can run an authorization server. The key servers are responsible for the keying material of the resource servers and are set up by the resource owners. Each client and resource owner is identified by at least one blockchain address, i.e., an asymmetric key pair. Interactions with the smart contracts and between clients and resource owners are made through transactions, which are signed with the private key part of the respective addresses. To send transactions, clients and resource owners either take part in the network storing the blocks or they connect to a blockchain node that broadcasts their transactions on the blockchain network.

### B. OSCAR

The key server in OSCAR [1] generates personal keys and passes them to resource servers that use them to take part in the self-healing group key distribution process (see Subsection II-E for details). The keys that result from such a process are used by the resource servers to encrypt their protected resources. Different resources, generated on the same resource server might be encrypted with different keys to enforce access control and different privilege levels. The personal keys are sent over a DTLS channel between the key server and the resource server. In a later stage, authorized clients obtain the group keys through the key server using DTLS. We assume that the key server and resource servers have valid certificates issued by a certification authority.

### C. Authorization Flow

In the **first phase**, the resource owner creates a smart contract and publishes it to the blockchain, see (1) in Fig. 1. A smart contract is a compiled program that has its own blockchain address. The contract generates an access token for a client when certain conditions are met. The access token describes the specific access rights for the protected resources. The specific layout of a token is application dependent, but a generic form is described by Jones et al. [7].

In the **second phase**, a client that wishes to access a protected resource, activates the corresponding smart contract by sending a transaction to the contract address (2). Because the contracts are deployed through normal transactions, a client can verify who published the contract by checking the address that signed the publishing transaction. The transaction to the contract is broadcast to all the nodes in the network. Every miner that includes the transaction in its block, will validate the transaction and execute the smart contract. The transaction

is only valid when the client includes the correct data for the execution of the smart contract.

For example, an energy company, the resource owner, has deployed a smart meter, a resource server, at the client house. To authorize the clients, the energy company publishes a smart contract on the blockchain. The smart contract can require a proof that the client lives at a specific address. This proof is added in the data field of the transaction, which is then sent to the the smart contract. When executed, the contract generates a token for the client. The token references the address of the client (i.e., a public key). It has a lifetime and describes which resources can be accessed. A resource owner can deploy multiple smart contracts for the same resource server on the blockchain. Each contract takes different input parameters and generates tokens with different privileges.

In the **third phase**, the block containing the contract transaction has been added to the blockchain and the token has been added to the contract internal storage (3). The client then requests the encryption keys necessary to decrypt the resources from the key server (4). The key server has a copy of the blockchain. The key server queries the internal storage of the responsible smart contract for the access token (5). To prove the authenticity of the client, the key server creates a challenge-response based on the client address referenced in the token. Only the legitimate client that triggered the smart contract can solve the challenge. After successfully completing the challenge, the client receives a personal key (6) and takes part in the self-healing group key distribution process (see Subsection II-E for details). To prevent issues with temporary blockchain forks, the key server must wait for $n$ blocks to be built on top of the block containing the token creation, $n$ being a security parameter. Large $n$ gives the key server strong guarantees on the validity of the token but might incur larger latency, e.g., in the Ethereum network 12 block confirmations are required, which approximately corresponds to 3 min latency.

During the **final fourth phase**, the client can download the encrypted resources either from a proxy server or directly from the resource server (7). Both entities provide a RESTful CoAP API that allows to *GET*, *PUT*, *POST* resources based on their Uniform Resource Identifier (URI). No authentication is necessary as an unauthorized client cannot obtain the cryptographic keys to decrypt the protected resources. When the protected resources are directly obtained from the resource server, the integrity of the protected resources can be secured with a symmetric key. This integrity key is also obtained from the key server. When the protected resources are published to a proxy server, they need to be protected with an asymmetric signature, which prevents colluding proxy servers and clients from corrupting the integrity of the protected resources.

### D. Adding and Revoking Entities

*1) Clients:* Clients can be easily added to the system. By providing the necessary information in a transaction to a smart contract, a token will be generated that allows the new client to access the decryption keys for a specified resource. To easily revoke clients, new encryption keys should be issued on a frequent basis by the key server. A token can for example be valid for a month. The key server can distribute new personal keys to the resource server on a daily basis. The resource server will always use the latest keys to encrypt new resources, e.g., new temperature measurements. As long as the client token is valid, the client can recover every day the new keys from the key server. When the token expires the client must rerun the smart contract on the blockchain to generate a new valid token and obtain the access rights for the keys, see Fig. 2. When the client misbehaves, the authorization servers in the blockchain network can add a transaction in block, which removes the client access token from the smart contract internal storage. The key server will receive this block once it is added to the blockchain and will know that the client is no longer authorized to receive the decryption keys to be a member of the key distribution group.

