# Decentralised Runtime Monitoring for Access Control Systems in Cloud Federations

Md Sadek Ferdous, Andrea Margheri, Federica Paci, Mu Yang, Vladimiro Sassone
University of Southampton
{s.ferdous; a.margheri; f.m.paci; mu.yang; vsassone}@soton.ac.uk

***Abstract*—Cloud federation is an emergent cloud-computing paradigm where partner organisations share data and services hosted on their own clouds platforms. In this context, it is crucial to enforce access control policies that satisfy the data protection and privacy requirements of the partner organisations. However, due to the distributed nature of cloud federations, the access control system alone does not guarantee that its deployed components cannot be circumvented while processing access requests. Therefore, in order to promote accountability and transparency of access control decisions in federated clouds, we present a decentralised runtime monitoring architecture based on a blockchain technology. The logging components and data of the proposed infrastructure are deployed on the blockchain. This guarantees that the runtime monitoring components and the access logs cannot be compromised by malicious users to disguise their actions. We evaluate the performance of the runtime monitoring infrastructure with respect to detecting policy violations, and the cost of deploying the logging components and data on the blockchain.**

***Index Terms*—Access Control, Cloud Federation, Runtime Monitoring, Blockchain, Security.**

## I. Introduction

The advent of cloud computing has enabled new collaborative scenarios in which users and organisations share resources, information and services in order to achieve a common goal or interest. An instance of this trend is provided by federated clouds [1], [2], [3]. These are cloud systems created dynamically to achieve a business goal, such as sharing computational resources and/or data. Resource sharing can however be hindered by security and privacy concerns: organisations which partner in a federation will consider who can access their shared resources, for what purposes, and what are the potential consequences of granting access.

An approach typically adopted to address these concerns is to deploy a federation-wide access control system to enforce access control policies attached to the federation by the resource owner [4]. This means that there will be distributed components that receive, exchange and process access requests and their corresponding access decisions. However, in this distributed setting, it is possible that these components are circumvented, e.g. an exchange message may be subverted or a component violated.

To prevent such attacks, this paper proposes a runtime monitoring solution for distributed access control systems: *Decentralised Runtime Access Monitoring System* (DRAMS). DRAMS enables runtime monitoring of federated access control policies by including distributed logging probes which sense access control activities and intercept access requests and decisions. These logs are then processed to check the integrity of the monitored components. Differently from classical (centralised) applications, DRAMS copes with the distributed nature of federated clouds: multiple components deployed as part of different computing systems, that interact together to enforce an access decision. Specifically, there should be an adequate infrastructure that allows logs collected on distributed components to be stored with the consensus of all probes. Hence, the logs can be monitored and appropriate guarantees on the monitoring checks can be provided to the involved federated clouds, i.e., they must have non-repudiable evidence on the integrity of the checks.

To implement this monitoring system, we build DRAMS upon the access control system of Federation-as-a-Service (FaaS) [3], a recently proposed approach to cloud federation devised and developed by the H2020 project SUNFISH [5]. Indeed, we put forward the first application of *blockchain* technologies [6] as an infrastructure for storing logs and perform non-repudiable monitoring checks. Blockchain is a novel technology that, besides its application to cryptocurrency, features fascinating properties of data integrity, distribution and control. We rely on a blockchain infrastructure to enjoy these properties while collecting and evaluating access control logs. More specifically, we utilise an advanced blockchain system based on so-called *smart-contracts*, which are arbitrarily complex programs deployed and executed autonomously on a blockchain. This approach is our key to guarantee the integrity and availability of the stored logs.

To evaluate our approach, we have deployed a proof-of-concept implementation on top of *Ethereum* [7], the state-of-the-art technology for distributed ledgers (blockchain) smart-contracts. Specifically, we consider a simplified cloud federation setting and multiple users carrying out different access strategies on federated resources. The evaluation results indicate that our approach is able to detect every policy violation.

*Contribution.* This paper presents the first adoption of a blockchain-based system in the design and implementation of a runtime monitoring system. We report here the architectural design, implementation strategy and performance evaluation of the prototype implementation of DRAMS, a distributed runtime monitoring system for federated cloud access control, and discuss the security assurances it provides.

