2018 IEEE Confs on Internet of Things, Green Computing and Communications, Cyber, Physical and Social Computing, Smart Data, Blockchain, Computer and Information Technology, Congress on Cybermatics

Blockchain Based Access Control Services

Damiano Di Francesco Maesa, Paolo Mori
Istituto di Informatica e Telematica
Consiglio Nazionale delle Ricerche, Pisa, Italy
Email: d.difrancesco@for.unipi.it, paolo.mori@iit.cnr.it

Laura Ricci
Department of Computer Science
University of Pisa, Italy
Email: laura.ricci@unipi.it

***Abstract*—This paper presents a new design approach for Access Control services leveraging smart contracts provided by blockchain technology. The key idea of our proposal is to codify Access Control policies as executable smart contracts on a blockchain. This transforms the policy evaluation process into completely distributed smart contract executions. In our fully blockchain based approach also the Attribute Managers required for the evaluation of the Access Control policies are managed by the blockchain, i.e., they are implemented as smart contracts as well. To study the feasibility of our proposal we present a working reference implementation using XACML policies and Solidity written smart contracts deployed on Ethereum. Finally we evaluate the advantages and drawbacks of the proposal, making also use of experimental results of our reference implementation.**

I. Introduction

Nowadays, most digital resources such as data or services are exposed on the Internet and they need to be protected by a proper security support granting access only to those subjects actually holding the corresponding rights. Access Control systems are meant for this and they express the rights of subjects to access resources by means of Access Control policies, which are evaluated against the current access context each time an access request is received to make the access decision [1]. In some scenarios, the right of performing the access is even continuously verified for the whole duration of the access itself, in order to interrupt a previously authorized access as soon as this right expires because of a change of the access context [2]. Several models have been presented in the scientific literature to define different kind of controls. Among them, the Attribute-based Access Control (ABAC) model [3] represents the access context through a set of attributes describing the relevant features of the subjects, resources and environment, and the Access Control policies consist of a set of conditions over these attributes.

The Access Control systems can be deployed, managed and run by the resource owners on their premises or by third parties trusted by the resource owners. The management of an Access Control system could represent a relevant cost for the resource owner, who, to avoid these costs, can outsource the evaluation of its security policies to external systems. For instance, in the last years, a number of Access Control systems implemented as independent services on top of the Cloud following the Software as a Service (SaaS) paradigm have been proposed (e.g., [4], [5]). These services often use an open-platform API, in a way such that users are not locked to a specific implementation, and they can exploit them to have a uniform management of policy enforcement for all the resources they own. As for the other Cloud services, each user pays the usage of the Access Control Services on a per use basis.

This paper proposes an alternative design for Access Control systems provided as a Service which is based on blockchain technology. In our proposal, the Access Control policy is represented as a smart contract which is stored on the blockchain through a proper transaction, and it is executed through other transactions each time an access request needs to be evaluated to make an access decision. The blockchain is also exploited to manage a number of attributes which are required for the evaluation of the Access Control policy. The proposed approach presents some relevant advantages w.r.t. traditional Access Control Services. For instance, the subjects issuing access requests are guaranteed against unduly denial of access. In fact, adopting a traditional Access Control system, the party which actually evaluates the policy and enforces the result on the resource could maliciously force the system to deny the access to a subject although the policy would have granted it. Instead, in our blockchain scenario, the subjects are enabled to verify how the policy was enforced when they performed an access request which has been suspiciously denied.

The paper is structured as follows: Section II presents a brief background, focusing on the XACML standard architecture presentation, as well as the few available related work. We do not provide a background on blockchain technology, assuming a basic knowledge of its concepts. In Section III we describe the general architecture of our blockchain-based Access Control system, while Section IV presents our proof-of-concept implementation based on Ethereum. In Section V we discuss the advantages and disadvantages of our approach and, finally, Section VI presents our conclusions and future work.

II. Background and Related Work

A. XACML Standard

This section gives a very brief description of the eXtensible Access Control Markup Language (XACML, defined by the OASIS consortium [6]) reference architecture (shown in Figure 1), which will be referred in the rest of the paper. For a detailed description of the XACML standard and its reference architecture see [6].

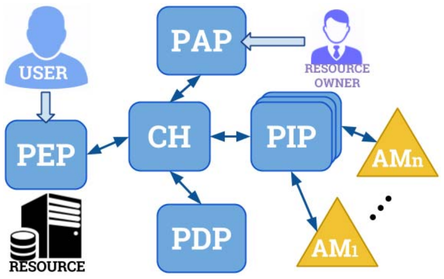


Fig. 1. XACML reference architecture.

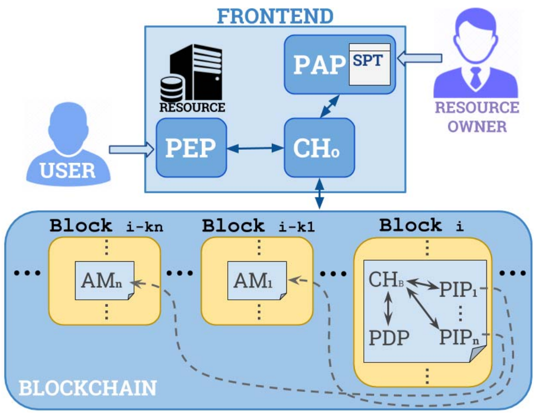


Fig. 2. Architecture of the blockchain based Access Control Service.

