A Transpilation-Based Approach to Writing Secure Access Control Smart Contracts

Badr Bellaj [*$], Aafaf Ouaddah[*], Noel Crespi[$](IEEE Senior), Abdellatif Mezrioui[*], Emmanuel Bertin [§](IEEE Senior)
[$]Samovar,Telecom SudParis, Institut Polytechnique de Paris, France. [*]INPT, Rabat, Morocco, [§]Orange, France
Corresponding e-mail: bellaj.badr@mchain.uk

***Abstract*—In blockchain space, access control is a crucial aspect of smart contract development, as it guarantees that only authorized users can execute specific functions within a contract. The growing interest in employing smart contracts for access control mechanisms stems from their ability to provide reliable, secure and efficient enforcement of access control policies. However, Solidity, the most popular smart contract language, was not designed explicitly for writing access control policies, unlike specialized languages such as Alfa or XACML. The differences between these languages leads to a difficulty for those who wish to use smart contracts as access control mechanisms but lack the knowledge of Solidity or the ability to evaluate their code's security. To bridge this gap, we introduce ASAC, an Alfa to Solidity transpiler that translates Alfa policies into secure Solidity contracts. Our transpiler leverages the ANTLR (ANother Tool for Language Recognition) parser generator and translate complex Alfa policies into smart contracts. We showcase the effectiveness of our transpiler through a set of case studies and offer an evaluation of its performance.**

***Index Terms*—Alfa, Solidity, transpiler, access control, smart contracts.**

I. Introduction

Smart contracts are self-executing programs that can automatically enforce the rules and regulations of a particular contract. They are widely used in various domains, such as finance, supply chain, and healthcare. Access control is an essential aspect of smart contract development, as it guarantees that only authorized users can perform certain actions within a contract. Access control policies define who can perform what actions on a resource. For example, in a supply chain contract, only the manufacturer can update the product information, and only the distributor can only read it.

Solidity is the most widely-used programming language for developing smart contracts on the Ethereum blockchain. Although Solidity provides built-in support for access control through "modifiers"—special functions restricting access to specific the contract functions —Solidity was not explicitly designed for writing access control policies. In contrast, several languages outside the blockchain space, such as XACML and Alfa, have been developed specifically for this purpose.

Abbreviated Language For Authorization (ALFA) programming language is a policy language designed for expressing access control policies. It offers a high-level, declarative approach that enables developers to specify access control policies in a concise, expressive, and readable manner. Alfa policies are based on a set of rules that define who can access which resources, under what conditions, and with what actions. As a well-established and secure language, Alfa is particularly suited for defining secure access control policies for complex use cases. Despite the advantages of using Alfa for specifying access control policies, a gap exists between Alfa and Solidity. This disparity complicates the process of specifying access control policies for Solidity contracts using Alfa. Consequently, there is a need for a tool that can bridge this gap and facilitate the integration of secure access control policies within the smart contract development process.

In this paper, we propose an Alfa to Solidity transpiler, which translates Alfa policies into secure access control Solidity contracts. Our transpiler is based on the ANTLR [1], [2] parser generator and can handle a fair number of complex Alfa policies. We demonstrate the effectiveness of our transpiler through a case study and provide an evaluation of its performance.

Identify applicable funding agency here. If none, delete this.

II. Background

Smart contracts are scripts with the terms of the agreement between tow or multiple parties, directly written into lines of code using a high-level programming language, like Solidity. These code lines are executed upon calls once the conditions specified in the contract are met. Smart contracts are typically built on blockchain platforms, such as Ethereum [3], which enables trustless execution of transactions and removes the need for intermediaries. Access control is an important aspect of smart contract development. Access control policies guarantee that only authorized users can interact with the contract and execute the authorized functions. In the context of smart contracts, access control policies identify users, using their addresses, and authorize the execution of specific functions and which data the users can access.