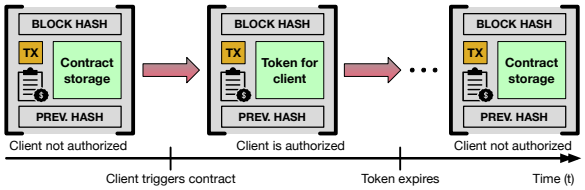


Fig. 2. Token life time on the blockchain (source of icons: [6]).

*2) Authorization and Key Servers:* New blockchain nodes (i.e., key or authorization servers) join the blockchain network by contacting active nodes and downloading the whole blockchain. The bootstrap information to join a blockchain network can be provided by bootnodes. Bootnodes provide an initial list of peers to which a new node can connect.

Every node is represented by a unique identifier generated by a node when joining the network for the first time. The nature of the blockchain protocol allows for rogue entities. Revoking is therefore not necessary but would also not be practical without an overseeing trusted third-party.

### E. Self-Healing Group Key Distribution

The self-healing group key distribution is a method enabling large and dynamic groups of users to create group keys for secure multicast communication over unreliable networks [8].

In self-healing group key distribution schemes, there is a Group Manager (GM) for each group. In the IoTChain architecture, the key server plays the role of the GM. Every authorized group member $U_i$ receives personal key $S_i$ from the GM upon joining the group. To keep the personal key secret, it is transmitted using DTLS.

Time is divided into sessions. As the $j$-th session begins, the GM sends the common key $B_j$ to all authorized members of group $G_j$ on a public authenticated channel. Common key $B_j$ is created by GM from group key $K_j$ and has the following properties:

- there is an efficient algorithm $\eta$ such that $K_j = \eta(B_j, S_i)$ for all $i : U_i \in G_j$;
- there is no computationally viable $\zeta$ allowing to recover $K_j$ for any set of nodes $R \in U \setminus G_j$.

Personal keys $S_i$ are valid in a limited number of consecutive sessions.

If some broadcast message $B_j$ gets lost, users are still able to recover the group key for the $j$-th session, by combining information from any message $B_l$ preceding the lost one with the information from any message $B_r$ following it.

## III. Security Considerations

### A. Token security

Similarly to ACE, IoTChain also proposes a Proof-of-Possession (PoP) concept to bind the client's identity to an access token. When a client requests a token in ACE, the ACE authorization server binds a cryptographic key, dubbed the PoP-key, to the token. The PoP-key is known to the client that originally requested the token and can be accessed by the resource server that receives the token. Based on the PoP-key, the resource server creates a challenge-response to verify if the client presenting the token is the legitimate owner. In IoTChain, the client triggering the smart contract provides its address (i.e., a public key). This public key is included in the access token, stored in the blockchain. The key server that verifies the token will create a challenge-response with the address bound to the token, which verifies the identity of the client requesting access to the decryption keys.

In ACE, it suffices for an attacker to compromise one authorization server to obtain illegitimate tokens. In IoTChain, an attacker needs to compromise at least $51\%$ of the miners.

### B. Client Privacy

The blockchain is a public ledger, therefore all the transactions stored in the blocks can be read by anyone. A client can protect its privacy by generating new addresses for each transaction. Doing so allows the client to isolate each of its transactions in such a way that it is harder for an attacker to associate them all together. Smart contracts and by extension the resource owners cannot see what other contract the client has triggered.

### C. Denial-of-Service

There are several techniques we can use to prevent Denial-of-Service (DoS) attacks on the blockchain infrastructure. The main issue is that because of the halting problem, we cannot predict if a smart contract deployed on the blockchain will terminate. The Ethereum blockchain [9] uses the concept of *gas* that a user has to pay for the execution of a smart contract. If the user does not pay for sufficient gas, the contract will not be fully executed and all the changes will be rolled back. Because we do not require a cryptocurrency and the smart contracts only generate access tokens, we can simply set a limit on the execution time of a smart contract.

A malicious client might attempt to launch a DoS attack on the network by constantly triggering smart contracts. Because all the transactions need to be broadcast to the rest of the network and every authorization server needs to verify these transactions, the network may become saturated. A simple defense might be to require each client to solve a cryptographic puzzle before each transaction. A similar approach is proposed in the HIP protocol [10].