1) Policy Enforcement Point (PEP): The Policy Enforcement Point is the component, paired with the resource to be protected, which is able to intercept the access requests, it triggers the decision process, and enforces the related result by actually allowing or denying the execution of the access.

2) Policy Administration Point (PAP): It is the component in charge of managing Access Control policies.

3) Attribute Managers (AMs): AMs are the components that manage the attributes of subjects, resources, and environment, allowing to retrieve and update their values. They could run in the authorization service itself, or they could be run in other machines in the same domain, in other administrative domanins, or even by third parties.

4) Policy Information Points (PIPs): The set of attributes required for the policy evaluation are, in general, managed by distinct Attribute Managers. Policy Information Points are the interfaces for interacting with Attribute Managers allowing to retrieve the latest values of such attributes and to update them.

5) Policy Decision Point (PDP): The Policy Decision Point is the evaluation engine that takes a policy, an access request, and the attribute values as input, evaluates the policy and returns the decision (i.e., access permitted or denied).

6) Context Handler (CH): The Context Handler is the component which acts as orchestrator of the decision process, interacting with the other components of the architecture to manage the workflow of the decision process.

B. Related Work

At the best of our knowledge, due to the novelty of the topic, only a few proposals of blockchain related Access Control systems have been presented. In [7] the authors combine blockchain and off-chain storage to build a personal data management platform focused on privacy. [8] proposes a new framework for blockchain based Access Control focused on IoT. A blockchain based lightweight and robust Access Control framework addressing the security and privacy issues in Big Data is introduced in [9].

III. Blockchain-Based Access Control Service

This paper proposes a novel and alternative approach to implement Access Control Services exploiting blockchain technology. Outsourcing the Access Control functionalities to third party Access Control services is quite common today, because this relieves the resource owner from the burden of configuring and maintaining complex Access Control systems that would cause relevant costs both for acquiring the hardware and software equipments and for their management and administration.

The basic idea underlying our approach is to build an Access Control Service on top of a blockchain, i.e., to exploit a blockchain both to store Access Control policies and to perform the access decision process (i.e., to evaluate the relevant policies every time an Access Control request is issued by a user who wants to access a resource). To this aim, we represent an Access Control policy through a smart contract, called SMART POLICY, which is stored on the blockchain with a proper transaction when the resource owner creates (or updates) it. Since the blockchain is an append only ledger, once uploaded on the blockchain a SMART POLICY will be stored on it forever. However, it can be updated or even disabled with a proper transaction. Each time a subject issues an access request, a proper transaction on the blockchain is created by our service. This transaction includes a reference which causes the evaluation of the SMART POLICY and the production of the related access decision (e.g., Permit or Deny). The evaluation of such policy is completely executed on the blockchain, as we detail in the following. For the sake of simplicity, here and in the following we say that "the policy evaluation is executed on the blockchain" meaning that this SMART POLICY execution is replicated among the miners elaborating the new block to be added to the blockchain.

The solution we propose in this paper is focused on ABAC policies, although we think that it could be easily extended to cover other Access Control models. An ABAC policy consists of a set of rules (see Section I) that are combined exploiting proper combining algorithms (e.g., simple logical operators such as AND, OR, or other algorithms) and they must be satisfied accordingly in order to grant the requested access. For the policy writing, we adopt XACML because it is a very expressive language allowing to write complex ABAC policies. Moreover, since it is a well known standard, some tools for policy editing and management are available from both academic and business organizations. An XACML policy is properly translated into a smart contract, the SMART POLICY, in order to store it on the blockchain and execute it when necessary. Hence, the SMART POLICY can be seen as an executable version of the XACML policy. In other words, following the XACML naming, we could say that the SMART POLICY embeds a Policy Decision Point (PDP) customized for the execution of a specific Access Control policy. The attributes required for the evaluation of the policies are stored on the blockchain as well, and they are managed by a set of proper smart contracts. Following the XACML naming, the entities which created those contracts are the Attribute Providers, and the contracts storing the set of attributes released by the same entity can be seen as the Attribute Managers of such Attribute Providers. Hence, we will call such smart contracts SMART AMs. A SMART AM is invoked by a SMART POLICY to retrieve the current values of the attributes it manages. Moreover, the part of the SMART POLICY which invokes the SMART AMs could be seen as the Policy Information Points (PIPs), i.e., the components of the XACML reference architecture devoted to the management of the attributes required for the evaluation of the policy.

The architecture of the proposed system is depicted in Figure 2. In the following of this section, we describe in details how we represent Access Control policies, how we create and store them on the blockchain and how we evaluate them by retrieving the required attributes to produce an access decision as a consequence of an access request. In particular, with reference to Figure 1, we describe how the architectural components in charge of the previous tasks according to the XACML standard are defined on top of blockchain technology in the proposed system. Do note that in our framework, the traditional coordination tasks of the CH are split between two components: an off chain component and the blockchain protocol itself (respectively, CH_O and CH_B in Figure 2).

