A blockchain based approach for the definition of auditable Access Control systems

Damiano Di Francesco Maesa[a,b,*], Paolo Mori[b], Laura Ricci[c]

[a]*Department of Computer Science and Technology*
University of Cambridge, UK
[b]*Istituto di Informatica e Telematica*
Consiglio Nazionale delle Ricerche, Pisa, Italy
[c]*Department of Computer Science*
University of Pisa, Italy

Abstract

This work proposes to exploit blockchain technology to define Access Control systems that guarantee the auditability of access control policies evaluation. The key idea of our proposal is to codify attribute-based Access Control policies as smart contracts and deploy them on a blockchain, hence transforming the policy evaluation process into a completely distributed smart contract execution. Not only the policies, but also the attributes required for their evaluation are managed by smart contracts deployed on the blockchain. The auditability property derives from the immutability and transparency properties of blockchain technology. This paper not only presents the proposed Access Control system in general, but also its application to the innovative reference scenario where the resources to be protected are themselves smart contracts. To prove the feasibility of our approach, we present a reference implementation exploiting XACML policies and Solidity written smart contracts deployed on the Ethereum blockchain. Finally, we evaluate the system performances through a set of experimental results, and we discuss the advantages and drawbacks of our proposal.

Keywords: Blockchain, Smart Contract, Ethereum, Access Control, XACML

1. Introduction

In the last years, the most part of existing digital resources (e.g., servers, data bases, services or even smart objects, from smart watches to last generation cars) has been connected to the Internet. This is due to the ever increasing coverage of Internet connectivity. It is obvious that, if on the one hand this connectivity enables the provision of new and better features, on the other hand, it introduces new security risks of unauthorized accesses

*Corresponding author
Email addresses: dd534@cam.ac.uk (Damiano Di Francesco Maesa), paolo.mori@iit.cnr.it (Paolo Mori), laura.ricci@unipi.it (Laura Ricci)

to such resources. In order to prevent privacy violations and erroneous or malicious uses, such resources need to be protected by proper security mechanisms, such as Access Control systems. These systems control the access attempts in order to grant the access only to those subjects actually holding the corresponding rights in the specific access context.

A solution for resource owners is to deploy and run their own Access Control systems on their premises. In this case, resource owners have the responsibility to choose and deploy proper Access Control systems for their resources. However, this solution causes resource owners a burden due to the configuration, deployment and management of the system. Furthermore, resource owners should bear a relevant cost, both in terms of hardware (the servers/VMs which host the Access Control system), software (the Access Control system product could require to pay a periodical fee), and man power (the time spent by the administrators for the Access Control system management). Hence, an alternative solution is to outsource the Access Control functionality to external systems provided (e.g., as services) by trusted third parties. For example, in the last years, some Access Control systems implemented as Cloud services following the Software as a Service (SaaS) paradigm have been proposed, such as OpenPMF SCaaS [1], ACaaS [2] and others. These services often use an open-platform API, in a way such that their users are not bounded to use a specific implementation, but they can exploit them to have a uniform management of policy enforcement for all the resources they own. Similarly, Amazon offers to the users of Amazon Web Services (AWS)[1] an Access Control Service called Identity and Access Management (IAM)[2]. IAM allows Amazon users to manage the access to their AWS services and resources by creating and managing their users and groups and by defining proper permissions to allow or deny the access to their resources.

In this work, we propose an alternative solution for resource owners to outsource the Access Control functionality. In fact, this work presents the design, implementation and validation of a general attribute based Access Control system built on top of blockchain technology. The proposed system follows the reference architecture defined by the XACML standard [3], and its innovative feature is that most of the architecture components are implemented as smart contracts which are deployed, stored and executed on a (smart contract based) blockchain. The key idea behind our approach is to codify Access Control policies as executable smart contracts and to manage them through the blockchain, hence obtaining decentralised self evaluating policies.

In this paper, we provide an in depth analysis of the application of our proposal to a specific innovative scenario, i.e., access regulation to smart contracts. Having the resource to be protected (i.e., a smart contract) and the system to protect it both deployed on the same blockchain would ease many of the issues that may arise by employing our proposal in other application scenarios, as further discussed in rest of this paper. Our approach presents some relevant advantages with respect to outsourcing the Access Control process to third parties. For instance, both resource owners and subjects issuing access requests can easily detect unduly authorizations or denials of access, thanks to publicly auditable proofs

[1] https://aws.amazon.com/

[2] https://aws.amazon.com/iam/

of misbehaviour. Indeed, a third party Access Control service could maliciously force the system to return deny, thus forbidding the access to a subject although the policy would have granted it. Viceversa, it could return permit even when the policy is not satisfied, thus granting the access to a subject not holding the corresponding right. In both previous cases, neither the resource owners nor the subjects could check a posteriori the correctness of the result of the access decision process, because the access context details that have been taken into account might be no longer available. Instead, in our blockchain scenario, both resource owners and subjects are enabled to verify how the policy has been evaluated for each access request that has been performed thanks to the properties of transparency and auditability inherited from blockchain technology. In fact, users can always browse the blockchain and control the access requests that have been performed to a given resource, the values of the attributes at that time, and the resulting access decisions returned as response to their access requests.

The idea of exploiting blockchain technology for Access Control has been preliminary presented in [4, 5], where we extended the Bitcoin protocol to represent, store, and retrieve the policies stating users' rights on resources, as well as to allow the transfer of such rights among users, while the policy evaluation process was executed by a traditional Access Control system. In [6], instead, we enhanced the approach proposed in [4] by exploiting smart contract capabilities to move the main functionality of an Access Control system, the policy evaluation process, on the blockchain as well. We validated our proposal presenting a preliminary implementation based on the XACML standard and the Ethereum protocol [7], due to its native support for smart contracts.

This paper refines and extends the approach proposed in [6] in several directions:

- The approach has been generalized, in order to define a general blockchain based architecture. The description of how the components of the Access Control system are implemented exploiting the blockchain technology has been considerably expanded;
- A new application scenario, where the resources to be protected are smart contracts as well, has been introduced as reference example. We describe in details how the proposed approach is applied in such a scenario, providing a reference implementation;
- The experiment section has been considerably expanded, mainly to better measure time performances. To this aim, a large set of completely new experiments is presented. To strengthen the results, the experiments have been performed not only on an Academic testnet, but also on a more realistic environment, a real world official Ethereum testnet, called Ropsten [8], de facto doubling our experimental environment.

This work is structured as follows. We first provide the reader the needed background in both blockchain technology and smart contracts, as well as Access Control systems, focusing particularly on the XACML standard in Section 2. We then present in Section 3 an application scenario that we use as reference example for validating our proposal. Section 4 presents the main concepts of the general idea behind our proposal to implement an Access Control system based on blockchain technology. In Section 5, we then explain in detail how

the proposed Access Control system can be adopted in the reference example introduced in Section 3, presenting also a working reference implementation. We provide a number of considerations about our proposal in Section 6, while we present a large set of experimental results in Section 7. Finally, Section 8 presents our conclusions and future work.

2. Background and Related Work

2.1. Access Control systems

An Access Control system (ACS) is in charge to protect the resources of an application scenario by checking the access requests performed by the subjects of that scenario in a given access context. In other words, the ACS decides whether these subjects have the rights to perform the accesses they request in the current access context or those accesses must be denied. These rights are expressed by means of Access Control policies, which consist of a set of conditions that are evaluated against the current access context to make the access decision each time an access request is received [9]. In some scenarios, the right of performing the access is not static, and hence it is continuously verified for the whole duration of the access itself, in order to interrupt it in case this right expires because of a change of the access context [10]. Several models have been presented in the scientific literature to find a different way of defining access rights, e.g., Mandatory Access Control (MAC), Discretionary Access Control (DAC), Role Based Access Control [11], and many others. Among them, the Attribute-based Access Control (ABAC) model [12] represents the access context through a set of attributes describing the relevant features of the subjects, resources and environment, and uses Access Control policies consisting of a set of conditions over the values of these attributes. Examples of attributes of the subject S could be, for instance: the ID of S, the ID of the company S works for, the role of S in this company, the name of the projects assigned to S, the number of resources S is currently using, and so on. Example of attributes paired with a piece of data D could be: the project D belongs to, the privacy level assigned to such data (e.g., *public*, *internal* or *confidential*), the ID of the producer of D, and so on. A very simple example of ABAC policy could be the following: a subject S is allowed to access the document D if D belongs to one of the projects assigned to S. This policy simply compares the value of the attribute representing the projects assigned to the subject with the value of the attribute representing the project of the document.

Several languages are currently available for writing ABAC policies, being the eXtensible Access Control Markup Language (XACML) described in the next section one of the most popular.

2.2. XACML Standard

This section gives a very brief description of the eXtensible Access Control Markup Language (XACML) standard defined by the OASIS consortium [3] for expressing ABAC policies, and of the related reference architecture.

XACML is a standard defining an XML based language to express ABAC policies, requests, and responses. Requests are used to express the attribute values that have to be provided by the subject to represent the access context, in the same format as policies. For

example, the subject could provide his `subject-id`, the `resource-id` of the resource he wants to access, and the `action-id` of the action he wants to perform on that resource. Responses, instead, contain the decision relevant to a previous request expressed in a fixed format. A response can be `Permit`, `Deny`, `Indeterminate` (in case of errors or missing values) or `Not Applicable` (the request does not regard any of the policies). The access decision process is based on *policies* and *policy sets*. A policy set is a collection of policies or other policy sets. Each policy, instead, contains a *target* and a set of *rules*. The *target* is a set of simplified conditions concerning the Subject, Resource and Action that must be met for the policy to apply to the request. Each rule expresses an internal *target*, which concerns this rule only, and a *condition* that represents a boolean function over an arbitrary complex combination of functions over a set of attributes. At request time, the target and the rules are evaluated exploiting the current attribute values to return a decision. Since policy sets may contain multiple policies each returning an access decision and policies themselves may contain multiple rules each returning possibly different results, all these decisions need to be combined properly to obtain the final result. This is achieved through *combining algorithms*, either at policy set level (i.e. *policy combining algorithms*) or at policy level (i.e. *rule combining algorithms*). Each algorithm defines the way to properly merge the different individual evaluation results to produce a unique authorization decision. For example, one such algorithm is the *Permit Overrides Algorithm*. It states that the final result is `Permit` if at least one component (either a rule or a policy) returned `Permit`. Few combining algorithms are available as standards in XACML (see for example Appendix C of [3]), but more can be custom defined as needed.

XACML does not only provide a standard to express policies and requests/responses, it also gives a standard for the evaluation architecture. The architecture scheme is shown in Figure 1, and includes the following components.

- **Policy Enforcement Point (PEP)**

  The Policy Enforcement Point is the component paired with the resource to be protected which is able to intercept and suspend the access requests, in order to perform the policy evaluation. In particular, the PEP collects the access requests and a set of available attributes, it triggers the decision process, and it enforces the related result by actually allowing or denying the execution of the access.

- **Policy Administration Point (PAP)**

  The Policy Administration Point is the component in charge of managing Access Control policies. Its main functionality is to act as Policy Repository for storing policies, in order to retrieve them when necessary for evaluating access requests. Another relevant functionality of the PAP concerns policy authoring, thus helping its users (i.e., policy makers) in creating and modifying policies. The PAP may also support more complex functions concerning policy production and management.

- **Attribute Managers (AMs)**

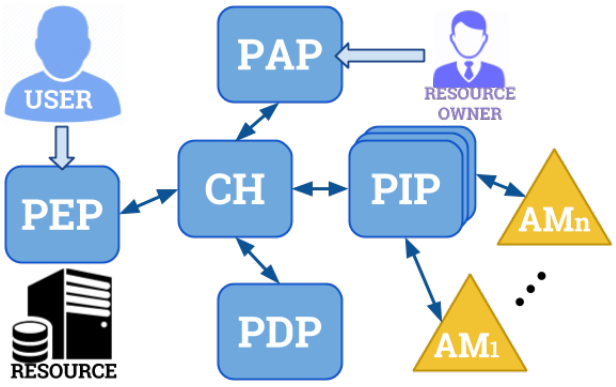

Figure 1: XACML reference architecture.

Attribute Managers are the components that actually manage the attributes of subjects, resources, and environment, allowing the Access Control system to retrieve and update their values. Each application scenario has its specific AMs, depending on the attributes which are relevant for that scenario. AMs could be part of the authorization service itself, they could be run in other machines in the same domain, in other administrative domains, or even by third parties. Existing services can be exploited as AMs.

- **Policy Information Points (PIPs)**

  The set of attributes required for the policy evaluation are, in the most general case, managed by a set of distinct Attribute Managers, each having its own protocol to be used for collecting the current attribute values. Policy Information Points act as plugins of the Access Control system, providing the interfaces for interacting with each Attribute Manager, thus allowing to retrieve the latest values of attributes and to update them.

- **Policy Decision Point (PDP)**

  The Policy Decision Point is the evaluation engine that takes a policy, an access request, and the current attribute values as input, evaluates the policy and returns the related access decision (i.e., `Permit`, `Deny`, `Indeterminate` or `Not Applicable`).

- **Context Handler (CH)**

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

2.3. Blockchain Technology and Ethereum

Blockchain technology allows to build an immutable, distributed, always available, secure and publicly accessible repository of data (ledgers). It relies on a distributed consensus protocol to manage this repository (e.g., to decide what valid new data to add) in a distributed manner [13]. The different types of blockchains basically differ for the trust level associated to *write* and *read* operations. By write operation we mean the ability to update the ledger, i.e., write a new content on it, while by read operation we mean the ability to read the existing contents. Blockchains are called public (resp. private) whether any trustless (resp. only trusted) entities can read. They are called permissionless (resp. permissioned) whether any trustless (resp. only trusted) entities can write [14]. For an example of a permissioned blockchain see [15].

Historically, blockchain technology was first introduced to support cryptocurrencies [16]. In such a scenario the blockchain is used as a public ledger to store transactions transferring value between entities. The first blockchain was used by the Bitcoin cryptocurrency protocol [17], but since then several interesting new proposals have been presented [18]. In this paper we use one of such proposals, i.e., Ethereum [19]. In this section we try to give a basic understanding of the protocol to the reader through a very high level explanation, omitting most of the implementation details not relevant to this paper. For a more detailed explanation see [7].

Ethereum starts from a simple assumption, coupling a Bitcoin style cryptocurrency (see [4]) with Turing complete applications. The protocol can still be seen as a cryptocurrency due to the *ether* currency on which it is based. The way in which value expressed in Ether is managed is in principle not too different from how Bitcoin manages value. The system is based on pseudonymous entities (masked by addresses [20]) that exchange value through special data structures, called transactions, that are broadcast to the underlying communication network. Those transactions are eventually validated and will then apply their effect by updating the global state (recording the passage of value to the new owners). Since the reference scenario is a trustless decentralized system, a distributed consensus algorithm is needed to reach an agreement on which new transaction to deem valid. Same as Bitcoin, the distributed consensus algorithm currently used is based on PoW (Proof-of-Work) and it is called *Ethash* [21], even if there are proposals to move to different consensus algorithm schemes. The pending transactions (i.e., transactions waiting validation) are grouped together by validator nodes, called *miners*, into data structures called blocks. One of the miners will eventually win the PoW consensus race and its block will be securely added to the globally accepted chain of blocks (this block and the transactions contained in it are said to have been *mined*). This chain of blocks is what determines the global state, and each new block addition is an update to such state.