### D. Secure Communications

The personal keys are exchanged over DTLS channels between the key server, the resource servers, and the clients. Authentication between the resource servers and the key server is achieved on the basis of certificates, and between the clients and the key server, through a challenge-response. Transactions sent in the blockchain have signatures to protect their integrity.

### E. Bootstrapping Key Server

As stated in Section II-D2, the new nodes need to contact bootnodes to synchronize to the blockchain. If a bootnode is not authenticated, an attacker can mount a man-in-the-middle attack (MITM) and point a key server to her personal blockchain where she can alter freely the history. To prevent this attack, the bootnodes should use certificates that allow the new nodes to verify their authenticity.

## IV. Implementation

### A. Ethereum Private Testnet

To evaluate the feasibility of our architecture, we have implemented the authorization blockchain on top of a private Ethereum blockchain network, based on the go-ethereum (geth) implementation of the protocol released by the Ethereum foundation (see https://github.com/ethereum/go-ethereum). Geth provides the possibility to connect to the Ethereum testnet. The testnet lets developers test and debug smart contracts for free. Because syncing with the entire Ethereum testnet would take too much time and resources, we decided to mount a local private Ethereum blockchain. The blockchain network consisted of three full nodes, storing the entire local blockchain, see Fig. 3.

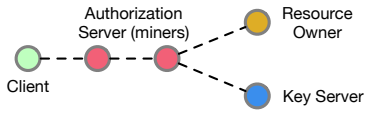


Fig. 3. Private P2P blockchain network. The miners and key server are full nodes, storing the entire blockchain.

The resource owner connects to a full node and deploys the smart contract shown in Fig. 4. Once the contract is added to the blockchain, clients can interact with it by calling its individual public functions. The function `addToken` will create an access token for a client. The access token is stored in the contract persistent memory. The token contains the client address and the resource server address. The `ttl` field of the access token holds a lifetime parameter, checked by the key server when the client requests a decryption key. The `deleteToken` function might be called to revoke the client access rights earlier than foreseen. The smart contract in Fig. 4 can be triggered by everyone in the blockchain network. More complex contracts may require *modifiers* that restrict access to certain functions or check if a condition is met before the

function is executed by the Ethereum Virtual Machine (EVM), e.g., has this client paid for the access token.

```
pragma solidity ^0.4.0;

contract AccessToken{
    mapping (address => Token) issued;

    struct Token{
        address res;
        uint ttl;
        bytes4 claims;
    }

    function addToken(...) public returns(bool){
        issued[clt].res = _res;
        issued[clt].ttl = _ttl;
        issued[clt].claims = _claims;
        return true;
    }

    function getToken(...) public constant returns(...){
        return (issued[clt].res,
                issued[clt].ttl,
                issued[clt].claims);
    }

    function deleteToken(...) public returns(bool){
        delete(issued[clt]);
    }
}
```

Fig. 4. A simple smart contract deployed in our local blockchain.

### B. Performance Evaluation

We have executed several experiments to assess the performance of different IoTChain components.

*1) Key Server:* The key server has been executed in a virtual machine provided with Ubuntu 16.04 LTS. In Fig. 5, we show the average time required to complete the DTLS handshake, considering an increasing number of clients deployed on a Windows 10 machine equipped with an Intel Core i7-4600M CPU @ 2.90 GHz (4 virtual cores), 8 GB RAM.

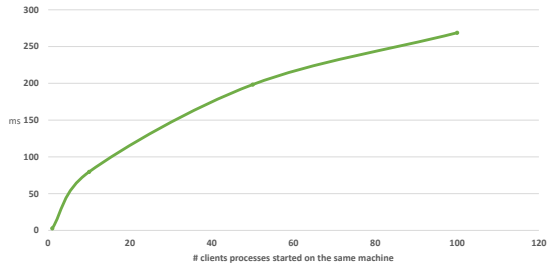


Fig. 5. Average time required to complete the DTLS handshake between the key server for an increasing number of clients.

*2) Resource Server:* In Fig. 6, we illustrate (1) the response time for a GET request from a client to a resource server, (2) the response time for a PUT request from a client to a resource server, (3) the time required to complete the DTLS handshake between a resource server and the key server. The resource server has been implemented in C language and deployed on a Raspberry Pi 2. In Fig. 7, we show the time required by the resource server for signing, verification, encryption and decryption.