The architecture of our Blockchain-based Access Control system is independent from the underlying blockchain technology chosen, provided that such blockchain supports smart contracts (the proposed implementation, however, is based on a specific blockchain, Ethereum). In a traditional Access Control system, the only information the resource owner has to provide to initialize the Access Control system is the XACML policy. However, although the usage of our system does not require technical knowledge of the underlying blockchain technology, further data could be required to use it depending on the blockchain chosen. For example in an Ethereum style blockchain, gas needs to be paid [10]. Any user that wants to use such a system needs to own a wallet holding some funds. So the user needs to provide a XACML policy alongside some (not sensible) wallet informations, and it will be required to perform additional operations (e.g. signing blockchain transactions) to use the system functionalities.

A. New policy creation

The first step of the system life cycle is the policy creation. First of all, the resource owner writes an XACML policy which defines the access rights on the resource(s). The policy is submitted to the PAP with the task of storing it on the blockchain. A simple solution would be to simply store the policy expressed in XACML (even compacted or referenced) on the blockchain (as proposed in [11]). In this paper, instead, we leverage smart contracts capabilities to define a more powerful solution. In particular, before storing the policy on the blockchain, the PAP translates the logic expressed by the XACML policy into a smart contract, the SMART POLICY. The SMART POLICY is not a simple rewriting of such policy, but it contains all the logic for its execution as well. For instance, each XACML statement referring to an attribute is translated inserting into the smart contract a function call to retrieve the current attribute value from the corresponding attribute manager each time the SMART POLICY is invoked. This is possible because in our approach the Attribute Managers are represented as smart contracts stored on the blockchain as well, the SMART AMs. Furthermore, the SMART POLICY also encapsulates the logic for the policy evaluation process. Summarizing, codifying a policy as a SMART POLICY allows us to write blockchain executable policies from XACML ones. Such SMART POLICIES perform the tasks that in traditional Access Control systems are delegated to the PDP and PIP. In fact, the decision process of the PDP is now performed through the smart contract execution, and the attribute values retrieval process of the PIP is achieved through smart contract function calls. In order to be revoked, the SMART POLICY also contains a self destruct function callable only by the owner. Do note that in an immutable blockchain a deleted contract is not actually removed, but rather marked as not callable.

Once the PAP finishes translating the XACML encoded policy into a SMART POLICY, such smart contract is compiled and deployed on the blockchain by the CH_O. The expenses related to the SMART POLICY deployment are paid by the policy creator. This is correct since the policy creator is the entity that benefits from having a controlled access on the resource. Finally, when the SMART POLICY gets accepted by the blockchain, its address (or any other form of contract linking provided by the specific blockchain technology adopted) is stored by the PAP on the SMART POLICY Table (SPT) in the entry corresponding to the resource it refers to.

B. Access request time

Once a SMART POLICY has been created and stored on the blockchain, the blockchain based Access Control service starts waiting for access requests. The PEP is embedded in the piece of code which implements the interface to access the protected resource in such a way that it intercepts each new access request. The component in charge of bridging the request with the underlying blockchain, instead, is the CH_O. Hence, when the PEP intercepts an access request, it forwards it to the CH_O. The CH_O interacts with the PAP sending it the ID of the resource in the access request to identify the SMART POLICY to be invoked on the blockchain. The PAP returns to the CH_O the address of this SMART POLICY by retrieving it in the SPT exploiting the ID of the resource as index to locate the right entry in the table. In order to trigger the execution of such

SMART POLICY, the CH_O translates the access request in the proper format. This usually means simply encoding it into a message (i.e., blockchain transaction) to be sent to the SMART POLICY, set with the correct parameters. This message will be processed by the blockchain triggering the SMART POLICY to be executed (possibly executing other smart contracts of the AMs needed) and eventually producing a Permit or Deny result. The CH_O reads back from the blockchain the decision and informs the PEP that then will actually enforce the decision by granting or denying the request accordingly. The expenses of the evaluation transaction are paid by the subject making the request (since the subject will be the entity benefiting from the granted access). This prevents subjects from spamming requests to the system, since they are limited by the value they own. The drawback is that the subject needs to manage a wallet holding value on the underlying blockchain.

IV. Proof of Concept Implementation on Ethereum

In order to validate and evaluate the proposed approach, we have developed a proof of concept implementation of the blockchain based Access Control System presented in this paper. We have chosen the Ethereum blockchain protocol (as of December 2017), because it is strongly focused on smart contracts and because it is nowadays a widely used smart contract ready blockchain protocol. We then chose Solidity [12] as programming language to write our smart contracts and Java to write the off-chain side of our framework.