*Structure of the paper.* Section II introduces background

concepts. Section III comments on the adversary model for the access control system. Section IV presents the DRAMS architecture along with the underlying monitoring protocols and checks. The proof-of-concept implementation is described in Section V, as well as its performance evaluation. Section VI discusses on the DRAMS approach, while Section VII reviews related work. Section VIII reports concluding remarks and touches upon future work.

## II. Background

This section introduces the access control language XACML (Section II-A), the access control system of a FaaS cloud federation (Section II-B) and blockchain technology (Section II-C).

### A. XACML

The eXtensible Access Control Markup Language (XACML) [8] is the de-facto standard for attribute-based access control [9]. This type of access controls are based on *attributes*, i.e. arbitrary security-relevant information exposed by the system, the involved subjects, the action to be performed, or by any other entity of the evaluation context relevant to the rules at hand. The attribute-base model permits defining fine-grained, flexible and context-aware access control rules that are expressive enough to uniformly represent all previous access control models [10], e.g. the well-known role-based one [11].

XACML consists of an XML language for access control policies and a structured evaluation process for their enforcement. Each policy enlists the controls on the attribute values to satisfy to gain the access to protected resources. These controls are hierarchical structures of positive and negative access rules, which are combined together by means of strategies for resolving possible conflicting access decisions (e.g., both positive and negative access rules apply to an access request).

The XACML policy evaluation process mainly involves the following components

- *Policy Decision Point* (PDP): it calculates an access decision for a user's request based on the available policies.
- *Policy Enforcement Point* (PEP): it enforces the decision calculated by the PDP in the system.

Indeed, once PEP receives a user's request, it forwards it to the PDP, which calculates the access decision. The decision is then enforced by the PEP.

The evaluation process carried out by the PDP relies, on the one hand, on contextual information provided by the *Policy Information Point* (PIP). On the other hand, it is based on the access control policies made available by the *Policy Retrieval Point* (PRP) and administered by the *Policy Administration Point* (PAP). This modularised architecture allows XACML to be exploited within any application domain.

The decision resulting from the policy evaluation process can be one among `permit`, `deny`, `not-applicable` and `indeterminate`[1]. The meaning of the first two ones is obvious, the third one means that there is no policy that applies to the request and the latter one means that some errors have occurred during the evaluation. Policies can automatically manage these errors by using the policy combing strategies.

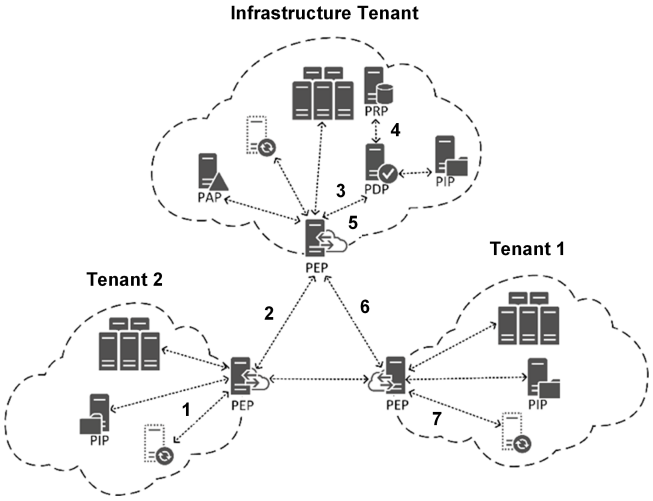

Figure 1: Distributed Access Control System of FaaS

### B. Access Control Systems for Cloud federations

Cloud federation is a recent solution to prompt aggregation and cooperation of cloud systems [1], [2]. In this context, the H2020 project SUNFISH [5] devised and developed an innovative cloud federation solution, Federation-as-a-Service (FaaS) [3]. FaaS offers a service that allows clouds to create and manage cloud federations. To securely managing federated data and inter-cloud interactions, FaaS includes the distributed access control system based on XACML introduced in [12] and shown in Figure 1.