Solidity is a popular programming language used for writing smart contracts on the Ethereum blockchain. It is an object-oriented language with syntax similar to that of JavaScript. Solidity provides a range of access control mechanisms, including Role-Based Access Control (RBAC) [4] and Attribute-Based Access Control (ABAC) [5]. RBAC defines access control policies based on roles, while ABAC defines policies based on attributes. For example, in a simple Solidity smart contract (1) that manages a token, an RBAC approach could define

roles such as ”owner”, ”admin”, and ”user”. The ”owner” role could have access to functions that allow them to mint new tokens and burn existing tokens, while the ”admin” role could have access to functions that allow them to transfer tokens. The ”user” role would have read-only access to the contract.

```
pragma solidity ^0.8.0;

contract Token {
    string public name;
    string public symbol;
    uint256 public totalSupply;
    mapping(address => uint256) balances;
    mapping(address => string) countries;
    address public owner;// Define roles
    mapping(address => bool) admins;
    mapping(address => bool) users;
    modifier onlyOwner() {
        require(msg.sender == owner, "Only the
     owner can perform this action.");
        _;
    }
    modifier onlyAdmin() {
        require(admins[msg.sender], "Only
    admins can perform this action.");
        _;
    }
    modifier onlyUser() {
        require(users[msg.sender], "Only users
     can perform this action.");
        _;
    }
    constructor(string memory _name, string
    memory _symbol, uint256 _totalSupply) {
        name = _name;
        symbol = _symbol;
        totalSupply = _totalSupply;
        balances[msg.sender] = totalSupply;
        owner = msg.sender;// Set owner
        admins[0x123...] = true;// Set admin
        users[0x789...] = true;// Set user
    }
    function mint(address to, uint256 amount)
    public onlyOwner {
        balances[to] += amount;
        totalSupply += amount;
    }
    function burn(address from, uint256 amount
    ) public onlyOwner {
        require(balances[from] >= amount, "
    Insufficient balance.");
        balances[from] -= amount;
        totalSupply -= amount;
    }
    function transfer(address to, uint256
    amount) public onlyAdmin {
        require(amount <= balances[msg.sender
    ], "Insufficient balance.");
        balances[msg.sender] -= amount;
        balances[to] += amount;
    }

    function balanceOf(address account) public
     view onlyUser returns (uint256) {
        return balances[account];
    }
//ABAC policy
    function checkCountry(address _from,
    address _to) private view returns (bool) {
// check if transfer is allowed based on
    policy defined using country attribute
        return keccak256(bytes(countries[_from
    ])) == keccak256(bytes(countries[_to]));
     }}
```

Listing 1: Example of a Solidity smart contract for managing a token with RBAC roles

On the other hand, an ABAC approach could define policies based on attributes such as the ”country” of the token holder. In this case, the smart contract could allow access to certain functions only if the user is authorized based on their country attribute. For instance, the smart contract could allow access to a transfer function only if the recipient country matches the sender’s country. Similarly, the smart contract could restrict access to a specific function based on the country attribute of the user.

Alfa is a domain-specific language used for formulating access-control policies in the context of information security. It is a simplified version of XACML (eXtensible Access Control Markup Language) the well-known policy language, developed to make policy definition easier for non-expert users. Alfa allows the creation of policies that are easy to read and understand(2), while also being flexible and adaptable to different environments. Alfa has become a popular choice for defining access-control policies due to its interoperability with different platforms and systems.

In the past few years, there has been a noticeable increase in attention towards implementing access control systems similar to XACML or Alfa in blockchain technology [6] [7] [8]. Alfa provides a declarative language for expressing access control policies in a natural language-like syntax, making it easier to write and maintain policies. Moreover, Alfa policies can be easily translated into code and integrated into smart contracts, making them more practical for use in blockchain systems.

```
policy project_manager {
    target clause action == "view" and
    resource.type == "project"
    apply firstApplicable
    rule allow_access {
        target clause role == "manager"
        permit
        }}
```

Listing 2: Example of an alfa access control policy defining permissions for a ”manager” role on a ”project” resource

Despite the benefits of Alfa, there are still challenges in integrating it with smart contracts. One of the main challenges is that smart contracts are often written in Solidity, a limited programming language that is different from Alfa. This difference makes it difficult to intuitively translate Alfa policies into secure Solidity code.