To deploy and test our system, we used the *International Educational blockchain* academic testnet (part of the *Open Blockchain* initiative [13]). This is an Ethereum based private testnet with nodes currently run by North American and European universities, and it allowed us to have an environment at the same time controlled and somewhat realistic. To interact with the testnet blockchain we used one of the most used Ethereum clients: `geth` [14]. We did not use the Ethereum main chain because of the obvious cost constraints, and also to avoid to burden the immutable Ethereum main chain with our test data that are intended to be temporary.

To allow our Java client to interact with `geth` we used the `web3j` [15] Java library. `web3j` is a lightweight library that, among its many functionalities, supports all of the JSON-RPC API offered by `geth`.It provided us with the needed tools to bridge the blockchain components of the system with the off-chain components.

We note that our system implements many utility functionalities for the system manager and the users (for example to obtain the list of the last *n* submitted access requests), but in the following we will only focus on the two main operations of new policy creation and access request, due to space constraints.

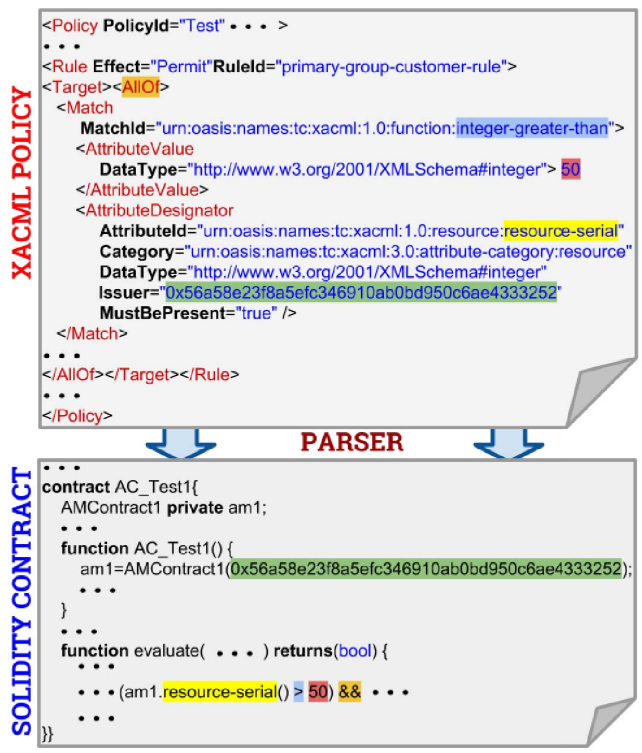
A. PEP

The PEP is written in Java and it is the component of the system interacting with the protected resource. Our system is designed as a program pluggable to already existing Access Control scenarios. This is why the PEP is inserted in the access interface of the resource and provides an API with a method to submit an access request (written in XACML) which returns a response (written in XACML as well). The PEP was left intentionally lightweight since its main task is to bridge our system with the resource and subjects. The logic functionalities of blockchain interaction are delegated to other internal components. This is why the PEP only forwards the received requests to the CH_O, possibly enriched with additional information, if available.

B. PAP

The PAP is written in Java and it is the software component in charge of policy management and retrieval. Its main tasks are to transform an XACML Access Control policy written by the resource owner into a SMART POLICY, and to remember the mapping between SMART POLICIES and resources. As a matter of fact, the PAP embeds a XACML parser that translates the policy in a smart contract written in Solidity. The smart contract generated contains some utility functions which are the same for all policies (e.g., a suicide function that is invoked to revoke the SMART POLICY) and data (e.g., the address of the resource owner, marked as private field). The main function of the SMART POLICY is called `evaluate` and represents the executable version of the XACML policy. In the following, we show how the XACML policy is translated in Solidity to produce the body of the `evaluate` function.

An XACML policy consists of a policy Target and a set of rules, each including their Targets and Conditions [6]. We focus our description on the translation of the Targets and rules of the policy, being the translation of the Conditions very similar. A Target is a combination of `<Match>`..`</Match>` elements, each such element will be called MATCHE in the rest of this paper. Each MATCHE is translated as a check of the `evaluate` function. The type of the Solidity function to be used to implement the check is derived from the XACML `MatchId` field and the data type from the XACML `DataType` field of `<AttributeValue>` and `<AttributeDesignator>`. Each check, to be performed, needs the current value of an attribute at access request time. Hence, the SMART POLICY must be also able to retrieve these values in order to compute the decision result. This would be a task of a PIP in a traditional XACML system. In our proposal we assume the existence of an ecosystem of SMART AMs (the smart contracts having the function of AMs deployed, maintained and advertised by third parties), and we integrate the PIPs functionalities in the SMART POLICY. For example the smart contract of an institution could offer public informations about its employees (e.g. their role). SMART AMS could also require some way of payment (either in or off chain) for the use of their services, so a market of AMs could naturally emerge. For the sake of simplicity, in this paper we assume to deal with static attributes only, i.e., attributes which rarely change their values as a consequence of administrative actions typically requiring human intervention. Any resource owner who creates a new policy needs to specify



```
<Policy PolicyId="Test" • • • >
• • •
<Rule Effect="Permit"RuleId="primary-group-customer-rule">
<Target><AllOf>
  <Match
    MatchId="urn:oasis:names:tc:xacml:1.0:function:integer-greater-than">
    <AttributeValue
       DataType="http://www.w3.org/2001/XMLSchema#integer"> 50
    </AttributeValue>
    <AttributeDesignator
       AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-serial"
       Category="urn:oasis:names:tc:xacml:3.0:attribute-category:resource"
       DataType="http://www.w3.org/2001/XMLSchema#integer"
       Issuer="0x56a58e23f8a5efc346910ab0bd950c6ae4333252"
       MustBePresent="true" />
  </Match>
• • •
</AllOf></Target></Rule>
• • •
</Policy>
```

```
• • •
contract AC_Test1{
  AMContract1 private am1;
  • • •
  function AC_Test1() {
    am1=AMContract1(0x56a58e23f8a5efc346910ab0bd950c6ae4333252);
    • • •
  }
  • • •
  function evaluate( • • • ) returns(bool) {
    • • •
    • • • (am1.resource-serial() > 50) && • • •
    • • •
}}
```