The novel contribution of the Ethereum protocol is to use the blockchain not only to store value transfers, but also code. Alongside traditional value exchange transactions, the users can also create transactions carrying executable Turing complete code. Those pieces of code deployed on the blockchain are called *smart contracts*. A transaction carrying the payload of the contract is first broadcast to the network. Its result is the deployment of the payload as code linked by its public address. Any new transactions can then refer

to this address to trigger the execution of the functions inside the contract, carrying the function parameters inside the transaction payload and showing the eventual return value to the outside. Basically, a smart contract contains some storage space for its data and some callable functions. Smart contracts can themselves act like regular users creating new contracts and sending payments or function calls between themselves. Validating (i.e., adding it to a block) a transaction containing code (either calling a contract function or deploying a new contract) means to execute such code, updating the global state accordingly. This means that the global state can be seen as a virtual machine running all the code on the blockchain (called Ethereum Virtual Machine or EVM). At the same time all the miners adding a transaction with code to a block actually execute that code, so the code execution is replicated between all the miners. The smart contracts are written in a low level bytecode language interpreted by the EVM. High level languages whose programs can be compiled in EVM bytecode (producing a *.bin* file containing the binary of the compiled contract and an *.abi* file containing the contract interface specification) have also been developed to ease human smart contract coding. The most widespread of such languages is a JavaScript style language called Solidity [22].

It is important to remark that every transaction has to pay a fee proportional to its complexity to repay the miners of their effort of maintaining the EVM. To every single operation of the EVM is assigned (by the protocol) a price proportional to its burden to the users (i.e., the number of computational steps needed for its execution and its storage weight), this is called *gas* and the total gas of a transaction is the summation of all the gas of every single instruction it contains. This is the gas that is consumed by the transaction upon validation. The entity (either a user or a contract) creating the transaction needs to decide two parameters, the *gas limit* and *gas price*. The gas limit is the maximum amount of gas the transaction is allowed to consume, if it is exceeded all gas is spent but the execution effects on the state are eliminated. This is useful to avoid too long or even infinite computations that would stall the EVM. Furthermore each block has associated a *block gas limit* to guarantee a limit to the amount of computation executed by all the transactions in that single block. The gas price is instead set by the user as the amount of ether the user is willing to pay for each unit of gas. Miners are free to choose what transaction to mine and so they can refuse the ones with a gas price too low.

2.4. Related Work

To the best of our knowledge, only a few proposal of blockchain related Access Control systems have been presented. In [23] the authors combine blockchain and off-chain storage to build a personal data management platform focused on privacy. However the control is limited to read operations on users stored data, the work does not address the general problem of Access Control systems. Similarly [24] proposes a method to store secret data (i.e. encrypted) on a blockchain, managed by a set of trustees, that are in charge of controlling the access to such data. The blockchain is used as a tamper proof log of access requests and to guarantee operations atomicity. As such the proposal concerns the problem of Access Control in a data sharing platform, in this paper, instead, we propose to bridge general traditional Access Control systems with blockchain technology. The same holds for [25]

where the blockchain is simply used as Access Control tool for IoT data stored elsewhere. [26] proposes a Role Based Access Control system which uses smart contracts and blockchain technology as infrastructures to represent the trust and endorsement relationship essential to realize a challenge-response authentication protocol that verifies users ownership of roles.

All the remaining works combining blockchain technology and Access Control we know of are applied to one of three very specific fields, either IoT, health care or cloud storage.

In the IoT field, both topics of Access Control [27] and blockchain integration [28] have been separately studied. The first proposal to put them both together has been [29]. This proposal uses access tokens exchangeable through transactions among users. Scripts inside the tokens are used to prove that only the right users can redeem them (i.e. only them hold the corresponding rights). This allows for access right exchanges (through tokens), similarly to the proposal in [4]. The main limitation of such proposal relies on the limited expressive capabilities of scripts. More recent proposals have been [30, 31, 32], that are all based, on some extent, on smart contracts. Main limitation of all these proposals is their particular scope as they concern only a very specific field of application instead of the general case. [33] proposes an authorization and delegation model for the IoT-Cloud based on blockchain technology.

In the field of cloud storage, [34] provides an access control over the data stored in the cloud without the provider participation. [35] proposes a data storage and sharing scheme for decentralized storage systems combining a decentralized storage system, the Ethereum blockchain and the ABE technology.

Regarding health care, the main contributions are focused on protecting the access to patients electronic medical records. One such practical proposal is [36], based on smart contracts implemented on Ethereum. Interestingly it also introduces access to aggregated and anonymized data (usable for example for research) as mining reward to foster participation in the expensive mining process. On the same topic a blockchain based lightweight and robust Access Control framework addressing the security and privacy issues in Big Data is introduced in [37]. Another proposal to manage the access to personal medical records through blockchain is presented in [38].

3. Reference Example

This section presents a novel application scenario for the blockchain based Access Control system proposed in this work that will be used through the paper in order to ease the reader understanding of the proposed system.

In our reference scenario the resources that we want to protect through the proposed Access Control system are smart contracts deployed and executed on the same blockchain. For the sake of clarity, we will call these smart contracts SMART RESOURCES. We do remark that no assumptions on the SMART RESOURCES is made, i.e., they can be contracts of any kind whose execution, for reasons that are immaterial here, needs to be protected through an Access Control system. The proposed Access Control system is independent from the specific operations implemented by the SMART RESOURCE (even if, of course, the policy defined for a specific SMART RESOURCE expresses the access rights on its operations).

Moreover, we remark that in this paper we use the term SMART RESOURCES to stress the fact that the entities to be controlled as *resources* are actually *smart* contracts. We point out that a SMART RESOURCE should not be mistaken for a *smart device*, i.e., a connected device capable of autonomous operations.

An example of Access Control policy for this scenario could control the access to operations offered by a contract managing payments and financial record keeping of a company. For example, possible conditions over attributes of the subjects, resource or environment could be that only recognized accountants could update the company balance sheet, employees can only request a salary payout after they have completed certain tasks assigned to them, and company managers can ask for bonuses only if certain external objectives are met (such as the company stock price raising above a certain threshold).

In this scenario, the adoption of the proposed blockchain based Access Control system has the following advantages:

- relieving the SMART RESOURCE creator from designing and developing the specific code to manage access control. It is worth noting that our approach also relieves the smart contract creator from rewriting the access control logic embedded in his SMART RESOURCE code each time the corresponding access control policy changes. Moreover, our system is also able to automatically embed the access control logic within the SMART RESOURCE with no effort from and independently of the smart contract creator;

- clearly separating the logic for access control from the SMART RESOURCE one. This simplify for both the SMART RESOURCE owner and the access requesting subjects to prove that the Access Control policy actually enforced is really the one declared by the SMART RESOURCE owner. As pointed out in the previous bullet this also simplifies updates in case of policy changes. Since both the SMART RESOURCE and Access Control logic and decisions are both stored on the blockchain, so access requesting subjects can inspect the transactions recorded on the blockchain to verify the reason that lead to any access decision, protecting them form unduly denial of access.

4. Blockchain-based Access Control system

To help the reader, we provide in Table 1 a recap of terms and acronyms defined in the literature or newly introduced, that will be used in the paper.

The basic idea underlying our approach is to exploit a blockchain to store Access Control policies and manage attributes, as well as to execute the access decision process, i.e., to evaluate the relevant policies exploiting the required attributes every time an Access Control request is issued by an user who wants to access a resource. We represent an Access Control policy through a smart contract, called SMART POLICY, which is created by the resource owner and stored on the blockchain by a proper transaction. Since the blockchain is an append only ledger, once uploaded, a SMART POLICY will be stored on the blockchain forever. However, it can be logically replaced by simply uploading on the blockchain a new one, or even disabled by a proper transaction. The execution of the access decision process

TERM	MEANING
PEP	Policy Enforcement Point, see Section 2.2.
PDP	Policy Decision Point, see Section 2.2.
PIP	Policy Information Point, see Section 2.2.
PAP	Policy Administration Point, see Section 2.2.
PTP	Policy Translation Point, module of the PAP in charge of translating an XACML policy into an executable SMART POLICY.
AM	Attribute Manager, see Section 2.2.
CH_E	Module of the Context Handler, see Section 2.2, in charge of starting the evaluation process to evaluate an access request.
CH_D	Module of the Context Handler, see Section 2.2, in charge of deploying a new smart policy into the underlying blockchain.
CH_B	Module of the Context Handler, see Section 2.2, in charge of coordinating the SMART POLICY execution.
SPT	SMART POLICIES Table, used to remember the pairing between SMART POLICY addresses and resource IDs.
PCTRANS	POLICY CREATION TRANSACTION, transaction to deploy a new SMART POLICY.
PUTRANS	POLICY UPDATE TRANSACTION, transaction to update an existing SMART POLICY.
PETRANS	POLICY EVALUATION TRANSACTION, transaction to trigger a SMART POLICY evaluation.
SMART POLICY	Access Control policy expressed into an executable format as smart contract.
SMART AM	Attribute Manager implemented through a smart contract.
SMART RESOURCE	Smart Contract which needs an Access Control mechanism.
MATCHE	<Match>..</Match> element of an XACML policy, used as finest granularity to estimate policy complexity.
evaluate	Main function of a SMART POLICY contract, used to obtain the policy evaluation result.
RIP	module in charge of inlining the PEP code into the SMART RESOURCE.

Table 1: Main acronyms and terms used in this paper.

leverages the blockchain as well. In fact, each time a subject issues an access request, it is issued a proper message on the blockchain to trigger the execution of the SMART POLICY. This message 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 will explain in Section 4.2.3. For the sake of simplicity, we say that "the SMART POLICY evaluation is executed on the blockchain" meaning that the SMART POLICY execution is replicated among the miners elaborating the new block to be added to the blockchain.

In the rest of this section, we describe in details how we represent Access Control policies, how we create and store them on the blockchain, how we revoke or update them, and how we evaluate them by retrieving the required attributes to produce an access decision. Since the proposed system is based on the XACML standard, we also describe how, in the proposed system, the architectural components in charge of the previous tasks, according to the XACML reference architecture, are defined on top of blockchain technology. The resulting architecture is shown in Figure 2.

The proposed system can be easily adopted in the reference application scenario presented

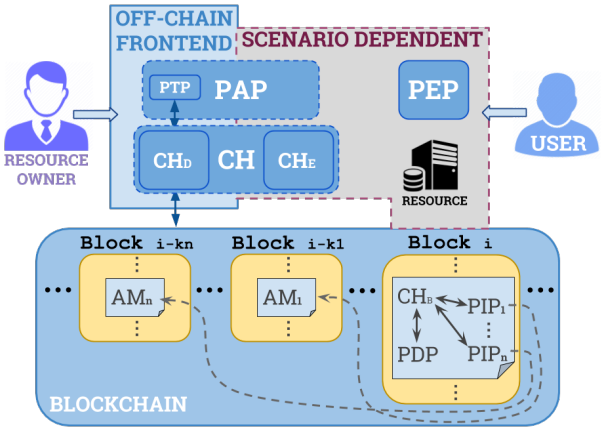

Figure 2: Architecture of the blockchain Based Access Control system.

in Section 3, as detailed in Section 5. Obviously, the kind of resource to be protected influences the structure of the PEP in charge of interacting with it, and of other components of the system as well. For instance, if the resource to be protected is a smart contract, as in our reference example, the PEP will be part of such smart contract. Nevertheless, some components of the architecture do not depend on the application scenario, and they are placed either in the off-chain frontend or in the blockchain environment in Figure 2. The scenario dependent components are instead depicted inside the "scenario dependent" box, as they can be placed either on or off chain (possibly split between both) according to the specific features of the application scenario. Do note that the off chain frontend can be deployed either on the resource to be protected, or on a trusted third party for that resource.

Moreover, we do note that the architecture of our blockchain based Access Control system is, in general, independent from the specific underlying blockchain technology chosen, provided that such blockchain supports smart contracts. However, although the usage of our system does not require technical knowledge of the underlying blockchain technology, the user may be required to provide some further blockchain-related information. For instance, in our implementation based on the Ethereum blockchain, gas needs to be paid (see Section 2.3). This means that users, to use the proposed system, do need to own a wallet and to provide (not private) information about it, as well as being required to perform additional operations (e.g., signing blockchain transactions).

4.1. Smart AMs

In our proposed system, the attributes representing the features of subjects, resources and environment are stored on the blockchain as well, and they are managed by a set of smart contracts. In this way, we exploit the blockchain advantages also for attribute management. In fact, the values of attributes are not alterable, since they are stored on the blockchain, and they are auditable, since their updates can be executed only through blockchain transactions, which are recorded on the blockchain as well. Following the XACML naming, the contracts which store the set of attributes can be seen as the Attribute Managers, hence, we will call them SMART AMs. In our proposal we assume the existence of an ecosystem of SMART AMs

deployed, maintained and advertised by third parties. For example, the smart contract of an institution could offer public information about its employees (e.g., their roles). These SMART AMs could also require some way of payment (either in or off chain) for the use of their services, so a market of SMART AMs could naturally emerge. A SMART AM is invoked by a SMART POLICY to retrieve the current values of the attributes it manages. The parts of the SMART POLICY who invoke 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.

4.2. Smart Policy

The solution we propose in this work 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 expressing conditions over the attributes representing the access context (see Section 2.2), that are combined exploiting proper combining algorithms and that must be satisfied accordingly in order to grant the requested access. For 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 both from academic and business organizations. In our approach, an XACML policy is properly translated into a smart contract, called SMART POLICY, in order to store and execute it on the blockchain. 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 XACML policy and the Policy Information Points (PIPs) required to collect the values of the attributes of that policy.

4.2.1. Smart Policy creation

The SMART POLICY creation process consists of three steps: *i)* XACML policy writing; *ii)* policy translation from XACML to smart contract; and *iii)* deployment of such contract on the blockchain. The policy writing is executed by the resource owner using existing XACML authoring tools. The second operation regards policy management and so it is tasked to the PAP. The SMART POLICY deployment, instead, requires an interaction with the blockchain, and so this functionality is delegated to the CH. In particular, in Figure 2, we denoted as Deployment CH (CH_D) the subcomponent of the CH devoted to the SMART POLICY deployment, achieved through a transaction called POLICY CREATION TRANSACTION (PCTRANS).

The lifecycle of a SMART POLICY starts when the resource owner writes a new XACML policy to define the access rights on his resource(s) and submits it to the PAP. The Policy Translation Point (PTP) is a module of the PAP which translates the logic expressed by the XACML policy in to the SMART POLICY. The SMART POLICY is not a simple rewriting of the XACML policy, but it also contains all the logic (i.e., the executable code) implementing the policy evaluation 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 SMART AM 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 (as shown in Section 4.1). The set of predicates over the attribute values included in a rule are translated in executable code as well, and the SMART POLICY includes the logic for combining the results provided by the rules into a single decision (`Permit` or `Deny`). 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 PIPs, as well as part of the tasks of CH, i.e., the workflow orchestration, represented in Figure 2 by CH_B.

Once translated, the SMART POLICY is deployed on the blockchain by the CH_D, which issues the PCTRANS. The expenses related to the SMART POLICY deployment are paid by the policy creator because the policy creator is the entity that benefits from having an Access Control on the resource.

4.2.2. Smart Policy revocation and update

Resource owners can decide at any moment to revoke their SMART POLICIES. This is achieved in our system by inserting a self destruct function in the SMART POLICIES. The contract enforces the constraint that only the resource owner (i.e., the creator of the contract) is allowed to call this function, by issuing a POLICY UPDATE TRANSACTION (PUTRANS). Since resource owners are the ones issuing the revoke operation, they also pay the price of revoking their policies. Do note that in most blockchain technologies available today, the blockchain is immutable and so the smart contract is not actually removed from the chain, but it is only disabled, thus allowing auditability. In fact, in such cases the SMART POLICY is marked as not callable and so future calls to that contract will fail as expected, but, at the same time, the actual contract remains publicly visible in the chain.

Updating a SMART POLICY means changing the related smart contract. Again, blockchains do not usually allow to change the code of a smart contract. So, updating a smart contract, such as a SMART POLICY, simply means deploying the new smart contract and use its new address instead of the previous one everywhere needed.

4.2.3. Smart Policy evaluation

The SMART POLICY evaluation is executed every time a subject tries to access the protected resource causing the PEP to invoke the Blockchain based Access Control system. To trigger the SMART POLICY execution, the PEP creates the access request and sends it to the Evaluation CH (CH_E) which, in turn, invokes the SMART POLICY.