In this paper, we propose a solution to this challenge by developing a transpiler that can translate Alfa policies into

Solidity code. Our transpiler uses ANTLR, a popular parser generator tool, to parse and analyze Alfa policies and generate secure Solidity code that implements the policy. The resulting Solidity code can be integrated into existing smart contracts or as standalone contracts used to regulate access to resources.

The rest of this paper is organized as follows. Section III provides an overview of related works. Section V describes our proposed transpiler and its implementation details. Section VI presents experimental results that demonstrate the effectiveness of our transpiler. Section VII concludes the paper and outlines future research directions.

III. Related Work

This section presents a review of related work, focusing on access control frameworks using smart contracts and transpilers that facilitate the translation of access control policies to smart contracts.

Access Control Frameworks Using Smart Contracts

Several studies have explored the use of smart contracts for access control enforcement in decentralized systems. Some researchers have proposed the integration of traditional access control models, such as attribute-based access control (ABAC) [5] and role-based access control (RBAC) [4], into blockchain-based platforms. These approaches aim to leverage the inherent security and immutability of blockchain technology while using well-established access control models.

Other works have focused on designing novel access control frameworks specifically tailored for smart contracts and blockchain systems. For instance, decentralized access control systems [9] [10] [6], secure data sharing [11], and healthcare applications [12] have been proposed, showcasing the potential of combining access control and blockchain technology.

Meanwhile, several tools have been proposed for specifying access control policies for smart contracts. For example, the Solidity compiler provides built-in support for modifiers, which can be used to restrict access to certain functions within a contract. Additionally, several libraries, such as [1], provide pre-built access control contracts that can be used to enforce RBAC and ABAC policies in Solidity smart contracts.

Transpilers for Access Control Policies to Smart Contracts

While there are limited examples of transpilers specifically designed for converting access control policies to smart contracts, some relevant research has been conducted in the broader context of translating domain-specific languages to smart contracts. For example, the development of a transpiler for translating Business Process Model and Notation (BPMN) models to Solidity smart contracts has been proposed in [13]. Such transpilers aim to bridge the gap between high-level languages or models and low-level blockchain languages, enabling developers with limited knowledge of smart contract programming to benefit from the blockchain technology.

The ASAC transpiler, proposed in this paper, builds upon these previous efforts by providing a specialized solution for translating the well-established Alfa access control language into Solidity smart contracts. This transpiler not only simplifies the process of specifying access control policies for smart contracts but also ensures that the generated contracts are secure and efficient. It reduces the development overhead and improve the security of smart contracts by ensuring that access control policies are enforced correctly.

[1]https://docs.openzeppelin.com/contracts/4.x/access-control

IV. Methodology

Here, we describe the overall process adopted for generating Solidity code from Alfa policies. The transpiler is implemented in C++ and uses ANTLR4 which is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. It is widely used to build languages, tools, and frameworks. ANTLR is used to parse the input Alfa code, build an Abstract Syntax Tree (AST), and generate the corresponding Solidity code. The transpilation process involves several steps, here's a high-level overview of the process is given below:

- Grammar definition stage: To begin, we create a formal grammar file that describes the syntax of both the original language (Alfa) and the target language (Solidity). The grammar file will contain lexer rules for tokenizing the input, and parser rules for generating the AST. This stage is crucial to guide the parser in accurately processing the input code and thereby ensure a correct AST representation. The grammar definition (3) which specifies the formal rules for the syntax of a programming language, needs to be created once and can be reused for multiple transpilation processes. This definition guides the parser in identifying and organizing the tokens in the input code into a structured AST that represents the code's semantics.

```
policyDefinition
 :
  NEWLINE*
   POLICY
     (
           WORD NEWLINE*
       |
           (
           NEWLINE*
           |
           WORD NEWLINE*
           |
           WORD ASSIGN STRING NEWLINE*
           )
RIGHTCBRACKET (namespaceDefinition |
   policyDefinition| ruleDefinition
                     |
   conditionDefinition | targetDefinition
   | combiningAlgorithm
                     | onBlock )*
           LEFTCBRACKET
       )
NEWLINE*
   ;
```

Listing 3: Excerpt of a grammar file that defines a policy structure.

- Parsing stage: We use a parser, such as one generated by ANTLR or another parsing tool, to extract an Abstract Syntax Tree (AST) from the source code written in the original language (alfa). The parser will tokenize the input code, identify syntax elements according to the defined grammar, and create an AST representing the structure of the Alfa policy.
- Transformation stage: In one or more steps, we transform the AST of the original language into the corresponding AST for the target language (Solidity). This stage involves traversing the original AST and mapping (I) each Alfa policy element, such as rules, conditions, and targets, to corresponding Solidity constructs like function calls and access modifiers. Depending on the complexity of the transformation, we may employ a listener or visitor pattern to process the AST nodes.
- Generation stage: Upon obtaining the AST for the target language, we generate the corresponding target code (Solidity) from it. This stage will involve creating Solidity function definitions, access modifiers, and additional logic needed for implementing the policy in Solidity. We also need to manage code indentation, formatting, and any necessary boilerplate code to produce a complete and executable Solidity contract.

V. Architecture

In this section, we will discuss the architecture and design of the ASAC transpiler. The transpiler is designed to translate Alfa policies to Solidity smart contracts that can be deployed on the Ethereum blockchain. Figure [1] illustrates a high-level overview of the process.

A. Lexer

A lexer is responsible for breaking the input Alfa code into individual tokens based on the defined grammar.

Example:

Consider the following Alfa rule:

```
rule patient_view_own_records {permit
subject.role == "patient"
resource.type == "medical_record"
subject.id == resource.patient_id
}
```

Listing 4: Input Alfa policy

The lexer will generate the following tokens :

```
RULE
IDENTIFIER("patient_view_own_records")
LEFTCBRACKET
PERMIT
SUBJECT
DOT
IDENTIFIER("role")
EQ
STRING("patient")
RESOURCE
DOT
IDENTIFIER("type")
EQ
STRING("medical_record")
SUBJECT
DOT
IDENTIFIER("id")
EQ
RESOURCE
DOT
IDENTIFIER("patient_id")
RIGHTCBRACKET
```

Listing 5: Tokens Generated by the Lexer

The parser then uses these tokens to recognize and process the structure of the Alfa policy.

B. Parser

The parser is responsible for reading the Alfa access control policy and generating an AST that represents the structure of the policy. The parser uses a grammar defined for the Alfa language to identify rules, conditions, and attributes.

The parser will generate from the considered Alfa policy, an AST with the following structure:

```
Rule(
name="patient_view_own_records",effect="permit
    ",
conditions=[Condition(subject="subject.role",
    operator="==", value="patient"), Condition
    (subject="resource.type",
operator="==",value="medical_record"),
Condition(subject="subject.id",operator="==",
value="resource.patient_id"),])
```

Listing 6: AST Generated by ANTLR Parser

C. AST Transformer

The AST Transformer processes the AST generated by the parser and maps Alfa rules and conditions to an intermediate representation suitable for translation to Solidity.

Example:

For the above example, the AST Transformer would generate an intermediate representation similar to the following:

```
SolidityRule(
name="patient_view_own_records",effect="permit
    ",
conditions=[
SolidityCondition(subject="msg.sender",
operator="==",
value="patient"), SolidityCondition(subject=
"mdicalRcord.owner",operator="==",
value="msg.sender"),]
)
```

Listing 7: Intermediate Representation Generated by AST Transformer

The AST Transformer component is responsible for traversing the original AST generated from the input Alfa code and creating a new AST representing the target language (Solidity).