Fig. 3. Simplified XACML to Solidity parser example.

in such policy the AMs to be queried to retrieve the required attribute values. To this aim, for each MATCHE the resource owner chooses the SMART AM to be called by specifying in the `<AttributeDesignator />` tag the SMART AM function name through the `AttributeId` field and the SMART AM address through the `Issuer` field.

Finally, the `evaluate` function needs to return the result of the decision process to the CH_O. One solution would be to simply save the result as data in the state of the contract, but, instead, we opted for firing an event containing the request id (i.e., the id of the transaction encoding the access request) paired with the corresponding result. Events are data saved on the EVM log instead of the contract storage space, exploiting them to return the result of the evaluation function is a cheaper way of storing the decision for every request, at the expense of making such decisions invisible ad so unusable to the contracts. In our current system this limitation is not a problem, but this approach could of course be changed if needed.

Once the PAP has finished creating the contract it sends it to the CH_O to be compiled and deployed on the blockchain, and waits for an answer from the CH_O. If the contract deployment phase from the CH_O (see later) is successful, the CH_O communicates to the PAP the address of the newly deployed contract. The PAP then stores in the SPT a new entry consisting of the resource ID and the received address.

C. CH

The off-chain side of the CH (i.e., CH_O) is a component written in Java, and it has the task of managing the access to the blockchain on behalf of the PAP and PEP. At policy creation time, when the CH_O receives from the PAP a SMART POLICY written in Solidity it compiles the Solidity code to EVM bytecode [10] using the `solc` compiler [16]. The CH_O then uses `web3j` to wrap it into a transaction for the deployment on Ethereum through the `geth` node. At this stage the CH_O can optionally perform additional checks on the contract deployment transaction. For example, it could query the blockchain (using the `geth` node) to check whether the policy creator has enough credit (i.e., ether) in its account to pay for the expected gas cost, or it could check whether the SMART AMs invoked in the SMART POLICY do actually exist on the chain.

An important consideration is that the contract deployment transaction needs to be signed by the user who is paying for it. In particular, once the transaction is ready to be signed, it is made visible to the resource owner who can check it (possibly with an automatic tool), sign it, and then communicate the signed transaction back. In our implementation the user interacts with the system through an interactive interface where it is first required to insert the policy it wants to add, then it is informed of the derived smart contract and relative deployment transaction waiting to be signed. The user can then check that the transaction correctly represents the policy it intended and, if it is satisfied, communicate the signed version of the transaction. Once the CH_O receives the signed transaction it checks the signature, and if it is correct it sends it to the `geth` node to broadcast it to the network. The user receives a confirmation or error message depending if the deployment was successful or not. This approach guarantees that the private information about the users wallets are not disclosed to our framework. Once the transaction is actually inserted by a miner in a block, i.e., the SMART POLICY is on the blockchain, the CH_O receives back from `geth` the contract address, which is returned to the PAP to be stored in the SPT.

At access request time, the CH_O receives an XACML access request from the PEP. The CH_O parses such request to retrieve the resource ID and the other attribute values, and it queries the PAP for retrieving the address of the SMART POLICY paired with that resource. Then, the CH_O wraps the request into a transaction encoding a call to the `evaluate` function of the SMART POLICY referenced by the address retrieved from the PAP. This transaction needs to be signed by the subject that submitted the access request (as explained in Section III-B). This step exploits the same signing process detailed above but with the difference that the accessing subject, and not the resource owner, is the one in charge of checking and signing the transaction. Once the transaction is signed, it is passed by the CH_O to `geth`, that will broadcast it on the Ethereum network. Once the transaction is mined the `evaluate` function fires an event with the evaluation result. This event is read by `geth` that passes it to the CH_O which, in turn, communicates the result to the PEP.

D. PDP & PIPs

The traditional tasks of PDP and PIPs are merged together into the SMART POLICY. This contract is dynamically generated by the PAP from an XACML policy and, once deployed, resides on the Ethereum blockchain in EVM bytecode. The decision process of the PDP is performed by the decentralized execution of the `evaluate` function of the contract and the attribute value retrieval is performed by internal calls of the contract directly to SMART AMs on the same chain. All the communication is achieved through smart contract function calls and event firing that are implicitly managed by the Ethereum protocol, so we can say that the SMART POLICY also performs some CH tasks (i.e., CH_B).