In some scenarios, the CH_E is external to the blockchain (and the PEP as well). Consequently, the CH_E creates a transaction, called POLICY EVALUATION TRANSACTION (PETRANS), and submits it to the blockchain. In other scenarios, such as the SMART RESOURCE one that will be detailed in Section 5, the PEP and the CH_E are both on the blockchain (i.e., they are embedded in a smart contract as well). In this case, the CH_E sends a message (usually a smart contract function call) to the SMART POLICY exploiting the blockchain communication mechanism. Despite being formally different from a transaction (that can only be created by external accounts), this message has the same tasks and

effect, so the subsequent evaluation process is exactly the same in both scenarios.

The evaluation transaction (or message) triggers the execution of the main method of the SMART POLICY, which coordinates the execution of the policy evaluation process (in the XACML reference architecture this is a task of the CH). In particular, the updated value of all the attributes required for the evaluation of the policy are collected. This phases are executed by the functions of the SMART POLICY which implement the PIPs. Such functions issue a number of messages triggering the execution of the SMART AMs corresponding to the required attributes. The function corresponding to the PDP tailored for the specific policy is then executed with the retrieved attribute values. This produces the decision that will be returned as result of the policy evaluation. Do note that attribute values are retrieved in a lazy way, i.e., they are retrieved only if and when needed for the function evaluation. We choose the lazy solution to minimize the number of external calls executed by the SMART POLICY. In fact, in general, each attribute is retrieved via a call to a different contract (the corresponding SMART AM) and this is an expensive operation (in terms of fees consumed) that is completely useless if the value is not actually used. Lazy attribute values retrieval allows for a cheaper and faster policy evaluation.

The expenses of the evaluation process are paid by the subject making the access 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. It also means that the subject needs to manage a wallet holding value on the underlying blockchain and it is required to interact with the Access Control system (e.g. for signing transactions) to validate the payment of the required expenses.

4.3. Proof of Concept Implementation

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 work for the reference example depicted in Section 3. We present this implementation in Section 5, while in this section we describe the common tools and environments independent of the chosen application scenario.

4.3.1. Deployment and execution environment

To implement our system we have chosen the Ethereum blockchain protocol (as of December 2018), because it is strongly focused on smart contracts and because it is nowadays a widely used smart contract ready blockchain protocol proposal. We then chose Solidity (see Section 2.3) as programming language to write the smart contracts and the Java language to write the off-chain side of our framework.

To allow our Java client to interact with `geth` we use the `web3j` [39] Java library. `web3j` is a lightweight library that supports all of the JSON-RPC API offered by `geth`. It also allows automatic creation of Java smart contract function wrappers from Solidity ABI files (see Section 2.3).

To deploy and test our system, we first used the *International Educational blockchain* academic testnet (part of the *Open Blockchain* initiative [40]). This is a 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. Due to the small size of the community currently using this testnet it provided us with a perfect simulation tool to deploy our system (e.g., we could artificially influence parameters, such as block congestion, as required), but it lacked the randomness and practical issues of a real and widely used network, running several different contracts all the time. This is why we also used the Ethereum official testnet *Ropsten* [41] for our experiments, to obtain results on a global and more realistic testnet. In fact, Ropsten is the official PoW (Proof of Work) Ethereum testnet, revived in March 2017 after a DoS attack [8]. Since it uses PoW, like nowadays Ethereum (despite the proposals to move past it), it better models the real network compared to the other two official testnets, *Rinkeby* [42] and *Kovan* [43], that both use PoA (Proof of Authority) to be protected against attacks like the aforementioned one. To access both testnet blockchains we used `geth` [44], one of the most used Ethereum clients. We did not use the Ethereum main chain because of the cost constraints, and mainly to avoid to burden the immutable Ethereum main chain with our test data intended to be temporary.

4.3.2. Smart Policy Translation

The component tasked with SMART POLICY translation is the PTP. The PTP is written in java and its task is to translate an XACML policy written by the resource owner to a Solidity smart contract.

The main function of the SMART POLICY is called `evaluate` and represents the executable version of the XACML policy. In the following of this section, we show how the XACML policy is translated in Solidity to produce the body of the `evaluate` function. However, the SMART POLICY also contains some utility functions which are the same for all policies (e.g., a self destruct function that is invoked to revoke the SMART POLICY as explained in Section 4.2.2).

An XACML policy consists of a policy Target and a set of rules, each including their Targets and Conditions [3]. 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 of these elements will be called MATCHE in the rest of this work.

Each MATCHE is translated as a check instruction which is embedded within the `evaluate` function. The Solidity function to be used to implement the check instruction is derived from the XACML `MatchId` field and the data type from the XACML `DataType` field of <`AttributeValue`> and <`AttributeDesignator`>. Checks, to be performed, needs the current value of attributes at access request time. Hence, the SMART POLICY must be also able to retrieve these values in order to compute the decision result. In our proposal, we integrate the PIPs functionalities in the SMART POLICY through smart contract messages to SMART AMS (see Section 4.1). Any resource owner who creates a new policy needs to specify in such policy the AMs to be interrogated 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. Each MATCHE

returns a boolean result and these are properly composed with conjunctions or disjunctions as defined by the policy through the tags <AllOf> and <AnyOf> respectively. In Figure 3 it is shown an example of how the components of a XACML policy get translated into the corresponding SMART POLICY contract.

We do remark that this process is in theory applicable to any type of MATCHE, to translate any possible XACML policy into a SMART POLICY. In practice this might be not feasible due to technical limitations of the smart contract programming language adopted or of the underlying blockchain adopted to run them. For example, the Solidity language adopted by our proof of concept implementation has known limitations in dealing with dynamic arrays (e.g. returning string arrays between function calls, see [45]). Our proof of concept implementation aims at proving the theoretical feasibility of our approach without tackling such generalization issues. As such our parser has been designed to translate a number of simple operands and data types (e.g. we did not consider operands over collections of values), which allow to express a number of common ABAC policies. More complex operations among more sophisticated data types may result in more complex checks and, in turn, more expensive smart contracts. In general, as long as the underlying blockchain supports it, any XACML operation and data type can be expressed by designing or using the appropriate libraries, hence preserving the proposal generality. Smart contracts might invoke library functions (if necessary, defined by library smart contracts) same as regular programs. Ad hoc libraries can be written and built for different data types and operations in case of real world adoption of our system. A complete development of such software is beyond the scope of this paper. Moreover, a discussion on how the features of the specific underlying blockchain could constraint, instead, the maximum number of MATCHES that could be included in the policy is given in Section 6.

Finally, the `evaluate` function needs to return the result of the decision process. 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 and 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.

We make now an important remark about the parser. Since XACML derives from XML, the parser is implemented as a classical XML parser. As such it has linear computational complexity. Nevertheless it is too resource intensive to be executed on the Ethereum blockchain. The parser execution would cause an out of gas exception (i.e., the error arising when the execution exceeds the block gas limit) for any but the most trivial policy. Moreover, this holds assuming that the smart contract to execute the parser could itself be deployed without incurring in an out of gas exception. Even if we could deploy and execute the parser as a smart contract, passing it an XACML policy as argument for its execution, would be prohibitively expensive. XACML is a very verbose language while message size is a scarce and priced resource on the blockchain. In conclusion, implementing and executing the parser through a smart contract on the blockchain would not yield any real advantage while re-

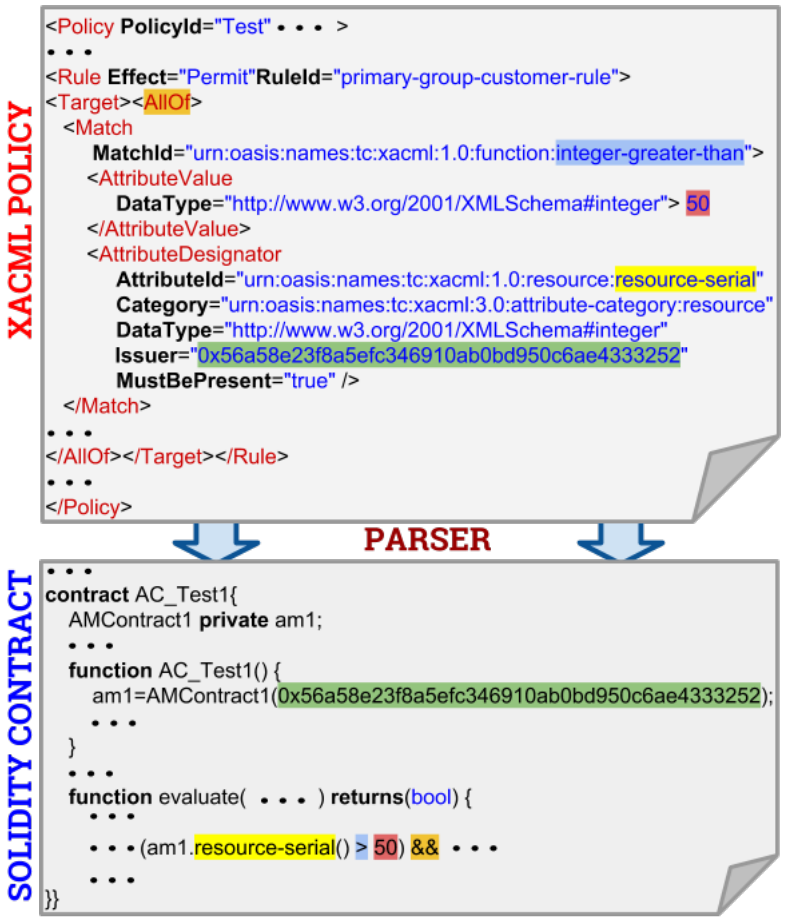

Figure 3: Simplified XACML to Solidity parser example.

maining indubitably cumbersome for the whole blockchain community and expensive for the single user. We will show later (see Section 5.3) that the user can check the parser result (i.e., the SMART POLICY code) before deployment, so any error or fraudulent behaviour by the parser can be detected and thwarted. Moreover, once deployed, any other user given the SMART POLICY code can check whether it represents a certain XACML policy simply by running the parser themselves and checking if the obtained result matches. This way any user can verify if a resource owner is protecting their resource with the policy they claim, by checking the corresponding SMART POLICY code (and it is in the interest of the resource owner to advertise the needed information, if challenged, to prove their have no fraudulent intentions).

5. Controlling the Access to Smart Contracts

This section describes the customization of the proposed blockchain based Access Control system (presented in Section 4) to be adopted in the reference application scenario defined in Section 3. In such scenario the resources to be protected are smart contracts deployed on

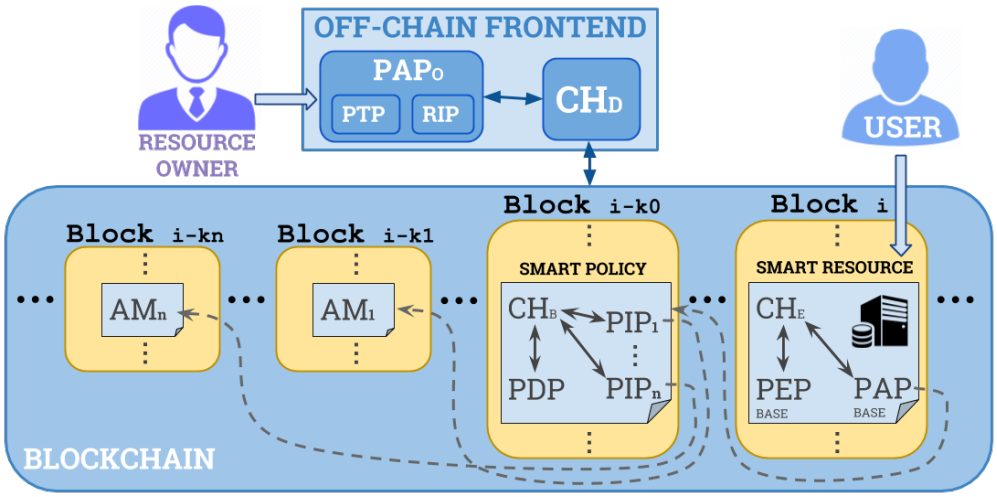

Figure 4: Architecture of the blockchain based Access Control Service customized for the SMART RESOURCE reference scenario. PEP_{BASE} and PAP_{BASE} represent the logic operations of traditional PEP and PAP performed by the SMART RESOURCE.

a blockchain, called SMART RESOURCES, and the owners are the authors of such contracts. The subjects, instead, are blockchain users who invoke a SMART RESOURCE to benefit of the provided functionality. The actions that a subject can perform on a SMART RESOURCE are represented by calls to the functions provided by its smart contract.

In order to protect their SMART RESOURCES, the owners want some *critical functions* of them to be executed only by subjects holding the corresponding rights. Hence, to define such rights, the SMART RESOURCES owner defines a XACML policy set where each of the policies defines the rights to access one of the functions of the SMART RESOURCES. In particular, each of these XACML policies specifies rules and conditions only applicable to a specific function of a specific SMART RESOURCE because the target section of each of these policies includes a predicate which is satisfied when the value of the attribute `resource_id` is equal to the address of the SMART RESOURCE, and the `action_id` attribute is equal to the name of the function the policy refers to. Do note that this does not prevent the resource owner from defining more than one policy applicable to the same `action_id` attribute value, as well as a policy with more than one rule applicable to the same `action_id` attribute value, since it would be a valid XACML policy set. As usual, the XACML policy and rule combining algorithms will solve possible conflicts among these policies and rules. Moreover, the system still accepts any valid XACML policy, so policies and their rules are not required to specify restrictions over the value of the attribute `action_id`. If no restriction are defined over a particular `action_id` value, i.e., over a specific function of the SMART RESOURCE, then the policy/rule will control the access to all the functions of the SMART RESOURCE.

We have developed a proof of concept implementation of the proposed system based on our execution environment (see Section 4.3.1) , i.e., developed on the Ethereum blockchain with smart contracts written in Solidity and off-chain side written in Java bridged over the blockchain by a `geth` node accessed through `web3j`. The resulting architecture is depicted in

Figure 4, and the following of this section describes the proof of concept implementation of each of the components, highlighting whether the component is deployed on the blockchain or on the off-chain frontend.

5.1. PEP

Our blockchain based Access Control system is designed to be easily integrated in already existing scenarios by integrating the PEP in the access interface of the protected resource.

In the reference example scenario, the code implementing the PEP tasks is embedded in the code of the SMART RESOURCE itself. In particular, each of the functions of the SMART RESOURCE that needs to be protected must invoke, as first operation, the proposed Access Control system to perform the evaluation of the Access Control policy, i.e., it must issue a policy evaluation message to trigger the execution of the SMART POLICY related to that SMART RESOURCE (see Section 5.2). Consequently, the address of the SMART POLICY must be directly embedded in the PEP that invokes the evaluation of such policy, thus playing part of the PAP role (i.e., policy retrieval). In other words, we can say that also part of the PAP runs on the blockchain. The decision returned by the SMART POLICY will determine whether the actual code of the SMART RESOURCE function will be executed or not.

5.2. PAP

In the XACML reference architecture, the PAP is the component in charge of policy management and retrieval (see Section 2.2). In the proposed blockchain based Access Control system, its main tasks are to transform the XACML policies written by resource owners into SMART POLICIES, and to manage the mapping between the resources and the related SMART POLICIES to allow their invocation on the blockchain.

In the SMART RESOURCE scenario, the PAP tasks concerning the SMART POLICY creation are performed by a module external to the blockchain (i.e., it runs within the blockchain Access Control system Frontend) called PAP_O (i.e., off-chain PAP). This component is also in charge of inlining into the SMART RESOURCE the PEP code, which actually consists in the creation of the message triggering the execution of the `evaluate` function of the SMART POLICY, through a proper additional module called Resource Inlining Point (RIP). To this aim, the PAP_O requires in input both the SMART RESOURCE contract code and the corresponding XACML policy to be enforced. Then, the PTP produces the SMART POLICY as shown in 4.2.1, and it properly embeds the invocations to such policy in the SMART RESOURCE functions through the RIP. Of course, the SMART POLICY needs to be deployed on the blockchain first, in order to retrieve its address to be embedded in the modified SMART RESOURCE code. We remark that the deployment phase can optionally be carried out directly by the resource owner instead of the CH_D, using our Frontend only as a translation tool, in case such approach is preferable.