The access control system is deployed along with the tenants (i.e., virtual spaces of computing resources belonging to different clouds) underlying a FaaS federation. The PDP and the policy management is placed in the infrastructural tenant (i.e., the tenant owned by all federation clouds that enabled the FaaS functionalities). PEP is instead deployed in a distributed manner on the tenant edge, thus to intercept all communications, interact with the distributed sources of information (i.e., the distributed PIP), and enforce the calculated accesses.

This architecture enables a user from any tenant to access services hosted in another tenants in a secure and controlled manner. It indeed enforces the access control protocol defined by the numbers on the arrows of Figure 1. Specifically, when a service consumer from Tenant 2 requests a service from Tenant 1 (step 1), the local PEP forwards (by possibly adding additional attributes via local interactions with PIP) to the PEP

[1]For the cognitive, a PDP evaluation process complaint with XACML relies on specialised `indeterminate` decisions, so-called *extended indeterminate*. However, any specialised `indeterminate` returned by the PDP is converted to `indeterminate` before being sent to the PEP; hence we retain to the single `indeterminate` decision.

of the Infrastructural Tenant the access request to authorise (step 2). The request is then forwarded to the PDP (step 3) that, on the base of the policies currently in force (step 4), calculates an access decision; the latter evaluation can require additional interactions with the local PIP. The access decision is then sent to the PEP (step 5) and, if positive, forwarded to the PEP of Tenant 1 (step 6) to enforce the allowed access (step 7).

### C. Blockchain and Smart-Contract

Blockchain is a novel technology that has appeared on the market in recent years. It was firstly used as a public ledger for the Bitcoin cryptocurrency [6]. It consists of consecutive chained blocks, replicated and stored by the nodes of a peer-to-peer network, where blocks are created in a decentralised fashion by means of a consensus algorithm called Proof-of-Work (PoW). The use of PoW enables several data integrity related properties in blockchain, such as distributed and democratic control of the data on the chain, distributed consensus on the chain state, and non-repudiation and persistency of transactions and data provenance.

Blockchains can be adapted to different needs, indeed, there can be multiple deployment strategies to pursue; mainly we have: *public*, i.e. an unpermissioned system open to any user, and *private*, i.e. a permissioned system with control on operating users. Depending on the blockchain type, parameters of PoW can be tuned accordingly. For instance, the higher the difficulty of the hashing procedure is, the higher the time to obtain integrity guarantees is.

Differently from Bitcoin, new types of blockchains have recently appeared featuring *smart-contract*s, that is, programs deployed and executed on blockchain. Being part of the blockchain makes contracts and their executions *immutable* and *irreversible*. Smart-contract permits creating so-called *decentralised applications* (ĐApps), i.e. applications that operate autonomously and without any control by a system entity. More specifically, a ĐApp is configured as a web server exposing APIs to invoke smart-contract on blockchain.

The state-of-the-art smart-contract blockchain is Ethereum [7]. It features a virtual machine, called Ethereum virtual machine (EVM), which allows smart-contracts to be deployed and executed. Most of all, EVM executes contracts according to an amount of *Ether* (i.e., the Ethereum cryptocurrency) given in input. As the execution of every contract action costs a certain Ether amount, it follows that, on the one hand, the termination of contract executions is always guaranteed and, on the other hand, that to operate on the public Ethereum it is currently required to pay around 9.6USD per Ether[2]. Additionally, Ethereum supports an event-based mechanism that, via specific interfaces, allows smart-contracts to raise an event signaling to a ĐApp that a certain action, e.g. the storing of data, has been carried out in the smart-contract.

[2]As per http://ether.price.exchange/ on 16 January, 2017

## III. Adversary Model

In this section, we model the capabilities of the adversary facing the access control system. To accomplish an attack, the adversary can either directly attack the access control system or violate the monitoring system.

The access control system of FaaS is mainly based on the communication protocol previously presented. Therefore, to model the attacker, we consider the well-known Honest-But-Curious [13] adversary model. Indeed, an attacker participates in the protocol and behaves correctly. But, he can intercept, send (resp., receive) any message to (resp., from) the protocols to which it participates. Additionally, an attacker can have the capabilities to violate the integrity of the Infrastructural Tenant where the PDP is located. Specifically, it can subvert the PDP evaluation process to return a different decision with respect to the semantics of the policy currently in force.