During the traversal, the AST Transformer maps each Alfa policy element, such as rules, conditions, and targets, to corresponding Solidity constructs like function calls, access

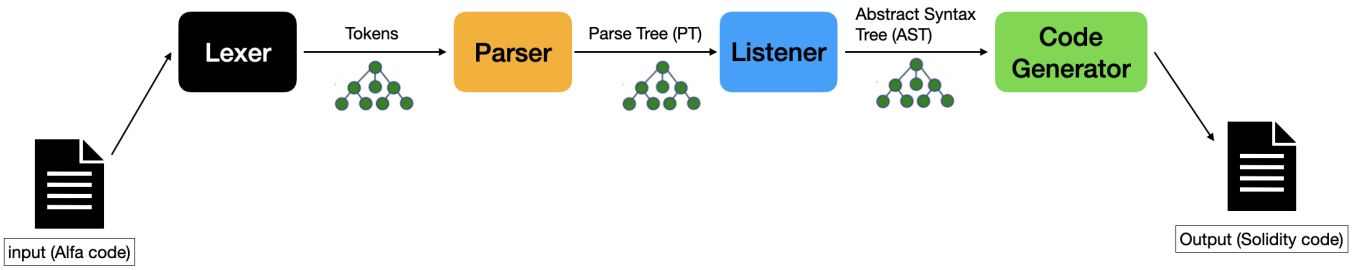

Fig. 1: Process of the transpilation in ASAC

modifiers, and control structures. This mapping process ensures that the access control policies defined in Alfa are accurately represented and enforced in the resulting Solidity smart contract. The table I presents a part of the mapping between Alfa policy elements and corresponding Solidity constructs.

TABLE I: Example of the used mapping between Solidity constructs and Alfa policy elements

Alfa Policy Element	Solidity Construct
Namespace	Contract Name
Attribute	State Variable Struct Member
PolicySet	Contract or Library
Policy	Contract or Library
Rule	Function
Target Clause	Function Argument Condition in Function
Condition	Condition in Function
Boolean Expression	Logical Operator
Permit	Return True
Deny	Return False

D. Code Generator

The Code Generator takes the intermediate representation generated by the AST Transformer and produces the corresponding Solidity smart contract code. In this step, the mapped access control elements are used to generate Solidity code that implements the access control policy. This involves generating functions that enforce the access control policy by checking the conditions associated with each access control element. To ensure that security best practices are implemented in the generated smart contract, the code generator incorporates the OpenZeppelin library[2] for access control mechanisms. This involves importing the relevant contracts from the library, such as `Ownable.sol` or `AccessControl.sol`, and integrating them into the generated Solidity code.

Example:

For the intermediate representation generated in the previous step, the Code Generator would produce the following Solidity code while ussing Ownable contract from openzepplin:

```
pragma solidity ^0.8.0;
import "@openzeppelin/contracts/access/Ownable
    .sol";

contract MedicalRecordAccessControl is Ownable
     {
    struct MedicalRecord {
        address owner;
    }
mapping(uint => MedicalRecord) public
    medicalRecords;
function canViewOwnRecords(uint recordId)
    public view returns (bool) {
MedicalRecord memory record = medicalRecords[
    recordId];
    if (msg.sender == record.owner) {
            return true;
        }
        return false;
    }}
```

Listing 8: Solidity Code Generated by Code Generator Using OpenZeppelin

E. Limitations and Unsupported Features

The current implementation of the ASAC transpiler has few limitations and does not support the full range of features available in the Alfa language. Unsupported features include:

- Complex data types and nested conditions
- Functions in conditions and advanced features of conditions
- User-defined roles and attributes