Alternatively, the PAP_O also could be implemented as a smart contract and deployed on the blockchain. This would leave no need for the CH_D, and so the entire system would reside on chain. However, we already stated the practical unfeasibility of such approach in Section 4.3.2, mainly due to cost constraints.

In our implementation, the RIP leverages the concept of *function modifiers* provided by the Solidity language to embed the PEP functionality within the SMART RESOURCE. In Solidity a function modifier is used to wrap a function around a given code. By creating modifiers containing checks at their beginning, and halting execution in case of failure through the `require (boolean function)` Solidity construct, we can control a function execution. To this aim, our system provides a fixed base contract called LINKER, that acts like a bridge between the SMART RESOURCE and the SMART POLICY. It contains a constructor requiring the address of the SMART POLICY as parameter to perform the linking between the policy and the resource. It then defines a modifier that is used to enforce the access request evaluation. This modifier simply calls the `evaluate` function of the SMART POLICY and halts the execution if it returns false. The SMART RESOURCE contract needs only to specify that it inherits from LINKER and add the modifier to the signature of every function, not just the *critical* ones.

5.3. CH

The CH_D and CH_E are two logic subcomponents of the CH with the task of managing the access to the blockchain on behalf of, respectively, the PAP and the PEP.

At policy creation time, the CH_D receives from the PAP a SMART POLICY written in Solidity, and it compiles the Solidity code to EVM bytecode (see Section 2.3) using the `solc` compiler [46]. The CH_D then uses `web3j` to wrap it into a transaction for the deployment on the Ethereum blockchain through the `geth` node. At this stage the CH_D 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 his account to pay for the expected gas cost, or it could check whether the SMART AMs invoked by the SMART POLICY do actually exist on the chain.

It is worth noticing 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 exploiting an automatic tool), sign it, and then communicate the signed transaction back. In our implementation, users interact with the system through an interactive interface where they are first required to insert the policies to add, then they are informed of the derived smart contracts and relative deployment transactions waiting to be signed. The users can then check that the transactions correctly represent the policies they intended and, if they are satisfied, communicate the signed version of the transactions. Once the CH_D receives the signed transactions, it checks the signatures, and if they are correct it sends them to the `geth` node to be broadcast to the Ethereum communication network.

The CH_D receives a confirmation or error message depending whether the deployment was successful or not. This approach guarantees that the private information of 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_D receives back from `geth` the contract address, which is returned to the PAP_O to be subsequently embedded in the SMART RESOURCE code through the RIP. We remark how the RIP needs such address to correctly instantiate the LINKER, and so the SMART POLICY needs to be deployed before the

SMART RESOURCE. This is semantically correct, since it prevents the owner from deploying an unprotected resource.

The role of the CH_E, instead, is minimal in the SMART RESOURCE scenario, because the code that is inlined in the SMART RESOURCE by the RIP to invoke the execution of the proper SMART POLICY, actually implements together the functionality of the PEP and CH_E of our general architecture through one Solidity modifier.

5.4. PDP and PIPs

The traditional tasks of PDP and PIPs are merged together into the SMART POLICY. This contract is dynamically generated by the PAP from a XACML policy and, once deployed, resides on the Ethereum blockchain in EVM bytecode. As already stated before (see Section 4.2), the decision process of the PDP is performed by the decentralised execution of the `evaluate` function of the contract and the attribute value retrieval is performed by function 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.

6. Considerations

The main advantage of our proposal is that the policy management and evaluation processes are performed on a blockchain and this causes our system to inherit the blockchain technology advantages, i.e., it is auditable, always available, distributed (so no single point of failure or attack), tamper resistant, etc.

6.1. Transparency

Since the SMART POLICY 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, for each access request, both the policy that has been enforced and the related evaluation result are stored on the blockchain, thus allowing auditability. Hence, any user whose access request has been fraudulently denied by the resource owner can use the data stored on the blockchain to prove that the access right should have been granted instead. Since the blockchain is immutable, even when a SMART POLICY is revoked, its code and the entire access request log remain still stored and accessible on the blockchain. Hence, long term auditability is supported.

This is a relevant advantage with respect to running the Access Control system on the premises of the resource owner. As a matter of fact, the blockchain represents a trusted execution environment for performing the decision process and a trusted storage system for storing the resulting access decision. On the contrary, running the Access Control system on the resource owner's premises would enable him to alter the execution of the decision process thus forging a fake access decision and/or to store in the access logs a different decision than the one resulting from the actual evaluation of the policy. Obviously, if, on the one hand, the adoption of our system would relieve the resource owner from installing and managing an Access Control system on his premises, on the other hand, it could introduce some costs for

blockchain transactions (e.g., in he case of the Ethereum protocol, gas expenses are required to deploy and execute smart contracts). A description and evaluation of such costs for our proof of concept implementation is given is Section 7.

Adopting a permissioned blockchain (see Section 2.3), such as Hyperledger Fabric [47], for the deployment and execution of the proposed Access Control system is a possible solution to reduce such costs. In fact, the introduction of a level of trust, by allowing only trusted miners to append new blocks to the chain, allows the use of less complex, and so cheaper, distributed consensus algorithms. Furthermore, a cryptocurrency backed by a permissioned blockchain is expected to not experience high price fluctuations (caused by public trading and speculation), if publicly priced at all, and so it is expected to be more stable and predictable even from a purely monetary point of view.

It is important to remark that, in our system, auditability concerns the access control policy storage and evaluation, i.e., the policies that have been uploaded on the blockchain and the decisions that have been taken concerning the rights to access resources. For what concerns the enforcement of such decisions, the crucial component actually executing it is the PEP. If the resources to be protected are traditional digital resources, the PEP is not deployed on the blockchain. Consequently, the actual enforcement phase does not benefit of the blockchain advantages, auditability among all. Hence, the PEP could fraudulently ignore an access decision received from the blockchain, or could even drop an access request avoiding to invoke the evaluation of the SMART POLICY, for instance because of a malicious resource owner. Instead, in the SMART RESOURCES scenario presented in our reference example, the PEP and the resources to be protected both reside on the blockchain, thus allowing the PEP code inspection as well as the auditability of the access decision enforcement.

Obviously, the adoption of the proposed system does not relieve the resource owner from adopting the other typical security measures available for their resources, depending on the resource type (e.g., in case of a server, promptly install all the newly available security patches to the operating system).

6.2. Privacy

A clear consequence of auditability is a potential privacy issue. As a matter of fact, the access control policies, the access requests, and the related access decisions are stored on the blockchain. Adopting a public blockchain means that all the users of such blockchain can read such data, and that all the miners are enabled to read them as well, in order to execute the SMART POLICY contracts when invoked [48]. The only anonymity protection mechanism offered by most current blockchain protocols is the use of pseudonymity. This property, weaker than proper anonymity, guarantees that users may take part in the protocol hidden behind any number of randomly generated identifiers that carry no information between themselves, nor about the user identity.

Thanks to the pseudonymity property, the access requests and the related access decisions, although publicly accessible, are not directly linked with the real identities of the users who performed them, but just with their anonymous IDs. However, in some Access Control scenarios some actors need to know the real identity of a user, violating user pseudonymity. For example, the resource owners should know which IDs have been assigned to those users

for which they what to enforce policies based on identity. Moreover, in some scenarios, the real user identities should be known also by some Attribute Providers, i.e., the entities owning SMART AMs to maintain users' attributes, because these entities must assign to each ID the right attributes values and must update these values when necessary.

Furthermore, some approaches have been proposed in the literature to break users pseudonymity, mainly based on pseudonyms behaviours on the blockchain. The most studied so called *deanonymization attacks* mainly concern the Bitcoin protocol and rely on heuristic rules based clustering [20, 49, 50, 51] to try to group together all pseudonymous IDs belonging to the same user, but similar approaches are possible for similar blockchain protocols.

If the proposed Access Control system is adopted for regulating the access to smart contracts, such as in the reference example we presented in Section 5, the privacy concerning access control events is not, however, an issue introduced by our approach. In fact, the resources protected by the proposed system are smart contracts and, consequently, their invocations and executions are already registered on the blockchain. Hence, in such case, our system does not introduce further major privacy issues than the ones already accepted by using the chosen blockchain protocol. Instead, for what concerns the application to traditional digital resources, the proposed system, in its current version, can exploit public blockchains only in those scenarios where the privacy concerning access control events is not an important requirement.

We point out how auditability and privacy issues are two sides of the same coin, as they both derive from the blockchain technology feature of storing information in a tamper free and publicly accessible way. So, in trying to tackle the privacy issue we should not hinder the desired system transparency. This is achievable by restricting the access to given information only to the interested parties, and the simplest way would be to adopt a private blockchain. In such a system, both write (adding new blocks to the chain) and read (accessing information stored in the blocks) operations are restricted to trusted parties only. Even if the data can be encrypted in a private chain to further restrict the information access to other parties, enough miners still need to be able to decipher such information to efficiently process it in clear. As such a private blockchain does not solve the issue in general, limiting itself to replace such issue with the need for trust in third parties (the private chain miners to be trusted).

If the information is obfuscated also for the miners, they consequently can not check its meaning, hence they are not able to guarantee its content. To allow a consistent chain without information disclosure, sophisticated cryptography techniques are required. For example, homomorphic encryption [52] could be used to process encrypted data without the need to decipher it first. Alternatively zero knowledge proofs could be used to guarantee that a certain constraint is enforced without disclosing private information. Unfortunately, the main issue with such approaches is usually their price. For example, homomorphic encryption is still too costly to be used for complex computations (see [53]). Do note that some notable exceptions of privacy aware blockchains are available, such as the Zerocash project [54] based on the Bitcoin protocol, but in general the performance overhead necessary to use the advanced cryptography techniques employed affects too much the transaction throughput and system scalability to be deemed worthy. Moreover such systems [54, 55] do

not usually enable smart contracts.

The current main proposals to achieve privacy in the presence of smart contracts are to simply encrypt the contract data and calls, only storing on chain such obfuscated information (either directly or through a cryptographic hash). Such approach is employed by Quorum (through so called private contracts and transactions) [56], based on the Ethereum protocol, and Hyperledger (through so called private channels) [47]. In both cases the miners have no access to the private information and so its consistency needs to be enforced by the participants (usually through distributed consensus algorithms). For example, double spending becomes possible within private transactions. To avoid such issue, a proposal to use the Zerocash tools in a smart contract environment has been proposed in the Quorum project. A proof of concept preliminary implementation has been presented in [57] to allow for the private management of digital assets, named *z-tokens*, while providing proof of assets transfers (through zero knowledge cryptography). Further approaches for enhancing privacy in smart contract based blockchains have been recently proposed in the literature as well, e.g., [58, 59]. However, as clearly claimed by the authors of [58], the cost of the proposed solutions for deploying and executing privacy preserving smart contracts in current blockchain systems is still too high.

6.3. Modularity

We want to remark that the main contribution of our proposal in the reference example scenario basically consists in outsourcing the Access Control logic from the SMART RESOURCE. This means that the user relies on a third party service (i.e. our system) to write all the Access Control logic. However, this does not introduce an heavy trust requirement in such third party. In fact, the service builds the SMART POLICY and integrates the SMART RESOURCE contract instead of the user, but it needs the user approval (i.e. its signature) to deploy both of them. This means that the user can check before deployment that the service was not malevolent and stop the deployment at any moment if they are not satisfied.

Such outsourcing of code writing grants a clear advantage in terms of user effort and errors avoidance. Since the SMART POLICY is automatically generated and linked to the SMART RESOURCE any human error is avoided. Furthermore, since the user does not have to write an Access Control contract custom implementation but relies on a statically written solution instead, checks of the contract security are easier and generalizable. Of course this also results in possibly fewer errors in the SMART RESOURCE implementation in general since the user can focus all its efforts and attention on its intended logic independently of the Access Control side. Finally, the introduced modularity in separating the control logic from the smart contract to be controlled allows for the two contracts to be deployed and managed separately. Since this results in two distinct transactions, this means that heavier (in terms of gas cost) contacts are deployable, since the two transactions can fit in different blocks even if their combined cost would be greater than the block gas limit.

6.4. Policy Complexity

The complexity of a SMART POLICY depends on the number of MATCHES in the original XACML policy and their individual complexity (dependent on operation and data type of

the single MatchE). Our proposal is to translate an entire XACML policy into a single smart contract. In general, blockchain protocols employ techniques to price the scarce resources on chain (such as storage and computation) through proportional fees. In such a scheme, as the complexity of the XACML policy grows, the corresponding SMART POLICY gets more expensive in terms of fees. Most blockchain protocols have limits on the maximum size of blocks (mainly to guarantee reasonable block propagation times among the peers), and so have a hard limit on transaction sizes as well. This introduces an hard cap on the maximum size (i.e., complexity) of smart contracts (both for deployment and execution), including SMART POLICIES, that would, in turn, introduce a limit in the maximum complexity of XACML policies supported by our system. For example, our proof of concept implementation is developed on Ethereum, that uses mandatory gas consumption to pay for transaction fees, and limits the size of each block by specifying a block gas limit (i.e., the maximum cumulative gas allowed to be consumed by all transactions in that block), as explained in Section 2.3. Consequently the complexity of the XACML policies that can be expressed is limited by the current block gas limit of the Ethereum blockchain.

However, since the policy complexity limit derives from the underlying blockchain protocol and not from the proposal itself, it is easy to circumvent. In fact it is possible to manage XACML policies of arbitrary complexity by splitting the resulting SMART POLICY in more than one smart contracts, i.e., to enhance the PTP in order to translate the XACML policy in a set of interconnected smart contracts. Since the complexity of a policy depends on the MatchEs it implements, which are translated into checks inside the `evaluate` function of the SMART POLICY, it is sufficient to only split the `evaluate` function checks among different contracts. The corresponding SMART POLICY, represented by a set of smart contracts rather than a single one, would contain a main smart contract as entry point to invoke the `evaluate` and the other functions, while the body of the evaluate function would contain an external call to a second smart contract, only tasked at computing a portion of the MatchEs of the policy, invoking the third contract for computing another portion of MatchEs, and so on, until all the MatchEs in the original policy are implemented by a smart contract. The last smart contract returns a result that is passed up to the caller, that combines it with the result obtained evaluating its portion of MatchEs, and passes this new result to its caller, and so on, until a result is returned to the main SMART POLICY smart contract, that finally returns the global result.

Of course the process could be designed efficiently to allow for short circuiting of the evaluation calls whenever possible. Only the PTP needs to be changed, since the main SMART POLICY contract would still be the only point to trigger a policy evaluation once deployed. The split of the SMART POLICY into different contracts would be transparent to the other modules of the system. As long as the single contracts representing the SMART POLICY are individually smaller than the gas limit they can be inserted into a block and so any policy would be deployable. The only drawback is that a SMART POLICY would require more than one deployment transaction, increasing deployment costs (since multiple transactions often cost more than a single cumulative transaction). Do note that, in general, the expected wait time does not necessary increase to validate many smaller transactions compared to a bigger one, since many small transactions can better fit into the free space

available in blocks, while a big one has to wait for a single block with a lot of free space (of course assuming that the big transaction would fit in a block at all).

The previous approach does not decrease the overall cost of the SMART POLICY deployment (it might increase it, instead). It simply allows to surpass the consumption limitations for a single contract. Furthermore the proposed solution only concerns policy deployment, not its execution. If we require the evaluation process to be performed on chain it needs to be triggered by a single transaction that would have to abide to the block gas limit.

6.5. Delegation

Finally, in order to be successfully adopted in a larger number of application scenarios, the proposed blockchain Access Control system could be enhanced by introducing further features, such as delegation capabilities. Delegation is a mechanism that allows a user, say A, to enable another user, say B, to act on his behalf, i.e., user A is enabled, through a delegation action, to transfer (some of) his access rights to user B [60, 61]. When the delegation action is performed by a user on his own initiative (i.e., without a centralized control or policy owner intervention), policy violations could arise because of such delegation. Hence, in order to regulate who can delegate whom and to avoid policy violations due to delegation, some delegation enabled authorization systems allow policy makers to specify a set of *delegation authorization rules*, i.e., policy rules which control who can delegate which privileges to which other users, and/or which additional privileges can be acquired through delegation by each user.