Based on these described attack capabilities, we list the potential threats for federated cloud access control systems.

**T1** *Compromised communication channel:* access requests and decisions exchanged between the access control components can be intercepted and modified.

**T2** *Compromised Policy Evaluation:* the PDP evaluation process is violated that, e.g., it always returns a specific decision.

According to the access control protocol in Figure 1, threat **T1** refers to the violation of the communication channels allowing the interactions of steps 2 and 6. For the sake of presentation, we assume that the intra-tenant communications between PEP and PDP, i.e. steps 3 and 5, cannot be compromised. (This assumption is feasible in the FaaS context, because the communication within the Infrastructural Tenant relies on secured VPN tunnels.) Instead, the threat **T2** refers to the PDP evaluation process that generates the access decision input of step 5.

The latter threat assumes that only the evaluation engine can be violated, while the evaluated policies cannot be. This simplifying assumption follows from the FaaS storage means for access control policies: their integrity in ensured by-design via a blockchain-based infrastructure (see [3] for further details). It is also worth noticing that we have only considered threats on the integrity of the PDP for the sake of presentation. Threats on other components, e.g. the PIP, can be easily subsumed under the PDP ones.

The runtime monitoring system is thus in charge of mitigating these risks. However, the monitoring system itself can be prone to attacks that could jeopardise the whole cloud federation. As the monitoring system does not rely on complex protocols, an attacker may directly focus on the monitoring architecture to compromise evidences or to affect service availability. Therefore, we have the following threats.

**T3** *Compromised logging system:* the monitoring system that stores logs can be compromised, e.g., an access log is removed or forged.

**T4** *Denial-of-Service:* the architecture can be the object of a Denial-of-Service attack that prevents the availability of the monitoring service.

Addressing the latter threats will then ensure that the monitoring service is reliable and available.

## IV. Runtime Access Control Monitoring

In this section, we introduce *Decentralised Runtime Access Monitoring System* (*DRAMS*), the runtime monitoring for cloud federation access control systems. The proposed system rests on a smart-contract blockchain to store logs and perform monitoring checks on them. Additionally, DRAMS is equipped with a formally-grounded policy analyser that permits evaluating whether an access decision is correct according to the semantics of the policies currently in force in the system.

In the following, we first present the architectural components of DRAMS (Section IV-A), then the pursued monitoring protocol (Section IV-B) and the monitoring checks mitigating the reported threats (Section IV-C).

### A. Architecture

DRAMS is a decentralised monitoring system formed by the following architectural components

- *Probing agents*: they are responsible for intercepting and forwarding data to create access logs.
- *Smart-contract blockchain*: it is the smart-contract blockchain system storing and comparing logs gathered by probing agents.
- *Logging Interface*: it is the ÐApp exposing the endpoints to the probing agents to store intercepted data and to manage events generated by the smar-contracts.
- *Analyser*: it checks the correctness of the access decisions calculated for the intercepted requests, with respect to the policies currently in force in the system.

Indeed, the key element of the system is the blockchain infrastructure: it is connected via the Logging Interface to all the other components. The agents are distributed on the monitored components of the access control system. According to the considered threats, probes are placed on top of PDP and PEP components.

Besides offering endpoints to access the blockchain, the Logging Interface also provides symmetric encryption and decryption functions, which are exploited by the other components to store/retrieve encrypted data in/from smart-contracts. Indeed, as data stored on a blockchain are visible to all users, encryption is used to protect the data confidentiality and, at the same time, to maintain plain data to use as inputs to semantic checks on the system.

The Analyser is a standalone entity that dynamically consumes and evaluates the gathered logs to ensure the correct enforcement of access decisions. On the base of a logical representation of the access control policies evaluated by the PDP, the Analyser checks if for a given request the calculated response is the expected one. This check has to be specialised according to the access control policies used by the monitored system, i.e. XACML.