V. Discussion and Experiments

The main advantage of our proposal is that the policy evaluation process is performed by smart contracts on a blockchain. This allows our decision system to inherit the blockchain technology advantages, i.e., always available, distributed (so no single point of failure or attack), tamper resistance, etc. An interesting property is auditability, which is derived from the immutability and transparency properties of blockchain technology. Since the smart contract execution is performed by the blockchain (i.e., replicated among the miners), it is beyond the control of both the resource owner and the subject making the request. So neither of them can forge a false decision. Moreover both the policy and the Permit or Deny evaluation result linked to the request identifier are stored on the blockchain as well, so they are both publicly visible. Any user whose access is fraudulently denied by the resource owner can prove that the access right should have been granted instead through the public data on the blockchain. Since the blockchain is immutable this also holds for old SMART POLICIES. Even if a SMART POLICY has been revoked, its code and entire access request log remain still accessible on the blockchain.

Do note that in our reference implementation the resource owner could still fraudulently ignore access requests before any trace of them is left on the blockchain, by simply modifying the PEP to discard some of them. This issue could be avoided by having the resource owner advertise publicly the address of the SMART POLICIES allowing the subjects to directly send transactions to such contracts to obtain the access to the resources.

A clear consequence of auditability is a potential privacy issue. Some solutions to properly mask the publicly available information stored on the blockchain might be needed in a real world application.

One disadvantage of our proposal can be that blockchain is still a novel technology, and so most users have never used such technology, nor they have familiarity with the basic concepts of it. Although the proposed system automatically interacts with the underlying blockchain, users still need a basic understanding of the protocol, since they are required to own a wallet to sign and pay for transactions. Even if transaction checking and signing (described in Section IV-C) as well as auditability checks described in the previous section are operations that can be performed automatically by a secure third party program, this would still require the user to trust such software. Hence the problem of trust would just be moved and not solved. The lack of user familiarity may be an issue for initial widespread adoption.

Another main concern about blockchain technology is performances. The need for a distributed consensus introduces an overhead non-existent in a centralised model where the system state updates are managed by a single (trusted) entity. Moreover, the replication of the shared state and replication of new data validation across all the nodes determines an additional burden for the participants. Because of these observations if we want to base our proposed system on a public blockchain we expect to incur in an inevitable loss of performances compared to a traditional centralised system, i.e., an increase in resource needed to run the Access Control system and an increase in the evaluation time to obtain an access decision. We also remark that, usually, public blockchain protocols introduce a cost per transaction (i.e., fees and gas).

To investigate this aspect, we evaluated the performance of our reference implementation. We analyzed the performance with three measurements: *time*, *resources* and *monetary cost*. We do note that even the centralised side of our reference implementation is different from a traditional system, mainly for the need of a XACML to Solidity parser. Nevertheless the parser complexity is guaranteed linear in the size of the policy (and easily parallelizable in case of policy sets).

1) Time: The time overhead introduced by on-chain operations of our implementation for the policy creation is caused by the SMART POLICY deployment phase, while at access request time, it is instead caused by the SMART POLICY execution. We do not consider the time needed for the user to check and sign a transaction since it is completely user dependent (and can be very short in comparison if automated). Both the SMART POLICY deployment and execution times mainly consist in the time elapsed for the corresponding transaction to be mined into a block. So, both operations times are roughly equal on average to the transaction confirmation time, that depends on the underlying blockchain chosen. In our case, the Ethereum blockchain is designed to add a new block on average every 14 seconds. This means that, as long as there is enough free space in new blocks, a new transaction will take at most 14 seconds on average. In practice this is complicated by transaction propagation times (that might become comparable to mining time for poorly connected nodes). In case of transaction congestion in blocks the transaction can actually take longer to be included in a block. But this time can be manipulated by our system as well as by the other blockchain users by choosing a more competitive gas price (i.e., choosing to pay more for faster confirmation). Due to this possible manipulation of confirmation time by the users there is no simple way to determine how many blocks on average a new transaction takes to be confirmed. We performed the corresponding measurements for our implementation, but our testnet was never congested and so transactions were almost always added on the first available block. This unsurprisingly proved our assumption correct without giving us any insight on a more general case. The main chain is currently (Ethereum block 4 853 654) experiencing a more relevant congestion problem, with almost all the latest blocks between 90 and 99% full. This results in an higher confirmation time, with a median over the last 1500 blocks of 5 blocks wait for transactions to be mined [17].