In order to support delegation, the proposed blockchain based Access Control system could be modified as follows:

- a new `evaluateWithDelegation` function is added to SMART POLICIES;
- the PTP is modified to translate the delegation authorization rules included in the policy in order to write the `evaluateWithDelegation` function (as explained below);
- the PEP is modified to also allow for processing of access requests with delegation;
- the PAP and CH_D are enriched to process new `createDelegation` requests;
- a new type of smart entity, named SMART DELEGATION, is introduced to represent a delegation on the blockchain.

When user A wants to delegate the right to perform an operation o on a resource r to user B, it starts the `createDelegation` process by invoking the PAP. The PAP requests a new `createDelegation` transaction (processed by the CH_D) that deploys on chain a new SMART DELEGATION containing the delegation data specifying, for example, the ID of user A, the operation o, resource r, ID of user B, and the delegation expiration date. Since it is requested by the user A to create the new delegation, its expenses are paid by A and are completely independent from the SMART POLICY and resource owner.

To perform the delegated action on the resource, B submits to the PEP a delegation access request, specifying the link to the SMART DELEGATION (i.e., the address of the

corresponding smart contract) that, supposedly, grants them the access rights. The PEP triggers the SMART POLICY evaluation process by creating a transaction which calls the `evaluateWithDelegation` function of the SMART POLICY (instead of the `evaluate` one) passing the SMART DELEGATION link as parameter.

The `evaluateWithDelegation` function checks that the SMART DELEGATION has been actually created by A, retrieves from it the needed delegation information, and checks them against the delegation authorization rules. Do note that this is possible, since the `evaluateWithDelegation` has been written by the modified PTP, that has translated them into executable code, similarly of how it does it for normal policy rules into the `evaluate` function. If the delegation authorization rules are satisfied, the `evaluateWithDelegation` function then calls the `evaluate` function specifying A as requesting user instead of B, and returns the access decision accordingly.

In the previous solution, we use a dedicated new entity, the SMART DELEGATION, to represent delegations instead of the simpler solution of storing delegation information directly inside the SMART POLICY for two main reasons: costs and security. Employing an external entity to the SMART POLICY makes it possible to have an arbitrary number of delegations without affecting the SMART POLICY. Saving all delegation information inside the SMART POLICY would instead bloat it and possibly slow down lookup operations for all users accessing it. Furthermore, allowing other users than the policy owner to store data inside the contract would open it to attacks, such as denial of service attacks, by filling it with useless delegations.

7. Experimental Results

To validate our system, we studied the performance of our proof of concept implementation analyzing the following three cost measures:

- monetary cost;
- resource cost;
- time cost.

We do note that even the off-chain Frontend of our reference implementation is different from a traditional XACML Access Control system, mainly for the need for a XACML to Solidity parser. Nevertheless, the parser complexity is guaranteed linear in the size of the policy (and easily parallelisable in some cases). We do not consider the parser during our performance evaluation, focusing only on the on-chain operations. In fact, on modern hardware, the parsing time is in the order of milliseconds even for big policies, while the on-chain operations take on average seconds to complete. Thus the different order of magnitude of the studied measures further justifies our simplification.

We point out that, in an hypothetical scenario where our system is used to regulate the access to traditional digital resources, the SMART POLICY deployment phase is exactly the same as the one concerning the SMART RESOURCE scenario taken into account in our

reference example. Consequently, the deployment gas and time costs are the same as well. The SMART POLICY evaluation phase is, instead, different for the two scenarios. In fact, in the traditional digital resource scenario, a transaction calling the `evaluate` function of the SMART POLICY is submitted to the blockchain, while in the SMART RESOURCE scenario the `evaluate` function is called by the SMART RESOURCE itself, through a smart contract message. However, the gas cost required for the execution of the `evaluate` function is the same in both scenarios, the only difference is that in the traditional digital resources scenario the transaction cost needs to be paid explicitly (for further details see Section 7.1), while in the SMART RESOURCE case this gas cost was covered already by the SMART RESOURCE caller. This means that, from the gas cost point of view, the gas price payed in case of traditional digital resources is an upper bound (i.e., the SMART POLICY cost plus the transaction cost) for the SMART RESOURCE scenario (SMART POLICY cost only), obviously, considering the same SMART POLICY. The `evaluate` function time cost is the same in both scenarios as well. However, in the SMART RESOURCE scenario, the SMART RESOURCE and SMART POLICY evaluations are inserted in the same block, and this would mitigate the time cost of the SMART POLICY. For the previous reasons, in the rest of this section, we evaluate SMART POLICIES deployment and execution costs (considering both gas and time) independently from the scenario chosen.

7.1. Monetary Cost

Using a fee (or gas) based blockchain, we introduce a monetary cost for every transaction that is mined. In particular, since our reference implementation is based on the Ethereum protocol, in which gas is consumed by transactions, we performed a set of experiments to estimate the gas cost of our two kinds of transactions: SMART POLICY deployment and evaluation.

7.1.1. 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 amount of gas that must be payed independently of the code of the contract to be deployed (e.g. the fixed cost to create a new transaction, 21 000 gas at the time of writing, and the fixed cost to deploy a new contract, 32 000 gas at the time of writing [7]), $CodeCost$ represents the cost to store the actual 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 requires 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 of the policy, to their complexity, and to the number of different SMART AMs they require to contact.

Obviously, more rules and more complex rules require more code to be stored and managed. Since each rule consists of a set of MATCHES elements, 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, instead, its execution cost). In our test we experienced that to deploy an empty policy (i.e., a policy which always returns `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. Allowing the policy to use an additional SMART AM (to get the attribute values to be used in the MATCHES) consumes approximately 26 000 gas alone (to store the AM information), 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 that the current gas limit in our academic testnet is 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 in our testbed. According to our rough estimation, for example, a policy using 10 different SMART AMs and 90 MATCHES would have about the maximum size that could fit in one block. We think that a policy of that kind is quite large, because common policies typically use two or three attributes, and we do remark that this is just a constraint of our reference implementation and not of the proposal itself (see Section 6.4).

7.1.2. SMART POLICY *evaluation transaction*

To trigger the evaluation of a SMART POLICY for a given access request we use a transaction calling the `evaluate` function of the contract (See Section 4.2.3). The gas cost of such transaction (Gas_E) can be expressed 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 4.3.2, the `evaluate` function is a combination 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. In fact, the execution of the same expression could have very different costs depending on the actual values of attributes at execution time. To test this, we performed some experiments where we evaluated a policy using 3 different SMART AMs and 80 MATCHES in conjunction which exploit the values of the attributes provided by such SMART AMs. The first time we

purposely choose attribute values to satisfy all the MatchEs, obtaining a `Permit` result and consuming 210 643 gas for the relative transaction. Instead, setting the attribute values in such a way that the first MatchE is not satisiefd, the entire expression short circuits to false without evaluating all the remaining MatchEs. This results 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 was due to the fixed cost of the transaction itself (more than 20 000 gas). Hence, considering only the cost due to the *EvalFunctionCost*, the second execution costs 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 a SMART POLICY consisting of 90 MatchEs referencing 10 SMART AMs. The policy was a conjunction of simple boolean conditions that we knew being all true with the values returned by the SMART AMs (in order to avoid short circuiting). The resulting cost of an evaluation transaction (returning a `Permit`) under the previous assumptions is of approximately 230 000 gas. This shows how the policy evaluation cost (in the worst case that all MatchEs need to be executed) is considerably lower than the initial policy deployment cost. For example, for the policy of the above example, the evaluation transaction cost is about 5% of the gas consumed by the corresponding deployment transaction. Of course, this is just a rough estimation which depends on the policy an on the SMART AMs chosen. For instance, using a very costly SMART AM arbitrarily increase the policy evaluation cost without influencing the initial deployment cost (that is independent of it).

Given a reference policy invoking a single SMART AM and consisting of a conjunction of n simple single attribute MatchEs crafted to be always true over the SMART AM returned values, we depicted in Figure 5 the deployment and execution costs for increasing values of n.

We observe in Figure 5 that both the deployment and execution cost are linear in the number of MatchEs (of the same complexity), and that the deployemnt cost is much higher than the execution cost. This is a desirable property since the deployment cost needs to be paid only once, while the executions cost needs to be paid for each access request, resulting in a high setup cost but relatively low usage cost for SMART POLICIES. Finally, we remark that the block gas limit value in Figure 5 refers to our Academic testnet. Other blockchain could have a different gas limit, because the block gas limit can be initially set for each network and even adjusted later.

7.2. 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 (at the time of writing less than 200 GB for storing the main Ethereum blockchain using `geth`). However, since we delegate the storage of policies to the blockchain, we save the storage

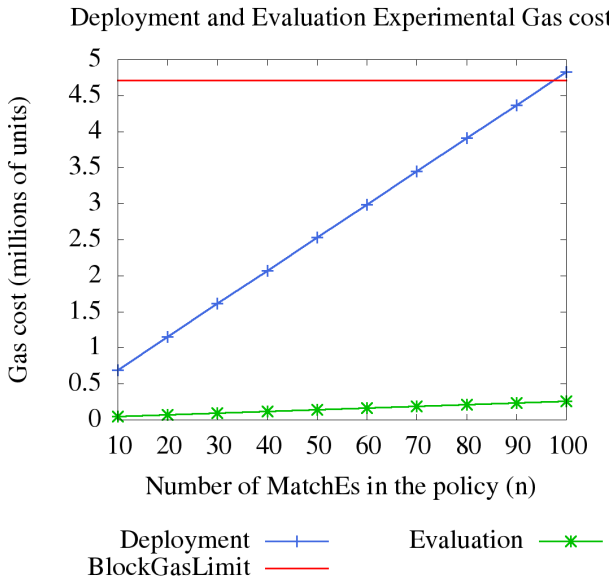

Figure 5: Gas cost of deployment and evaluation of the reference policy with increasing number of MATCHES n. The last value (n=100) is obtained by artificially increasing the block gas limit.

space required by traditional PAP implementations, although this 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 one relays on a third party that is in charge of managing 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 (depending on how the light node is actually implemented) in other full nodes, i.e., nodes following the full protocol specifications and storing the entire blockchain. 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. As already stated in Section 6, we remark that this is only an issue for the traditional digital resource reference example. In fact, in the reference example involving SMART RESOURCES, the user needs a blockchain client anyway to manage such SMART RESOURCES, and so our system could use the same client, not introducing additional resource requirements.

7.3. Time

The time overhead introduced by operations on the blockchain for the policy creation is caused by the SMART POLICY deployment phase, while at access request time, it is instead due to the execution of the SMART POLICY `evaluate` function. We do not consider the time needed for the user to check and sign a transaction since it is user dependent. Both the SMART POLICY deployment and execution times mainly consist in the time elapsed for

the corresponding transactions to be mined into a block. So, they both are expected to be roughly equal, on average, to the transaction confirmation time, that depends on the underlying blockchain. 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 should take, on average, 14 seconds. In practice, to estimate this cost, we should also take into account the transaction propagation times (that might be comparable to the mining time for poorly connected nodes). Moreover, in case of transaction congestion, a transaction can actually take longer to be included in a block. However, 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 confirmations). For these reasons, there is no simple way to determine how many blocks on average a new transaction will take to be confirmed. The most important factors influencing confirmation time of a transaction are highlighted more precisely in the following list:

- **Random mining time**: in a PoW based blockchain, blocks are generated at random times. It is guaranteed that the difference between the generation times of two consecutive blocks is equal *on average* to a predefined constant (and PoW difficulty is adjusted accordingly). This means that, in practice, users can expect an high variance in the time to be waited between the generation of two subsequent blocks. This is especially true during difficulty adjustment periods. For example, if the network experiences a sudden drop or increase in the computing power of miners dedicated, this can negatively affect the block generation time (respectively increasing or reducing the expected average time) before the difficulty adjustment process can automatically keep up with the change. Also, during forks, the validation time might increase since the mining power is split between different branches, and each branch is working to solve a PoW with a difficulty not correctly proportionate to the actual computing power.

  Supposing a constant flow of new transactions submitted to the network, the transactions that are submitted during the mining of the blocks that take more time, have to compete with more transactions waiting to be mined, and so will overall have a smaller probability to be added in that block with respect to the transactions that are submitted during the mining of the blocks that take less time. This is not a problem in general for a blockchain since the constant expected generation time guarantees that on average equal transactions will have the same chances of being added and so the same expected wait time to be validated (assuming blocks are not full). However, in case of time sensitive applications this could be an issue affecting user experience in some circumstances.

  This issue can be mitigated using different distributed consensus algorithms with guaranteed constant mining times, e.g. Proof of Authority (PoA) [42], but such algorithms usually require to weaken the *lack of trust* assumptions of public blockchains and so are used for permissioned ones (see Section 2.3).

- **Discrete block creation**: a blockchain timestamps transactions by dividing the set of records in blocks. Basically, the system takes discrete temporal snapshots of its state

at each block. This discrete time snapshots can be taken at constant times intervals on average (e.g., in PoW blockchain) or exact constant time intervals (e.g., in a PoA blockchain), as explained before. In both cases, transactions that are submitted closer to the moment when the next snapshot is taken have to wait less time for the next block and so a chance to be validated. In general, transactions submitted immediately after a new block is found have to wait more than transactions submitted immediately before a new block is found, assuming that there is enough free space in new blocks and miners keep listening and adding transactions to their blocks while mining. If blocks are saturated, instead, the opposite is true since equal transactions will have more chance to be added to the next block if they are submitted immediately after a new block is found.

- **Latency**: the chances for a transaction to be mined increase as more miners get to know about its existence. This means that transactions submitted by poorly connected nodes are expected to wait on average more time than identical ones submitted by better connected nodes. Furthermore, the unpredictability of network latency influences the time needed for the transaction to be known and so its chances to be mined. Such latency is also influenced by the mean used to connect to the blockchain communication network. For example users connecting through a browser based service (such as the popular *MetaMask* [62]), where request needs to travel through the internet to reach a third party node directly connected to the communication network, will incur in an higher latency, in general, then users connected directly to the network through a full node. Finally latency can also be manipulated by malicious entities trying to isolate nodes to delay their transactions (for example during double spending attempts).

- **Congestion**: as block space gets exhausted it increases the competition among transactions. The same transaction will have to wait more for validation during times of high congestion with respect to times that free space in blocks is more abundant. Do note that in this paper we consider as *block congestion* the ratio between the total gas used by all transactions in a block and that block gas limit.

- **Fees**: as already noted above, users can decide to offer higher fees for their transactions to be more desirable by miners. This becomes more relevant during high congestion periods. In fact, if space is available in blocks than it is convenient for miners to insert any transaction as long as the fees repays for their effort (e.g., computing the contract contained), so users have no reason to offer higher fees. On the contrary, if blocks are full (i.e., there are more pending transactions than the space in a block to contain all of them) then the miners will choose only the higher paying transactions to reap higher rewards with the same effort (since the *gasCost* is independent from the gas actually consumed).

- **Lack of a global clock**: as for most distributed systems, the communication network lacks a global clock. This means that each node, and, especially, each miner has its own local time that may differ (even significantly) from the ones of others. Since miners

solving a block are the one deciding the block timestamp, this means that blocks can have inconsistent times among them. Currently, Ethereum only requires that a block timestamp must be greater than the previous block timestamp [7]. So miners can maliciously alter their own timestamps. To mitigate this problem, each node could use its local timestamp to keep a local log recording when transactions and blocks are first seen by such node. However, this is not a good solution as well, since, as already explained, network latency could cause a transaction or a block to be seen by a (poorly connected) node after a significant delay with respect to the time when, respectively, the transaction has been submitted or the block has been created.