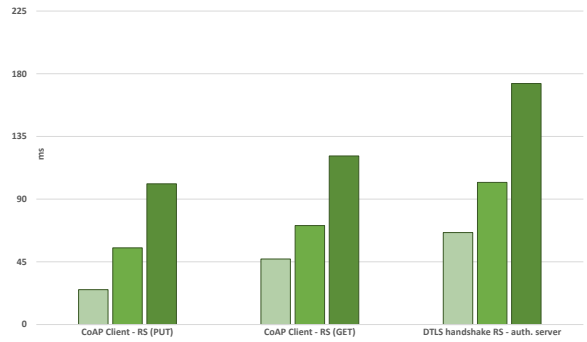


Fig. 6. Performance of the resource server, considering different requests from the clients and the DTLS handshake with the key server.

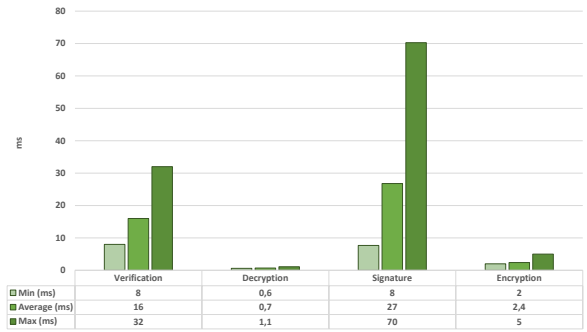


Fig. 7. Performance of the resource server, considering signing, verification, encryption, and decryption.

*3) Cloud-Based Proxy Server:* We have deployed a proxy server on the OpenStack-based cloud installed at the Department of Engineering and Architecture of the University of Parma. This facility includes one master server and several slave servers running a dynamic set of virtual machines (VMs). All the servers host a *Ubuntu 14.04 LTS* operating system and are equipped with two QuadCore Intel Xeon @ 2.00 GHz processors and 16 GB of RAM @ 800 MHz. All VMs host a *Ubuntu 14.10 Utopic* operating system and have one Intel Core i7 @ 2.0 GHz processor, 1 GB of RAM, and 3GB of disk space.

In Fig. 8, we present the performance of the operations (including encryption/decryption) involving the cloud-based proxy server in case of PUT (from a resource server) and GET (from a client) requests. Each request to the Cloud crosses three tiers: REST frontend (executing on the master server), dispatcher (a Java-based software component we have developed and integrated with the OpenStack-based system, running on the master server) and proxy server (running on the replicated VMs).

## V. Related Work

Motivated by the recent explosion of interest around blockchains, Christidis and Devetsikiotis [11] examined

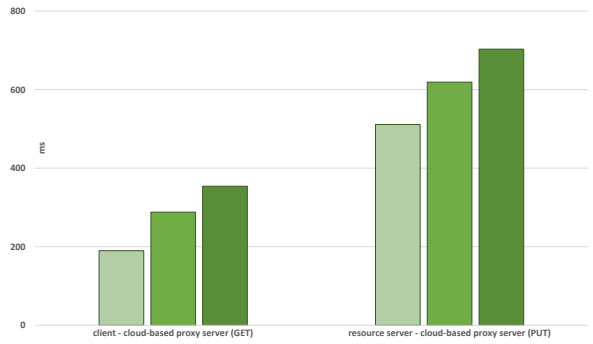


Fig. 8. Performance of the cloud-based proxy server.

whether they make a good fit for the IoT sector. Their conclusion was that the blockchain-IoT combination is powerful and can cause significant transformations across several industries, paving the way for new business models and novel, distributed applications. In particular, the blockchain-IoT combination facilitates the sharing of services and resources and allows one to automate in a cryptographically verifiable manner several existing, time-consuming workflows. For this reason, several partial architectures and prototypes have been recently proposed. Due to the lack of space, we cannot list them all. Instead, we focus on those that are more intriguing and challenging, in our opinion.

Samaniego and Deters [12] proposed a novel architecture where software-defined IoT devices are combined with a permissioned blockchain for provisioning IoT services on edge hosts. It would be interesting to study the possibility to implement software-defined IoT devices as smart contracts.

Kravitz and Cooper [13] proposed a methodology for creating permissioned blockchain ecosystems, where identity and attribute management is built in for both users and devices. Identities, and their associated attributes, transcend any single device or group of devices. The proposed framework is interesting, but no implementation is currently available.

Dorri et al. [14] studied the possibility to build a lightweight, optimized blockchain for resource-constrained devices. Overtaking the energy-consuming Proof-of-Work (PoW) mechanism adopted by major blockchains to sculpt and add data blocks is a major research issue not only for the IoT community. A Proof-of-Stake (PoS) version of Ethereum (denoted as Casper) is going to be released, promising to be more efficient and secure. We argue that Casper will further improve the possibility to build robust blockchain-IoT systems.