2) Resources: In our framework, the main need for computational resources is for running the blockchain client. Our reference implementation is based on the Ethereum client `geth`, that nowadays runs fine on standard hardware (i.e., two or more CPU cores, 4 or more GB of RAM memory, and a good network connection) but it requires some storage space (currently more than 255 GB for storing the main Ethereum blockchain). However, since we delegate the storage of policies to the blockchain, we save the storage space required by traditional PAP implementations, although the size of this space is not comparable to the space needed to save the entire blockchain. The storage space requirement can be an issue in some scenarios. To solve this issue two different solutions are possible. The first solution relays on a third party that will manage the blockchain side of the client. Of course this introduces a new cost in the system as well as a point of centralization that needs to be trusted. The second solution, instead, is to use a light client to interact with the blockchain. This would result in a reduction of the storage requirements from hundreds of GB of memory to a few GB, at the expense of potential trust requirement in other full nodes (depending on how the light node is actually implemented). Currently `geth` provides a 'light node mode' but it is still in beta version. Choosing a light client based implementation would allow to deploy our system on most of the nowadays common machines.

3) Monetary Cost: Using fee (or gas) based blockchain technology introduces a monetary cost for every transaction that is mined. Our reference implementation is based on the Ethereum protocol, in which gas is spent by transactions. This cost can be thought as the one charged by a third party service to use its services. To estimate such a cost we evaluated the gas cost of our two kinds of transactions: SMART POLICY deployment transactions and SMART POLICY evaluation execution transactions.

SMART POLICY *deployment transactions:* The gas cost of a transaction deploying a new contract (Gas_D) can be expressed as follows:

$$Gas_D = FixedCost_D + CodeCost + InitCost$$

Where $FixedCost_D$ represents the fixed cost of a contract deployment, $CodeCost$ represents the cost to store the actual data of the contract (i.e., the code) on the blockchain and $InitCost$ represents the computational cost incurred to run the constructor instructions and initialize the contract. In our implementation each contract representing a policy has a fixed core of utility methods and variables that contribute as a constant amount to both $CodeCost$ and $InitCost$. For instance, to save the address of the contract creator in the constructor we require a store operation (which currently costs 20 000 gas) that contributes to $InitCost$, while adding functions to revoke the SMART POLICY increases $CodeCost$ because it increases the code length. The policy dependent contributions to $CodeCost$ and $InitCost$ are mainly due to the number of rules, their complexity and the number of different SMART AMs they require.

Obviously, more rules and more complex rules require more code to be stored and managed. Since each rule consists of a set of MATCHEs, we measure the complexity of a policy as a function of the number of MATCHEs and of their complexities. Less obvious is the contribution of the number of SMART AMs. This is due to the fact that the SMART POLICY stores the addresses of the SMART AMs needed to retrieve required attributes. These addresses are known at deployment time and are saved in contract variables by the constructor, so each address to be remembered causes a costly store operation. Do note that the internal complexity of the SMART AMs does not influence the deployment cost of a contract requiring them (but it will influence its execution cost). In our test we experienced that to deploy an empty policy (i.e., always Permit), we consumed about 175 000 gas, while to deploy a policy with one simple rule performing the comparison of the value of one attribute with a constant (i.e., invoking one SMART AM) we consumed about 280 000 gas. Adding each additional SMART AM consumes approximately 26 000 gas and each additional simple MATCHE in the policy consumes about 46 000 gas (but complex MATCHEs may consume more gas). These are very rough estimations, and should only be considered as a lower bound of the actual cost. Knowing our current gas limit of about 4 700 000 for each block, it is possible to estimate the maximum size (i.e. number of different SMART AMs and MATCHEs) of a deployable policy. According to our optimistic estimation, for example, a policy using 10 different SMART AMs and 90 MATCHEs would be about the maximum size that could fit in one block. We do remark that this is just a constraint of our reference implementation and not of the proposal itself.

SMART POLICY *evaluation transaction:* To evaluate a SMART POLICY we use a transaction calling the `evaluate` function of the contract (See Section IV). The gas cost of such transaction (Gas_E) can be expresses as follows:

$$Gas_E = FixedCost_E + EvalFunctionCost$$

Where $FixedCost_E$ represents the fixed cost of the transaction performing the call (and carrying the function parameters), and $EvalFunctionCost$ represents the cost to execute the `evaluate` function. As explained in Section IV-B, the `evaluate` function is a disjunction or conjunction of boolean functions representing a MATCHE each. Furthermore each encoded MATCHE usually invokes one (or more) SMART AM to retrieve attribute values. This means that the cumulative evaluation cost depends not only on the number of

MATCHES and their individual complexity, but also on the complexity of the SMART AMS invoked. The gas cost estimation is further complicated by the use of short circuiting logical operations. Hence, the execution of the same expression could have very different costs depending on the actual values at execution time. To test this we performed some experiments where we evaluated a policy using three different SMART AMS and eighty MATCHES in conjunction. The first time we purposely choose attribute values to satisfy all the MATCHES, obtaining a Permit result consuming 210 643 gas for the relative transaction. Instead, setting the attribute value of the first MATCHE to fail the condition, the entire expression short circuits to false without evaluating all the remaining MATCHES, resulting in a Deny decision consuming 32 267 gas only for the relative transaction. The second execution consumed approximately 15% of the amount of gas consumed by the first execution. Moreover two thirds of this cost where due to the fixed cost of the transaction itself (more than 20 000 gas), considering $EvalFunctionCost$ alone for both the second execution cost about 5.6% compared to the first.