- **Context dependent evaluation** (evaluation only): the same transaction containing a function call can take different times to compute depending on the current context it is executed in. This is strictly connected to the monetary cost we explained before in Section 7.1, here it suffices to say that since a function execution depends on the current state it can have different costs and so require more space in a block. This influences its probability of being mined sooner. For example we can think of a function that has a conditional clause depending on the current block height number, if it is even, it terminates, otherwise it does a lot of heavy computations. Clearly the function will require more computations and so it will cost a lot more gas if the current block number is an odd number, and heavier transactions have longer expected validation times.

To alleviate all those issues and reduce the variables that concur to determine a transaction confirmation time, it is generally advisable to consider the blockchain itself as measurement of time through the *block height* (i.e., the distance of a block from the genesis block). In fact, by grouping the transactions in immutable discrete snapshots (blocks), it is possible to see the blockchain as a timestamping service. Using such a solution the confirmation timestamp of a transaction is the height of the block it was mined in. This means that we can measure the time it took to confirm a transaction as the difference in block height between the last known block that the network had mined at transaction deployment time (relative to the transaction creator) and the block where the transaction has been inserted. This solution solves the *random mining time*, *discrete block creation* and *lack of global time* problems showed above. Considering that *latency* is an unsolvable problem inherent of the system and the *context dependent evaluation* problem can be avoided by choosing functions and context appropriately, using the block height allows us to perform experiments with only *congestion* and *fees* variables to take into account. This way, even if we still have some degree of unpredictability due to latency, it is easier to isolate the cause of long waits and delays.

Using block heights as timestamps for transaction confirmations is especially effective in our academic testnet. In fact the network is heavily underused and so there happens to be long periods of time during which the blocks are almost empty. This allows us to perform experiments without worrying about block congestion and fee races.

To test the deployment time of a SMART POLICY we measured the confirmation time, expressed in block height, of a reference SMART POLICY, increasing the number of MATCHES,

used as an approximated measure to estimate policy complexity. Each measurement has been repeated for 50 times. As reference policy we used the aforementioned reference policy showed at the end of Section 7.1, i.e., a SMART 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. Furthermore, to avoid to introduce artificial congestion in the blocks with the experimental data themselves, we also temporized the deployment of each policy by waiting for one block after each successive contract deployment.

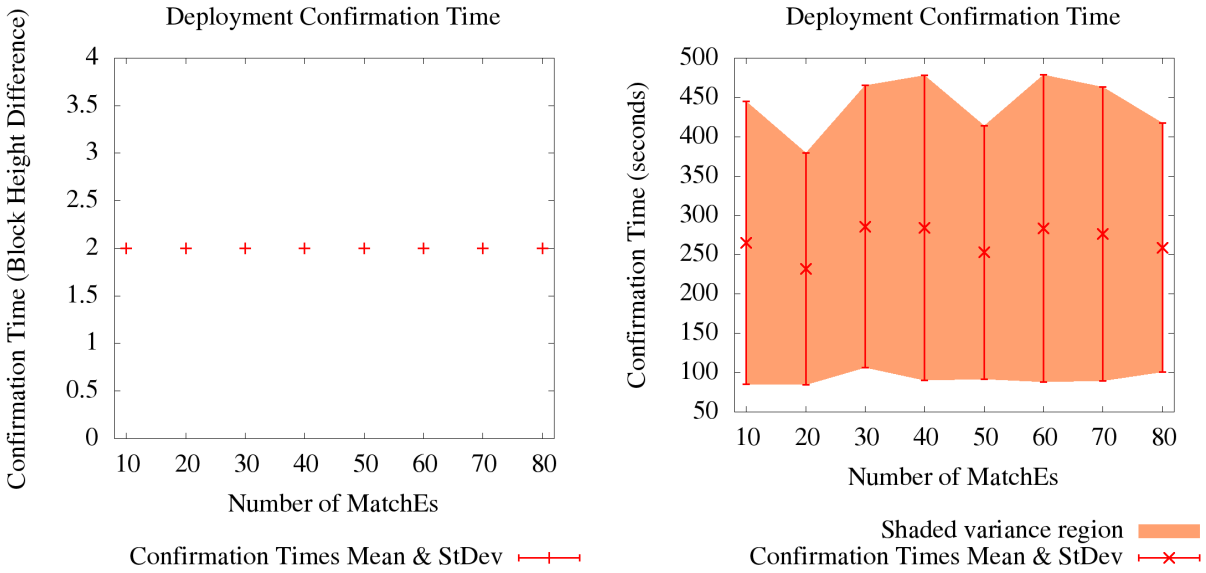

Figure 6: Average (with standard deviation) deployment confirmation times, expressed through block height difference (a) and in seconds passed between the current block timestamp and the timestamp of the block the transaction gets confirmed in (b). Do note that in (a) the standard deviation is always zero. Experiments on the Academic testnet.

Figure 6(a) shows the average deployment confirmation time (expressed in number of blocks) and its variance for an increasing policy complexity, expressed by the number of MATCHES in the SMART POLICY. We do note that 1 is theoretically the optimum time, because it means that the transaction is confirmed in the first block available. We observed that, in our testnet, the actual optimum is 2 instead of 1, i.e., no transaction is inserted in the immediately available block. This fact can be explained as consequence of the mining behaviour of our nodes. In fact, it looks like each node, once has created a candidate block and started to try to solve the corresponding proof of work, ignores new transactions until it succeeds or some other nodes solves its proof of work first. This means that new transactions are considered for insertion in new blocks only after the first block that could contain them is mined, and so they have to wait at least two blocks to be confirmed.

The situation is different for the Ethereum main chain which which is currently experiencing a more relevant congestion problem, as we show in Figure 7.

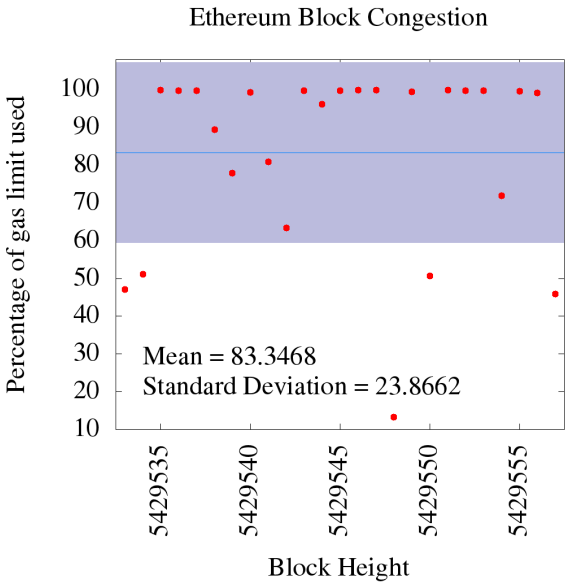

Figure 7: Percentage of gas used in 25 consecutive blocks of the Ethereum main chain from block 5 429 533 to block 5 429 557 [63], average congestion 83.35% with standard deviation 23.87 .

Figure 6(b) shows the results of the same set of experiments considering block confirmation time measured in seconds instead of using block height. Hence, the transaction confirmation time was measured as the difference in seconds between the timestamp of the transaction confirmation block and the timestamp of the last block known by the transaction creator at the time the transaction was broadcast to the network. Moreover we point out that the block confirmation time in our academic testnet (that has been set up to 136 seconds at the time of our experiments) is higher with respect to the main Ethereum network (14 seconds). It is clear from the comparison of the two figures how using block height as time measure gives a cleaner understanding of the phenomenon. In fact the comparison between Figure 6(a) and Figure 6(b) shows the impact of the *random mining time* issue. We can see clearly in Figure 6(a) that the number of MATCHES in the SMART POLICY has no effect on the deployment time as long as blocks have enough free space to include the policy deployment transaction. Instead, from Figure 6(b) it looks like SMART POLICIES with 20 or 50 MATCHES have shorter deployment time. Of course there is no reason why this should be the case, as proved by Figure 6(a) that is derived from the same set of experiments, it is just a consequence of the randomness of mining times manifested by a relatively low number of experiments (50). We can compute a theoretical expected validation time as two times the block confirmation time, i.e., $2 * 136 = 272$ seconds (although an unlucky combination of all the causes affecting confirmation time listed before could cause an extraordinary long wait). We do note that the expected validation time matches with our experimental results, as the average of all confirmation times measured is 267.34 seconds, close to the theoretical value of 272 seconds.

As shown in the previous paragraph, performing experiments on our academic testnet

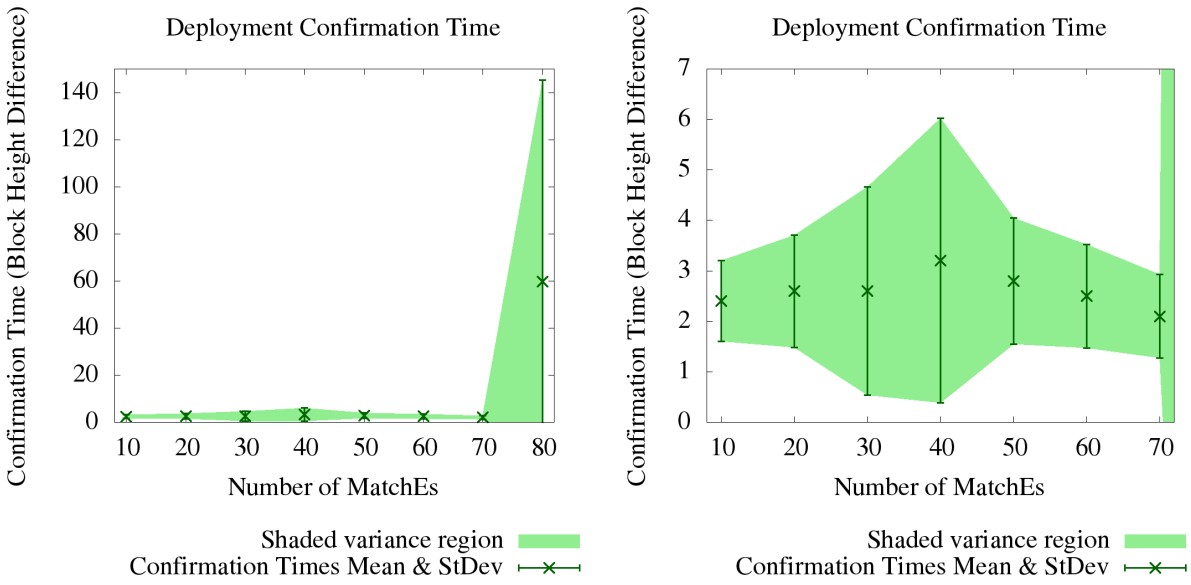

Figure 8: Average (with standard deviation) deployment confirmation times results expressed through block height difference. Figure (a) depicts the complete results, while Figure (b) represents a zoom on the results of the experiments with less than 80 MatchEs only. Experiments on the Ropsten testnet.

allowed us to prove some theoretical assumptions, but was not a good simulation of a real world scenario. To have performance evaluations closer to a real scenario, we decided to perform our experiments also on the public Ethereum testnet Ropsten. Considering that it is widely used globally by many users, testing their applications before deployment on the main Ethereum chain, we believed this was a better simulation of a real world scenario. In fact both the expected mining times (14 seconds) and block congestion of Ropsten are closer to the ones of the Ethereum main network. So we repeated the same experiments on the Ropsten testnet, repeating each measurements 10 times for each experiment. The deployment confirmation times results, measured in block height difference, are shown in Figure 8(a).

Figure 8(a) shows that the confirmation times of the experiments concerning the smart policies with 80 MatchEs are considerably larger w.r.t. the experiments concerning smart policies with a smaller number of MatchEs. Moreover, this problem has not been detected in our academic testnet, as shown by Figure 6(a). This happened because the size of a transaction to deploy a smart policy with 80 MatchEs is close to the *gaslimit* of the blocks (see Section 7.1), so this transaction needs blocks with a lot of free space to be confirmed. This is not an issue in our academic tesnet, where most of the blocks are empty. Instead, finding empty blocks to fit this transaction is more difficult in a real world network such as Ropsten. In Figure 8(b) we show the same results zooming on the first seven sets of experiments only to give a better insight excluding the 80 MatchEs special case.

The results are worse than the ones obtained for the academic testnet, and shown in

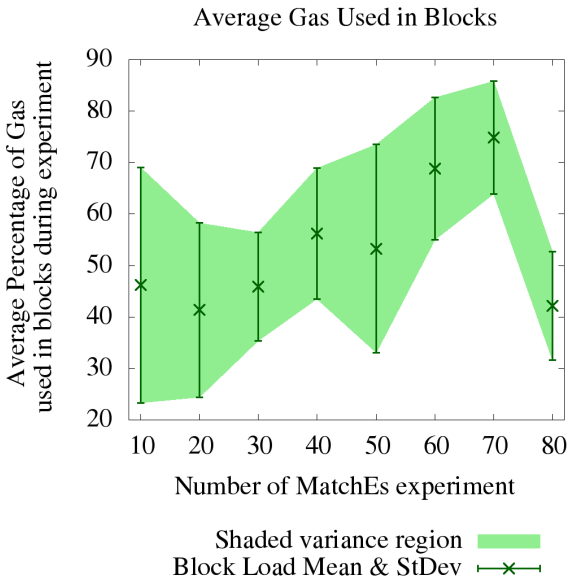

Figure 9: Average block congestion times experienced during the deployment transaction confirmation experiment shown in Figure 8.

Figure 6(a). Of course this was expected since Ropsten is more congested than our academic testnet. We note that in Ropsten is possible to achieve the optimum result of mining a transaction in 1 block only, and this is due to a smarter mining policy adopted by the Ropsten testnet nodes. We also note that we obtained an average confirmation time quite low (always less than 4 blocks), and independent from the number of MatchEs. Comparing this with the results from the experiments with 80 MatchEs, we can conclude that the number of rules in a SMART POLICY does not influence confirmation times as long as the resulting transaction is small enough to fit into the average space available in blocks. To further prove our point we measured the average block congestion during the experiments we conducted. The results are depicted in Figure 9. To measure the average block congestion experienced by a transaction, we compute the average of the congestion values of each block that the transaction had to wait for its confirmation. We then compute the average among the 10 measurements performed for each experiment. We can see that block congestion does not influence the block confirmation times. The case with highest block congestion (i.e., 70 MatchEs) is also the one with a lower average confirmation time.

Also for the Ropsten testnet we measured the time in seconds instead of considering the block height difference. The corresponding results are shown in Figure 10. The same considerations made before for Figure 6(b) do hold also in this case. We observe that even if the transaction confirmation times expressed in block height difference are higher in Ropsten than in the academic testnet (see Figure 8(b)), thanks to the much lower average block confirmation time (14 seconds vs 136 seconds), the resulting average confirmation times in seconds are lower in Ropsten (always below 50 seconds in Figure 10 (b)) than in the academic testnet.

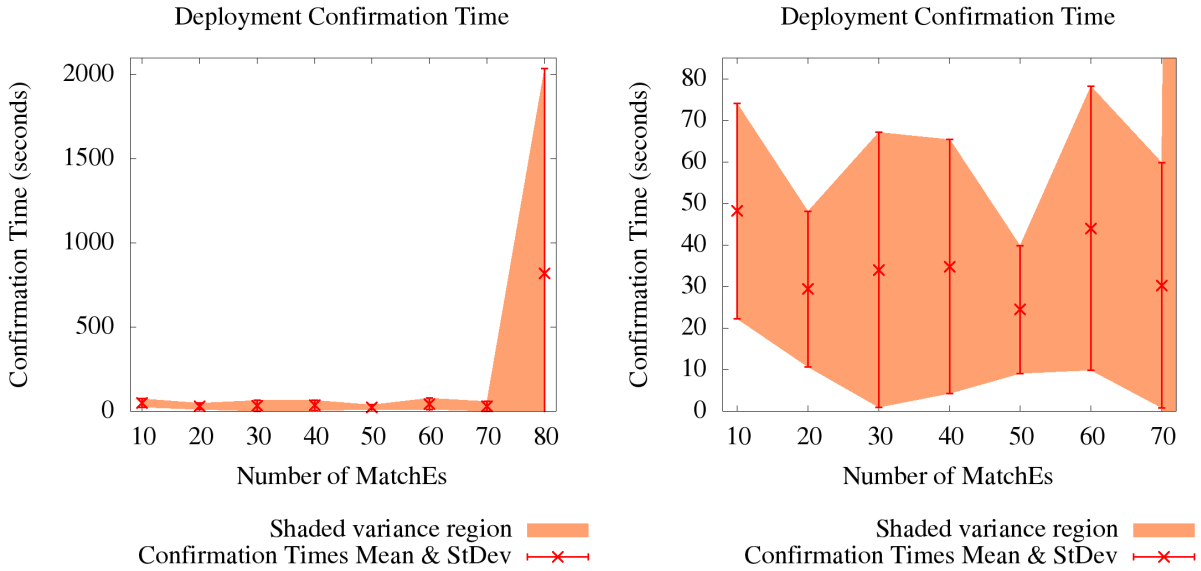

Figure 10: Average (with standard deviation) deployment confirmation times results expressed in seconds passed from the current block timestamp and the timestamp of the block the transaction gets confirmed in. Figure (a) depicts the complete results, while Figure (b) represents a zoom on the results of the experiments with less than 80 MatchEs only. Experiments on the Ropsten testnet.