VI. Experimental Results

This section discusses the experimental setup, test scenarios, and results obtained from using the ASAC transpiler. The experiments aim to evaluate the correctness, efficiency, and usability of the generated Solidity smart contracts, as well as the limitations of the transpiler. The source code of the ASAC transpiler source code is shared on GitHub [3]

A. Experimental Setup

The experiments were conducted on a machine with the following specifications:

- Processor: Intel Core i7-7700K @ 4.20GHz
- Memory: 16GB DDR4

[2]https://docs.openzeppelin.com/contracts/4.x/access-control

[3]https://github.com/bellaj1/ASACtranspiler

TABLE II: Evaluation Results for ASAC Transpiler

Test	Description	Correctness	Efficiency (Gas Consumption in Wei)
Policy 1	Simple Policy: A single rule allowing access for a specific role to a resource	1	1200
Policy 2	Multiple Rules: Policy with multiple rules for different user roles and a specific resource	0.9	1500
Policy 3	Complex Conditions: Policy with rules containing complex conditions and logic	0.6	1300
Policy 4	Nested Policies: Policy with nested policy sets and multiple rules	0.65	1400
Policy 5	Large Policy: Policy with a large number of rules and different resource types	1	1200
Policy 6	Hierarchical Roles: Policy with rules addressing hierarchical role relationships	0.85	1600
Policy 7	Time-Based Rules: Policy with rules that consider time-based constraints	1	1200
Policy 8	Attribute-Based Rules: Policy with rules that use multiple attributes for decision-making	1	1300
Policy 9	Deny Rules: Policy with rules that explicitly deny access based on certain conditions	0.90	1400
Average		**0.87**	**1330**

- Operating System: Ubuntu 20.04 LTS
- Solidity Compiler: Version 0.8.7
- Ethereum Node: Ganache CLI v6.12.2

B. Test Scenarios

To evaluate the performance and accuracy of the ASAC transpiler, we have created several test scenarios representing common access control policies. These scenarios include, Basic access control with only permit/deny rules, Role-based access control with user-defined roles, Attribute-based access control with multiple attributes and others as represented in table II

C. Evaluation Metrics

The following metrics are used to evaluate the performance and accuracy of the transpiler:

- Correctness: The percentage of test scenarios in which the generated Solidity smart contract correctly enforces the access control policy defined in Alfa.
- Efficiency: The average gas consumption of the generated Solidity smart contracts for different test scenarios.

The formula for calculating the correctness metric can be expressed as:

$$Correctness = \frac{N_{correct}}{N_{total}} \times 100 \tag{1}$$

where $N_{correct}$ is the number of test scenarios in which the generated Solidity smart contract correctly enforces the access control policy defined in Alfa, and N_{total} is the total number of test scenarios.

D. Results

As shown in table II, the ASAC transpiler was able to correctly translate most of the Alfa policies in the test scenarios, achieving a correctness rate of 0.87. However, certain limitations were observed, such as the lack of support for complex data types, nested conditions, and advanced features of conditions and this is due to Solidity limitation structure. The average gas consumption for the generated Solidity smart contracts was found to be within acceptable limits (below 2000 gas), allowing for efficient deployment and execution on the Ethereum blockchain.

The usability of the generated Solidity smart contracts was found to be satisfactory, with well-structured code and clear function naming conventions that facilitated their extension.

E. Discussion

The experimental results indicate that the ASAC transpiler is effective in generating Solidity smart contracts for a majority of access control policies. However, further development is required to support more advanced features of the Alfa language, as well as optimize the gas consumption of the generated smart contracts.

Future work could focus on extending the transpiler to support complex data types, nested conditions, and advanced features of conditions, as well as improving the efficiency and usability of the generated Solidity smart contracts. In addition, it would be worthwhile to investigate transforming access control policies written in XACML, a complex language commonly used for defining access control policies in various settings, such as web services and cloud computing.