## VI. Conclusion

In this paper, we have proposed IoTChain, an architecture that combines the elements of OSCAR and the ACE authorization framework. The idea is to make the ACE authorization phase trustless and flexible. To this purpose, we use a blockchain to replace the single ACE authorization server. The blockchain handles the authorization requests through smart contracts. Execution of a smart contract adds a token to the contract storage to authorize a client. We also use a self-healing group key distribution scheme to allow efficient multicasting of IoT resources. Clients request keys from the key server to join the key distribution groups associated with the desired resources.

In the future work, we plan to implement different applications on top of IoTChain, to further evaluate its robustness and performance. Moreover, we will update our private Ethereum blockchain network to use the PoS-based version of the ledger, as soon as it will be released by Ethereum developers.

## Acknowledgments

This work has been partially supported by the French Ministry of Research projects DataTweet under contract ANR-13-INFR-0008-01, the PERSYVAL-Lab under contract ANR-11-LABX-0025-01, the FUI IoTize project funded by Région Auvergne-Rhône-Alpes, and the project PHC Galilée 2017, PROJET N° 37409RJ, “Security protocols for the Cloud-oriented Internet of Things (SeCIoT)”, the University of Parma Research Fund - FIL 2016 - Project “NEXTALGO: Efficient Algorithms for Next-Generation Distributed Systems”.

## References

[1] M. Vučinić, B. Tourancheau, F. Rousseau, A. Duda, L. Damon, and R. Guizzetti, “OSCAR: Object Security Architecture for the Internet of Things,” *Ad Hoc Networks*, vol. 32, pp. 3 – 16, 2015.

[2] L. Seitz, G. Selander, E. Wahlstroem, S. Erdtman, and H. Tschofenig, “Authentication and Authorization for Constrained Environments (ACE),” Internet Engineering Task Force, Internet-Draft draft-ietf-ace-oauth-authz-07, Aug. 2017, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-ace-oauth-authz-07

[3] F. A. Alaba, M. Othman, I. A. T. Hashem, and F. Alotaibi, “Internet of Things Security: A Survey,” *Journal of Network and Computer Applications*, vol. 88, pp. 10 – 28, 2017.

[4] D. Hardt, “The OAuth 2.0 Authorization Framework,” Internet Requests for Comments, RFC Editor, RFC 6749, October 2012, http://www.rfc-editor.org/rfc/rfc6749.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6749.txt

[5] E. Rescorla and N. Modadugu, “Datagram Transport Layer Security Version 1.2,” Internet Requests for Comments, RFC Editor, RFC 6347, January 2012, http://www.rfc-editor.org/rfc/rfc6347.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6347.txt

[6] (2017) The Noun Project - Icons for Everything. [Online]. Available: https://thenounproject.com/

[7] M. Jones, H. Tschofenig, E. Wahlstroem, and S. Erdtman, “CBOR Web Token (CWT),” IETF, Internet-Draft draft-ietf-ace-cbor-web-token-05, Jun. 2017, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-ace-cbor-web-token-05

[8] T. Rams and P. Pacyna, “A Survey of Group Key Distribution Schemes With Self-Healing Property,” *IEEE Communications Surveys & Tutorials*, vol. 15, no. 2, pp. 820–842, 2013.

[9] G. Wood, “Ethereum: A Secure Decentralised Generalised Transaction Ledger,” *Ethereum Project Yellow Paper*, vol. 151, 2014.

[10] R. Moskowitz, T. Heer, P. Jokela, and T. Henderson, “Host Identity Protocol Version 2 (HIPv2),” Internet Requests for Comments, RFC Editor, RFC 7401, April 2015, http://www.rfc-editor.org/rfc/rfc7401.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc7401.txt

[11] K. Christidis and M. Devetsikiotis, “Blockchains and Smart Contracts for the Internet of Things,” *IEEE Access*, vol. 4, pp. 2292–2303, 2016.

[12] M. Samaniego and R. Deters, “Using Blockchain to Push Software-Defined IoT Components Onto Edge Hosts,” in *Proceedings of the International Conference on Big Data and Advanced Wireless Technologies*, ser. BDAW ’16, 2016.

[13] D. W. Kravitz and J. Cooper, “Securing User Identity and Transactions Symbiotically: IoT meets Blockchain,” in *2017 Global Internet of Things Summit (GIoTS)*, June 2017, pp. 1–6.

[14] A. Dorri, S. S. Kanhere, and R. Jurdak, “Towards an Optimized BlockChain for IoT,” in *Proceedings of the Second International Conference on Internet-of-Things Design and Implementation*, ser. IoTDI ’17, 2017, pp. 173–178.