As we have shown in Section 7.1 the SMART POLICY deployment transactions need to store a new smart contract on the blockchain and execute its constructor. This often results in transactions consuming a considerable amount of gas and space in a block. In comparison, SMART POLICY evaluation transactions (i.e., transactions executing calls to the SMART POLICY `evaluate` function) are in general much lighter. This causes those transactions to be easier to be added to a block and so they take, on average, less time than deployment ones. This is a good property since each SMART POLICY needs to be deployed only once, but can be executed several times (as long as the policy remains active). We decided to give a measure of the system *time efficiency* by studying the execution of SMART POLICY evaluation transactions alone. As time efficiency measure we chose to use the *execution throughput*. Given n SMART POLICY evaluation requests submitted at the same time, we define as *execution throughput* the percentage of them that got evaluated immediately (i.e., got confirmed in the first available block). Instead the *execution time* is simply the average time that took for all the n requests to be evaluated.

As explained above, the *context dependent evaluation* issue may affect these results. To mitigate this, we chose to execute all our experiments on the reference policy described above, which is built to be non short circuiting, thus ensuring that all the MatchEs of such policy are always all evaluated in our tests. Furthermore we performed our experiments on the Ropsten testnet since it better reflects a real world scenario as explained before. We measured both execution throughput and execution time for an increasing number of requests and of MatchEs in the SMART POLICIES. In particular, we performed a set of five

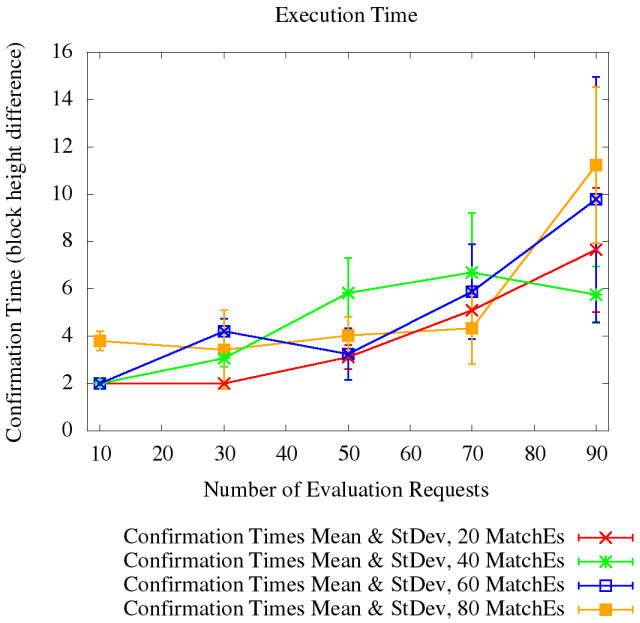

Figure 11: Average (with standard deviation) evaluation confirmation times results expressed in block height difference, for increasing number of execution requests and number of MATCHES in the reference SMART POLICY executed. Experiments on the Ropsten testnet.

experiments for each fixed number of evaluation requests, the number of simultaneous evaluation requests were increasing from 10 to 30, then 50, 70 and finally 90. These experiments were repeated calling the evaluation of a reference SMART POLICY with 20, 40, 60 and 80 MATCHES. We chose to repeat our experiments only five times for each setup and only for four different SMART POLICIES to avoid excessive network bloating. In fact, the presented experiments already involved five thousand single transactions. Since we performed those experiments on the global official testnet we tried to keep the number of transactions contained to avoid affecting all the other users concurrent tests. The cumulative results showing the average *execution times* expressed in block height difference are reported in Figure 11, while in Figure 12(a), Figure 12(b), Figure 12(c) and Figure 12(d) are shown the individual results for each SMART POLICY. As expected we can notice from the figures how the average confirmation time increases with the number of evaluation requests. This is expected since more evaluation transactions compete for the same finite space in each block.

In Figure 13(a), Figure 13(b), Figure 13(c) and Figure 13(d) are shown the results of the same experiments of Figure 11, but considering the *execution throughput* instead of the *execution time*. The definition of *execution throughput* is relaxed to consider the percentage of transactions that are confirmed within x blocks, with x a parameter. The figures show the results for x between one and four. Unsurprisingly and despite some exceptions, probably caused by random block congestion, the results show the expected trend that more requests result in a lower *execution throughput* (as already verified for *execution time*). We do note that no transaction is ever confirmed in one block (i.e., inserted in the first available block),

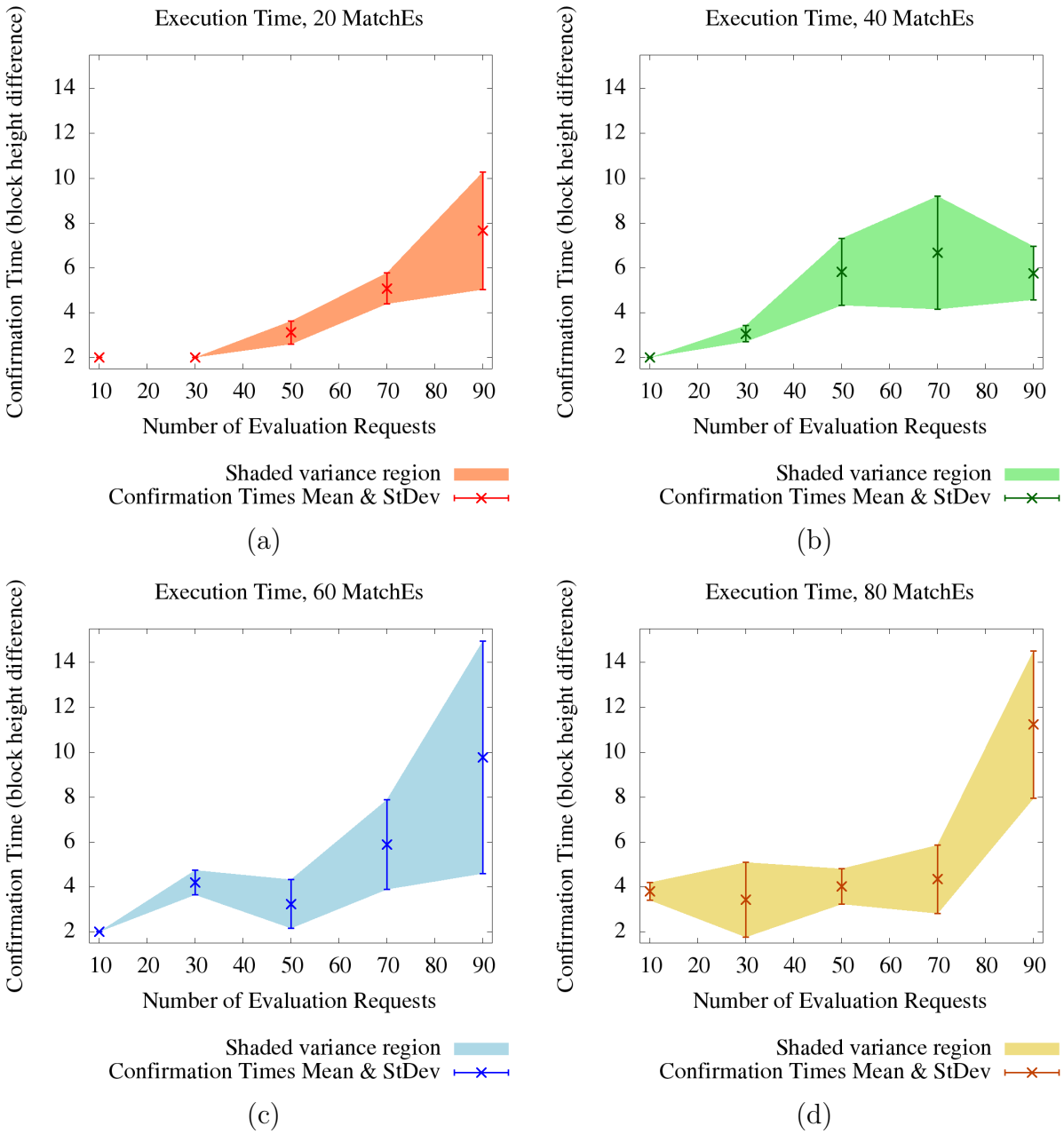

Figure 12: Average (with standard deviation) evaluation confirmation times results expressed in block height difference, for increasing number of execution requests to a reference SMART POLICY with 20 (a), 40 (b), 60 (c) or 80 (d) MATCHES. Experiments on the Ropsten testnet.

this is an artificial consequence of our experiments manager program. In fact, we first observe the current block (i.e. the last block known by our node) and then spend some initial time to create evaluation requests and broadcast all transactions at once. This, coupled with network latency, may be an explanation for the observed delay, that, for example, was not present during the deployment time experiments (where many SMART POLICIES got deployed

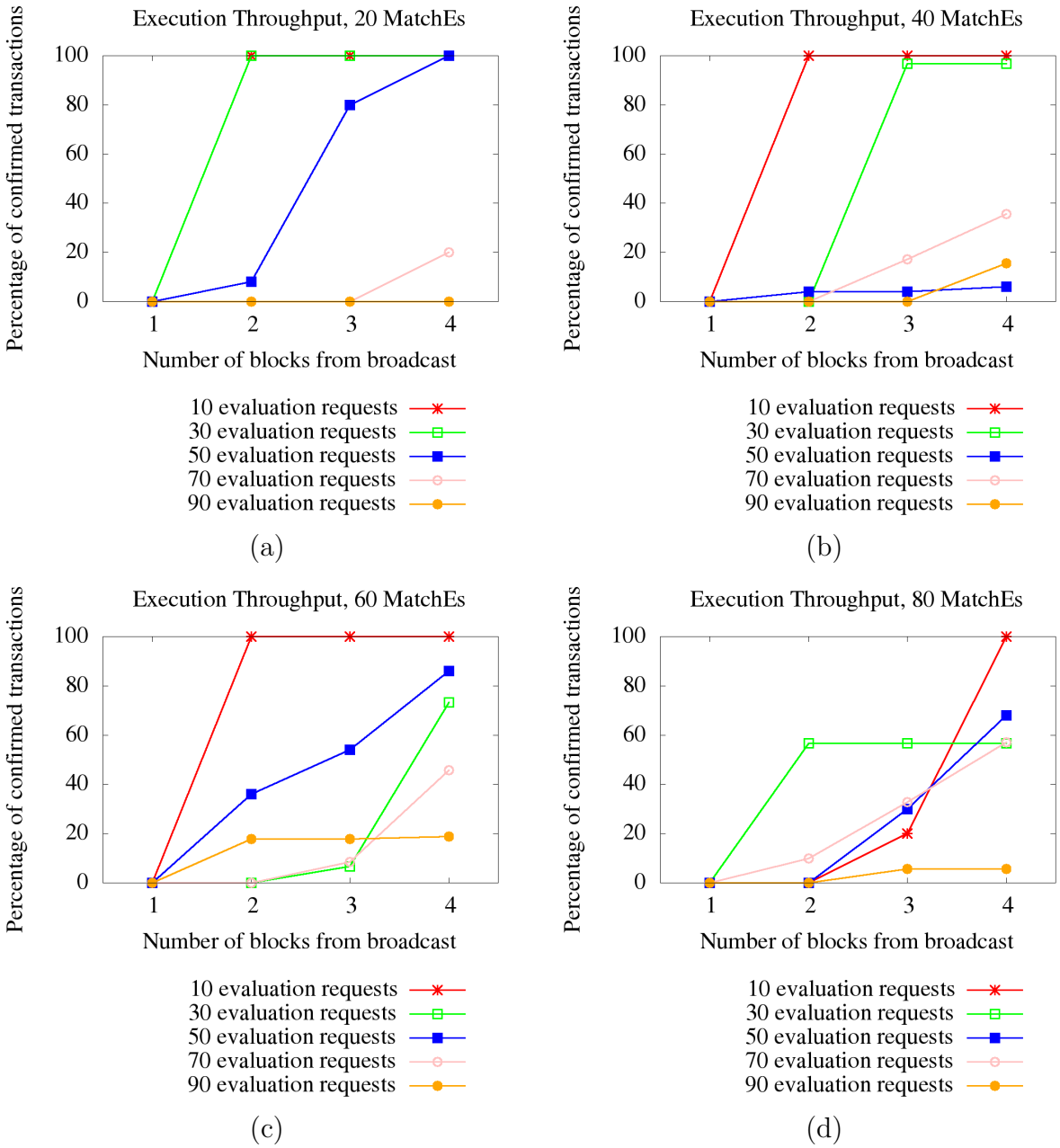

Figure 13: Average percentage of confirmed transactions in x blocks, for increasing number of x and increasing number of requests. Experiments regarding a reference SMART POLICY with 20 (a), 40 (b), 60 (c) or 80 (d) MATCHES on the Ropsten testnet.

in just one block, see Figure 8(b)).

7.4. Discussion

We would like to remark that the results presented in this section concern the evaluation of our proof of concept implementation of the reference example proposed. Such system has

been deployed on a specific testbed based on Ethereum, and the performance we achieved are heavily dependent on it. However, the aim of this work is to show how it could be possible to integrate traditional Access Control systems with blockchain technology and to analyze the resulting main advantages and drawbacks. This is why we would like to maintain a general point of view rather than focusing on a particular blockchain protocol.