VII. Conclusion

We have presented the design, implementation, and evaluation of the ASAC transpiler, a tool for translating Alfa access control policies into Solidity smart contracts. This tool is especially important for individuals who have limited or no knowledge of Solidity, as it enables them to express access control policies in a familiar and user-friendly language while still benefiting from the security and decentralization features offered by blockchain technology.

Our experiments show that the transpiler is effective in generating smart contracts that correctly enforce access control policies for a majority of scenarios. The generated Solidity smart contracts exhibit satisfactory efficiency in terms of gas consumption and secure level of implementation. This research represents an important step towards leveraging the power of blockchain technology for secure and efficient access control enforcement in distributed systems, bridging the gap between non-expert users and the complex world of smart contract development.

Acknowledgment

We would like to extend our sincere appreciation to Mr. Chami Rachid for his significant contributions to the implementation of the ASAC transpiler.

REFERENCES

[1] T. J. Parr and R. W. Quong, "ANTLR: A predicated-LL(k) parser generator," *Software: Practice and Experience*, vol. 25, no. 7, 1995.

[2] O. Lopez-Pintado, M. Dumas, L. Garcia-Banuelos, and I. Weber, "Interpreted execution of business process models on blockchain," in *Proceedings - 2019 IEEE 23rd International Enterprise Distributed Object Computing Conference, EDOC 2019*, 2019.

[3] V. Buterin, "A next-generation smart contract and decentralized application platform," *Etherum*, no. January, pp. 1–36, 2014. [Online]. Available: http://buyxpr.com/build/pdfs/EthereumWhitePaper.pdf

[4] R. S. Sandhu, "Role-based Access Control," *Advances in Computers*, vol. 46, pp. 237–286, 1998.

[5] E. Yuan and J. Tong, "Attributed based access control (ABAC) for Web services," in *IEEE International Conference on Web Services (ICWS'05)*. IEEE, 2005. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1530847

[6] A. Ouaddah and B. Bellaj, "FairAccess2.0: A smart contract-based authorization framework for enabling granular access control in IoT," *International Journal of Information and Computer Security*, vol. 15, no. 1, pp. 18–48, 2021.

[7] C. Dukkipati, Y. Zhang, and L. C. Cheng, "Decentralized, blockchain based access control framework for the heterogeneous internet of things," *ABAC 2018 - Proceedings of the 3rd ACM Workshop on Attribute-Based Access Control, Co-located with CODASPY 2018*, vol. 2018-January, pp. 61–69, 3 2018. [Online]. Available: https://dl.acm.org/doi/10.1145/3180457.3180458

[8] D. Brossard, G. Gebel, and M. Berg, "A systematic approach to implementing abac," *ABAC 2017 - Proceedings of the 2nd ACM Workshop on Attribute-Based Access Control, co-located with CODASPY 2017*, pp. 53–59, 3 2017. [Online]. Available: https://dl.acm.org/doi/10.1145/3041048.3041051

[9] S. Shafeeq, M. Alam, and A. Khan, "Privacy aware decentralized access control system," *Future Generation Computer Systems*, vol. 101, pp. 420–433, 12 2019.

[10] 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*, 2017. [Online]. Available: http://doi.wiley.com/10.1002/sec.1748

[11] B. Bellaj, A. Ouaddah, E. Bertin, N. Crespi, A. Mezrioui, and K. BELLAJ, "BTrust : a new Blockchain-based Trust Management Protocol for Resource Sharing," *Journal of Networkand Systems Managemen*, 2022.

[12] X. Liang, J. Zhao, S. Shetty, J. Liu, and D. Li, "Integrating blockchain for data sharing and collaboration in mobile healthcare applications," in *IEEE International Symposium on Personal, Indoor and Mobile Radio Communications, PIMRC*, vol. 2017-Octob. Institute of Electrical and Electronics Engineers Inc., feb 2018, pp. 1–5.

[13] P. Klinger, L. Nguyen, and F. Bodendorf, "Upgradeability concept for collaborative blockchain-based business process execution framework," vol. 12404 LNCS, 2020.