To this aim, there are available in the literature a few full-fledged analysis frameworks for XACML [14], [15], [16]. Differently from others, the framework in [16] enjoys the benefits of using formal method techniques to model and analyse XACML policies. The pursued approach rests on FACPL, a formal language to represent XACML, and on its translation procedure to Satisfiability Modulo Theory (SMT) constraints [17]. Therefore, the Analyser translates the policies currently in force to SMT constraints and evaluates their satisfiability according to monitored requests and responses. Further details are reported in the following sections.

Figure 2 presents the DRAMS architecture deployed on top of the FaaS access control system of Figure 1. Specifically, there is an agent for each tenant where the monitored access control components (i.e., PDP and PEP) are placed. To enable the storing of logs in the blockchain, a Logging Interface is placed in each tenant. The Analyser is logically placed within the Infrastructural Tenant, but it is deployed within a different cloud section (i.e., a set of computing resources belonging to cloud) with respect to the access control components. The Analyser relies on a Logging Interface to gather and analyse the access logs.

### B. Monitoring Protocol

The runtime monitoring of DRAMS transparently intervenes in the access control protocol of FaaS. Specifically, with respect to Figure 1, the system monitors access requests and responses exchanged during steps 2 and 6. Due to the assumption of secure intra-communications, we assume that step 2 directly occurs between the PEP of Tenant 2 and the PDP; similarly, step 6 occurs between the PDP and the PEP of Tenant 1. Hence, we abstract from the PEP of the Infrastructural Tenant; anyway its violation is subsumed by that of the PDP.

As a matter of notation, we use *req* (resp., *res*) to range over requests (resp., responses) and the subscript *pep* (resp., *pdp*) to denote that the request/response is sent/received by PEP (resp., PDP). For instance, $res_{pdp}$ refers to the response generated by the PDP.

The monitoring protocol is reported in Figure 3. To ease the presentation, we only report the monitoring steps, which are triggered by both the sending and receiving of the monitored interactions (recall, steps 5 and 6 of Figure 1). Therefore, the single authorisation of an access request generates four monitoring activities.

We comment the steps of Figure 3 in the following.

When a triggering communication activity occurs (step 1), the corresponding probing agent intercepts the communication and creates an initial access log (step 2). This log is defined by the following tuple

$$ac\_agent \triangleq \langle\ reqId,\ type,\ text\ \rangle$$

where *reqId* is the identifier of the monitored request (or its associated response), *type* is one among $\{req_{pdp}, req_{pep}, res_{pdp}, res_{pep}\}$, and *text* corresponds to the plain text of the intercepted XACML request (or response).

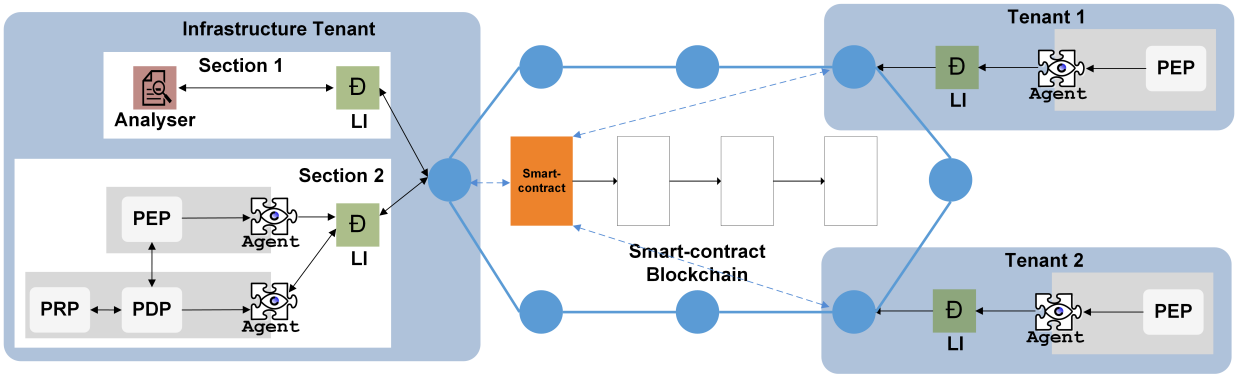


Figure 2: DRAMS architecture deployed on the access control system of a FaaS cloud federation (where 'Section i' stands for a set of computing resources belonging to a cloud 'i', while LI stands for Logging Interface)