Due to all this it is difficult to get a general estimation of the cost of the evaluation function. To give an estimation of the gas cost of an evaluation we deployed the SMART POLICY of a simple policy of 90 MATCHES referencing ten SMART AMS. The original policy was a conjunction of simple boolean conditions that we knew being true with the values returned by the SMART AMS chosen (decided to avoid short circuiting). The resulting cost of an evaluation transaction (returning a Permit) in this controlled environment is of approximately 230 000 gas. This shows how the evaluation cost (in the worst case that all MATCHES need to be executed) is a lot lower than the initial deployment cost. For example, for the policy of the above example the evaluation transaction cost about 5% of the gas consumed by the corresponding deployment transaction. Of course this is just a rough estimation, it is very easy to change such estimations by using a very costly SMART AM to arbitrary increase the evaluation cost without influencing the initial SMART POLICY deployment cost (that is independent of it). Given a reference policy invoking a single SMART AM and built by a conjunction of n simple single attribute MATCHES crafted to be always true over the SMART AM returned values, we depicted in Figure 4 the deployment and execution costs for increasing values of n.

VI. CONCLUSIONS

This paper presents an approach to integrate a traditional Access Control system with blockchain technology.

By leveraging blockchain technology properties, our approach enables the subjects requesting access to the resources to verify that the policy has been correctly evaluated, disclosing malicious or faulty third parties fraudulently denying access to subjects. We have presented a proof of concept implementation, which exploits Ethereum to map a subset of the basic functionalities of the reference XACML architecture to smart contracts executed on the blockchain.

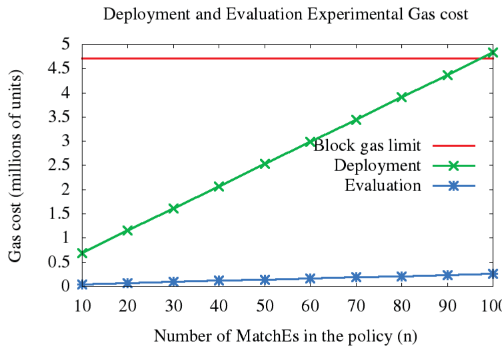


Fig. 4. Gas cost of deployment and evaluation of the reference policy with increasing number of MATCHES n. The last point (n=100) is obtained by artificially increasing the block gas limit.

We plan to investigate alternative mappings of the XACML reference architecture logical modules to blockchain and off-chain functionalities. We will also address the potential privacy issues of the current implementation. Finally, we also plan to extend our approach in order to deal withthe Usage Control model [2], which is an extension of the Access Control one for mutable attribute management.

REFERENCES

[1] Sandhu, R.S., Samarati, P.: Access control: principle and practice. Communications Magazine, IEEE **32**(9) (1994) 40–48
[2] Lazouski, A., Martinelli, F., Mori, P.: A prototype for enforcing usage control policies based on XACML. In: Trust, Privacy and Security in Digital Business. TrustBus 2012. Lecture Notes in Computer Science, vol 7449, Springer-Verlag Berlin Heidelberg (2012) 79–92
[3] Vincent C. Hu, David, F., Rick, K., Adam, S., Sandlin, K. Robert, M., Karen, S.: Guide to attribute based access control (ABAC) definition and considerations (2014)
[4] Lang, U.: Openpmf scaas: Authorization as a service for cloud & SOA applications. In: Second International Conference on Cloud Computing (CloudCom 2010). (2010) 634–643
[5] R. Wu, X. Zhang, G.J.A.H.S., Xie, H.: ACaaS: Access control as a service for iaas cloud. In: International Conference on Social Computing. (2013)
[6] OASIS: eXtensible Access Control Markup Language (XACML) version 3.0 (January 2013)
[7] Zyskind, G., Nathan, O., et al.: Decentralizing privacy: Using blockchain to protect personal data. In: Security and Privacy Workshops (SPW), 2015 IEEE, IEEE (2015) 180–184
[8] Ouaddah, A., Abou Elkalam, A., Ait Ouahman, A.: Fairaccess: a new blockchain-based access control framework for the internet of things. Security and Communication Networks **9**(18) (2016) 5943–5964
[9] Es-Samaali, H., Outchakoucht, A., Leroy, J.: A blockchain-based access control for big data. International Journal of Computer Networks and Communications Security **5**(7) (2017) 137–147
[10] Wood, G.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper (2014)
[11] Di Francesco Maesa, D., Mori, P., Ricci, L.: Blockchain based access control. In: IFIP International Conference on Distributed Applications and Interoperable Systems, Springer (2017) 206–220
[12] Solidity documentation. https://solidity.readthedocs.io/en/develop/
[13] Open Blockchain initiative. https://github.com/ethereum/solidity/releases
[14] geth client. https://github.com/ethereum/go-ethereum/wiki/geth
[15] Svenson, C.: Blockchain: Using cryptocurrency with java. Java Magazine, January/February (2017) 36–46
[16] Solidity compiler releases. https://github.com/ethereum/solidity/releases
[17] Ethereum gas station. https://ethgasstation.info