If we take into account the performance aspect only, it is true that the currently available public blockchain systems are, in general, less efficient than traditional centralised ones in executing programs. As a matter of fact, the distributed consensus mechanism introduces an overhead w.r.t. centralised models where the system state updates are managed by a single (trusted) entity. Moreover, the replication of the shared state (i.e., the blockchain data) and the replication of new data validation across all the nodes determines an additional burden for the participants. For those reasons, implementing our system exploiting a public blockchain protocol is expected to result in worse performances in performing policy evaluation w.r.t. a traditional centralised Access Control system. In fact, for an access decision to be taken, at least one transaction needs to be mined. For example, in the Ethereum protocol, used by our proof of concept implementation, the average mining time to generate a new block is about 14 seconds (see Section 7.3 for an in-depth analysis concerning our reference implementation). However, different blockchain protocols have different expected mining times but, currently, they are typically in the order of seconds. This means that our system would be slower than traditional ones that could have, in general, decision times even in the order of milliseconds. As an example, the centralised Access Control system proposed in [64] for protecting OpenNebula based Cloud services takes about 22 milliseconds to perform the evaluation of a policy (called *pre-decision* phase) with 10 attributes managed by an AM located on the same machine of the Access Control system, and about 163 and 353 milliseconds to evaluate the same policy where the attributes are managed by, respectively, 2 and 5 AMs located on distinct remote machines. As such, our proposed Access Control system is a valid alternative to traditional ones only for those application scenarios where the additional properties provided by blockchain integration are desirable enough to cover for the diminished performances. This might not be the case for time sensitive scenarios. For example, a scenario in which our Access Control system can be successfully applied is the reference example we have chosen. In fact, if we adopt the proposed system to regulate the execution of smart contracts, then its performance should not be an issues because it is obviously comparable with the execution time of the protected smart contracts. Furthermore, w.r.t. traditional systems, we have also to take into account the fee paid for the execution of each transaction. As such, depending on the scenario in which our system is adopted, controlling the right of executing actions that are not security relevant could become too expensive. E.g., controlling the access to an office door has different security requirements and requests frequency compared to a bank vault door. The properties provided from our proposal do come at a price that is acceptable or not depending on the application scenario. Furthermore cheaper solutions (in terms of both resources and fees paid) can be employed, as already suggested in Section 6.1

The main limits to our system performance are determined by the actual blockchain protocol chosen for running it. We have chosen to implement our proof of concept on Ethereum

because it currently is, by far, the most used and studied smart contract enabling blockchain platform. Nevertheless, Ethereum has still some serious performance issues itself. For instance, despite its developers claim about a *world computer* [65], the CryptoKitties launch [66] was able to paralyze the entire Ethereum blockchain for days due to a single game application. We showed in Section 7.1.2, how a very complex policy can take up to 230.000 gas to be evaluated. This would mean that our system could process, in approximately the worst case supported, at most 20 evaluate request per block (a new block is created, on average, every 14 seconds), assuming a block gas limit of 4.700.000 and no other concurrent transactions. Instead, a policy with 10 MatchEs on 10 different attributes (more common in practice) would cost approximately 47.000 gas (see Figure 5), resulting in about 99 transactions per block. Such throughput could seem not sufficient for a real world application, but we should consider that under the previous assumption an Ethereum block could process at most 223 transactions of the simplest possible kind (i.e., fund transfers), and that common smart contracts function calls are much more expensive than that, so the real throughput is actually lower.

This is to remark how the performances of our current implementation are constrained by the blockchain protocol chosen, Ethereum, that is the most used one today (basically the reference protocol supporting smart contracts). Finally, we would like to highlight that the future launch of new and better performing blockchains, or any performance improvement of the Ethereum one, would be beneficial for our system automatically improving its performance (since the proposal is independent from the blockchain chosen).

8. Conclusions and Future Work

In this paper we presented our blockchain application proposal related to Access Control systems. Our main contribution is the integration of blockchain technology with a general Access Control systems to obtain a blockchain based Access Control system. Such novel system inherits the advantages of blockchain technology.

In our opinion the main advantage obtained is auditability. In fact, delegating the Access Control policies management and evaluation (through smart contracts) to the blockchain hands over such tasks to a decentralized, transparent, and immutable system. This means that resource owners can not fraudulently deny access to subjects without leaving an auditable trace of the misbehaviour. A deeper analysis of pros and cons is provided in Section 6.

Our main future work direction of research is split among different topics.

- First we plan to properly address the possible privacy issues of our proposal, starting from the options showed in Section 6.2. In fact, in order to provide complete auditability, our system stores all information publicly visible on the blockchain (either as simple data or stored inside the state of the smart contracts involved). It is worth studying which is the best method to mask such data while still allowing auditability, if necessary weakening it by providing it only to the parties involved.

- Another interesting research direction is expanding our work to encompass more Access Control standards and models. For example, we are studying the possibility to extend our approach in order to support advanced authorization models, such as the Usage Control one [10], and to deal with access history [67].

- Finally, we are currently implementing different proof of concept implementations on blockchain protocols different from Ethereum, in order to obtain a better insight on the performance of the derived system with different blockchains. Moreover we are investigating the possibility of developing a dedicated Access Control blockchain as well as the implications of applying our proposal to permissioned blockchain scenarios.

References

[1] U. Lang, OpenPMF SCaaS: Authorization as a service for cloud & SOA applications, in: Second International Conference on Cloud Computing (CloudCom 2010), 2010, pp. 634–643.

[2] R. Wu, X. Zhang, G. J. Ahn, H. Sharifi, H. Xie, ACaaS: Access control as a service for iaas cloud, in: International Conference on Social Computing, 2013, pp. 423–428.

[3] OASIS, eXtensible Access Control Markup Language (XACML) version 3.0 (January 2013).

[4] D. Di Francesco Maesa, P. Mori, L. Ricci, Blockchain based access control, in: IFIP International Conference on Distributed Applications and Interoperable Systems, Springer, 2017, pp. 206–220.

[5] D. Di Francesco Maesa, L. Ricci, P. Mori, Distributed access control through blockchain technology, Blockchain Engineering (2017) 31.

[6] D. Di Francesco Maesa, P. Mori, L. Ricci, Blockchain based access control services, in: IEEE International Symposium on Recent Advances on Blockchain and Its Applications (BlockchainApp), 2018 IEEE International Conference on Blockchain, IEEE, 2018.

[7] G. Wood, Ethereum: A secure decentralised generalised transaction ledger, Ethereum Project Yellow Paper 151.

[8] Ropsetn Revival, `https://github.com/ethereum/ropsten`. Retrieved June 30 2018.

[9] R. S. Sandhu, P. Samarati, Access control: principle and practice, Communications Magazine, IEEE 32 (9) (1994) 40–48.

[10] A. Lazouski, F. Martinelli, P. Mori, 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, pp. 79–92.

[11] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, C. E. Youman, Role-based access control models, Computer 29 (2) (1996) 38–47. doi:10.1109/2.485845.

[12] F. Vincent C. Hu, David, K. Rick, S. Adam, M. Sandlin, K. Robert, S. Karen, Guide to attribute based access control (ABAC) definition and considerations (2014).

[13] L. S. Sankar, M. Sindhu, M. Sethumadhavan, Survey of consensus protocols on blockchain applications, in: Advanced Computing and Communication Systems (ICACCS), 2017 4th International Conference on, IEEE, 2017, pp. 1–5.

[14] X. Xu, I. Weber, M. Staples, L. Zhu, J. Bosch, L. Bass, C. Pautasso, P. Rimba, A taxonomy of blockchain-based systems for architecture design, in: Software Architecture (ICSA), 2017 IEEE International Conference on, IEEE, 2017, pp. 243–252.

[15] C. Cachin, Architecture of the hyperledger blockchain fabric, in: Workshop on Distributed Cryptocurrencies and Consensus Ledgers, Vol. 310, 2016.

[16] M. Iwamura, Y. Kitamura, T. Matsumoto, Is bitcoin the only cryptocurrency in the town? economics of cryptocurrency and friedrich a. hayek.

[17] S. Nakamoto, Bitcoin: A peer-to-peer electronic cash system (2008).

[18] K. Wüst, A. Gervais, Do you need a blockchain?, IACR Cryptology ePrint Archive 2017 (2017) 375.

[19] V. Buterin, et al., A next-generation smart contract and decentralized application platform, white paper.
[20] F. Reid, M. Harrigan, An analysis of anonymity in the bitcoin system, in: Security and privacy in social networks, Springer, 2013, pp. 197–223.
[21] Ethash - Ethereum Wiki, `https://github.com/ethereum/wiki/wiki/Ethash`. Retrieved June 30 2018.
[22] Solidity documentation, `https://solidity.readthedocs.io/en/develop/`. Retrieved June 30 2018.
[23] G. Zyskind, O. Nathan, et al., Decentralizing privacy: Using blockchain to protect personal data, in: Security and Privacy Workshops (SPW), 2015 IEEE, IEEE, 2015, pp. 180–184.
[24] E. Kokoris-Kogias, E. C. Alp, S. D. Siby, N. Gailly, P. Jovanovic, L. Gasser, B. Ford, Hidden in plain sight: Storing and managing secrets on a public ledger, Tech. rep., Cryptology ePrint Archive: 209 https://eprint. iacr. org/2018/209. pdf (2018).
[25] H. Shafagh, L. Burkhalter, A. Hithnawi, S. Duquennoy, Towards blockchain-based auditable storage and sharing of iot data, in: Proceedings of the 2017 on Cloud Computing Security Workshop, ACM, 2017, pp. 45–50.
[26] J. P. Cruz, Y. Kaji, N. Yanai, Rbac-sc: Role-based access control using smart contract, IEEE Access 6 (2018) 12240–12251.
[27] A. Ouaddah, H. Mousannif, A. A. Elkalam, A. A. Ouahman, Access control in the internet of things: big challenges and new opportunities, Computer Networks 112 (2017) 237–262.
[28] A. Dorri, S. S. Kanhere, R. Jurdak, P. Gauravaram, Blockchain for iot security and privacy: The case study of a smart home, in: Pervasive Computing and Communications Workshops (PerCom Workshops), 2017 IEEE International Conference on, IEEE, 2017, pp. 618–623.
[29] A. Ouaddah, A. Abou Elkalam, A. Ait Ouahman, Fairaccess: a new blockchain-based access control framework for the internet of things, Security and Communication Networks 9 (18) (2016) 5943–5964.
[30] Y. Zhang, S. Kasahara, Y. Shen, X. Jiang, J. Wan, Smart contract-based access control for the internet of things, arXiv preprint arXiv:1802.04410.
[31] C. Dukkipati, Y. Zhang, L. C. Cheng, Decentralized, blockchain based access control framework for the heterogeneous internet of things, in: Proceedings of the Third ACM Workshop on Attribute-Based Access Control, ACM, 2018, pp. 61–69.
[32] R. Xu, Y. Chen, E. Blasch, G. Chen, Blendcac: A blockchain-enabled decentralized capability-based access control for iots, arXiv preprint arXiv:1804.09267.
[33] N. Tapas, G. Merlino, F. Longo, Blockchain-based iot-cloud authorization and delegation, in: 2018 IEEE International Conference on Smart Computing (SMARTCOMP), IEEE, 2018, pp. 411–416.
[34] I. Sukhodolskiy, S. Zapechnikov, A blockchain-based access control system for cloud storage, in: Young Researchers in Electrical and Electronic Engineering (EIConRus), 2018 IEEE Conference of Russian, IEEE, 2018, pp. 1575–1578.
[35] S. Wang, Y. Zhang, Y. Zhang, A blockchain-based framework for data sharing with fine-grained access control in decentralized storage systems, IEEE Access 6 (2018) 38437–38450.
[36] A. Azaria, A. Ekblaw, T. Vieira, A. Lippman, Medrec: Using blockchain for medical data access and permission management, in: Open and Big Data (OBD), International Conference on, IEEE, 2016, pp. 25–30.
[37] H. Es-Samaali, A. Outchakoucht, J. Leroy, A blockchain-based access control for big data, International Journal of Computer Networks and Communications Security 5 (7) (2017) 137–147.
[38] J. P. Dias, L. Reis, H. S. Ferreira, Â. Martins, Blockchain for access control in e-health scenarios, arXiv preprint arXiv:1805.12267.
[39] C. Svenson, Blockchain: Using cryptocurrency with java, Java Magazine, January/February (2017) 36–46.
[40] Open Blockchain initiative, `http://blockchain.open.ac.uk/`. Retrieved June 30 2018.
[41] Ropsten Ethereum Testnet, `https://ropsten.etherscan.io/`. Retrieved June 30 2018.
[42] Rinkeby Ethereum Testnet, `https://github.com/ethereum/EIPs/issues/225`. Retrieved June 30 2018.

[43] Kovan Ethereum Testnet, https://github.com/kovan-testnet/proposal. Retrieved June 30 2018.
[44] geth client, https://github.com/ethereum/go-ethereum/wiki/geth. Retrieved June 30 2018.
[45] Solidity 0.4.24 documentation, Frequently Asked Questions, https://solidity.readthedocs.io/en/v0.4.24/frequently-asked-questions.html. Retrieved June 30 2018.
[46] Solidity compiler releases, https://github.com/ethereum/solidity/releases. Retrieved June 30 2018.
[47] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, J. Yellick, Hyperledger fabric: A distributed operating system for permissioned blockchains, in: Proceedings of the Thirteenth EuroSys Conference, EuroSys '18, ACM, New York, NY, USA, 2018, pp. 30:1–30:15.
[48] S. Yoshihama, S. Saito, Study on integrity and privacy requirements of distributed ledger technologies, in: 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, IEEE Computer Society, 2018, pp. 1657–1664.
[49] D. Ron, A. Shamir, Quantitative analysis of the full bitcoin transaction graph, in: Financial Cryptography and Data Security - 17th International Conference, FC 2013, Okinawa, Japan, April 1-5, 2013, Revised Selected Papers, 2013, pp. 6–24.
[50] S. Meiklejohn, M. Pomarole, G. Jordan, K. Levchenko, D. McCoy, G. M. Voelker, S. Savage, A fistful of bitcoins: characterizing payments among men with no names, in: Proceedings of the 2013 Internet Measurement Conference, IMC 2013, Barcelona, Spain, October 23-25, 2013, 2013, pp. 127–140.
[51] D. Di Francesco Maesa, A. Marino, L. Ricci, Detecting artificial behaviours in the bitcoin users graph, Online Social Networks and Media 3 (2017) 63–74.
[52] C. Gentry, A fully homomorphic encryption scheme, Stanford University, 2009.
[53] M. Van Dijk, C. Gentry, S. Halevi, V. Vaikuntanathan, Fully homomorphic encryption over the integers, in: Annual International Conference on the Theory and Applications of Cryptographic Techniques, Springer, 2010, pp. 24–43.
[54] E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, M. Virza, Zerocash: Decentralized anonymous payments from bitcoin, in: Security and Privacy (SP), 2014 IEEE Symposium on, IEEE, 2014, pp. 459–474.
[55] I. Miers, C. Garman, M. Green, A. D. Rubin, Zerocoin: Anonymous distributed e-cash from bitcoin, in: Security and Privacy (SP), 2013 IEEE Symposium on, IEEE, 2013, pp. 397–411.
[56] J. Morgan, Quorum whitepaper, Available: https://github.com/jpmorganchase/quorum-docs/blob/master/Quorum Whitepaper v0.1.pdf 1.
[57] Quorum - ZSL Integration: Proof of Concept, retrieved 30 Dec 2018, https://github.com/jpmorganchase/zsl-q/blob/master/docs/ZSL-Quorum-POC_TDD_v1.3pub.pdf.
[58] A. Unterweger, F. Knirsch, C. Leixnering, D. Engel, Lessons learned from implementing a privacy-preserving smart contract in ethereum, in: New Technologies, Mobility and Security (NTMS), 2018 9th IFIP International Conference on, IEEE, 2018, pp. 1–5.
[59] A. Kosba, A. Miller, E. Shi, Z. Wen, C. Papamanthou, Hawk: The blockchain model of cryptography and privacy-preserving smart contracts, in: 2016 IEEE symposium on security and privacy (SP), IEEE, 2016, pp. 839–858.
[60] E. Barka, R. Sandhu, A role-based delegation model and some extensions, in: Proc. 23rd National Information Systems Security Conference, 2000, pp. 101–114.
[61] Q. Wang, N. Li, H. Chen, On the security of delegation in access control systems, in: S. Jajodia, J. Lopez (Eds.), Computer Security - ESORICS 2008, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 317–332.
[62] MetaMask, https://metamask.io/. Retrieved June 30 2018.
[63] Etherscan Ethereum Main Chain, https://etherscan.io/. Retrieved June 30 2018.
[64] E. Carniani, D. D'Arenzo, A. Lazouski, F. Martinelli, P. Mori, Usage control on cloud systems, Future Generation Comp. Syst. 63 (2016) 37–55.

[65] What is Ethereum? - Ethereum Frontier Guide, retrieved 30 Dec 2018, https://ethereum.gitbooks.io/frontier-guide/content/ethereum.html.

[66] Loveable Digital Kittens Are Clogging Ethereum's Blockchain - CoinDesk, retrieved 30 Dec 2018, https://www.coindesk.com/loveable-digital-kittens-clogging-ethereums-blockchain.

[67] F. Martinelli, P. Mori, Enhancing java security with history based access control, in: Foundations of Security Analysis and Design IV, FOSAD 2006/2007 Tutorial Lectures, Vol. 4677 of Lecture Notes in Computer Science, Springer, 2007, pp. 135–159.