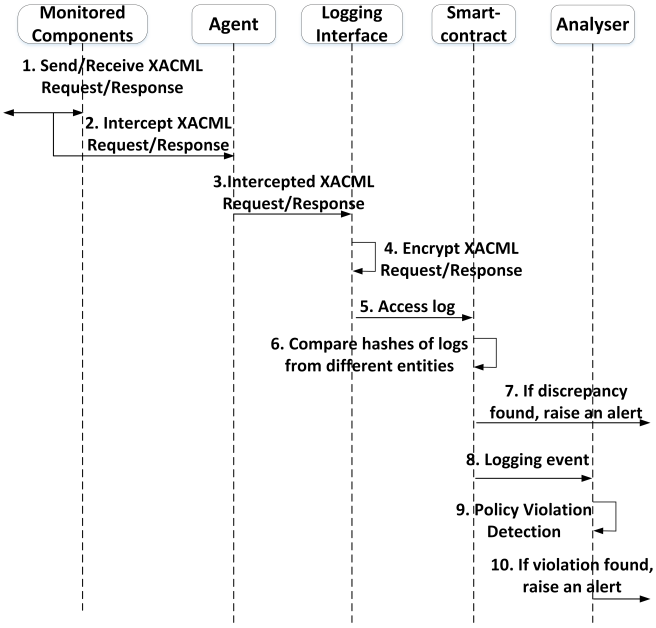


Figure 3: Monitoring Protocol

The generated log *ac_agent* is then sent to the Logging Interface (step 3). The latter adds a timestamp and performs the hasing and encryption of the received plain text (step 4). The resulting access log, ready to be stored on the blockchain, is thus defined by the following tuple

$$ac\_LI \triangleq \langle\ reqId,\ type,\ n,\ hash,\ \{text\}_K\ \rangle$$

where $n$ is the timestamp, *hash* is the hash value generated by applying the SHA-256 hashing function to *text*, and $\{text\}_K$ is the encrypted text. It is worth noticing that *text* is stored in the form of both hash and encrypted data to enable both on-chain monitoring checks (i.e., the smart-contract checks) and off-chain checks on the access control policy semantics (i.e., Analyser checks).

The access log tuples are then stored on the blockchain smart-contract (step 5). Specifically, the smart-contract groups the four logs generated by an access request according to its identifier *reqId*[3]. The access control logs maintained by the smart-contract are thus as follows

$$ac \triangleq \langle reqId\ \ \texttt{pdpReq} : ac\_LI_1\ \ \texttt{pepReq} : ac\_LI_2\ \ \texttt{pdpRes} : ac\_LI_3\ \ \texttt{pepRes} : ac\_LI_4\ \rangle$$

where tuple fields are named according to the type of $ac\_LI_i$.

When all the four access logs $ac\_LI_i$ are stored, the smart-contract checks the hashes (step 6) and, if a violation has been detected, an alert is generated (step 7). Similarly, by exploiting the event mechanism of the blockchain, the smart-contract informs the Analyser about a new access log (step 8). In its own turn, the Analyser checks the correctness of the calculated response (step 9) and possibly generates a security alert (step 10). Therefore, the monitoring check at step 6 deals with threat **T1**, while that at step 9 deals with threat **T2**. Both checks are defined in the next section.

Finally, notice that we address only two interactions for the sake of presentation, additional flows comprising multiple entities (e.g., the interactions intra-tenant with the PIP) could be monitored similarly.

### C. Monitoring Checks

In the following, we describe the checks performed during the monitoring protocol to address the considered threats.

*1) T1 - Compromised Communication Channel:* Relatively to threat **T1**, the blockchain smart-contract performs the comparison check defined by Algorithm 1. Specifically, given in input a smart-contract monitoring tuple *ac*, it checks if the sensed PDP and PEP requests correspond. To this aim, it accesses the tuples $ac\_LI_i$ via the corresponding field label, e.g. `pdpReq` to retrieve $ac\_LI_1$. Then, it retrieves via the auxiliary function *getHash()* the corresponding hash value and compares them each other. If the hashes do not correspond, a violation of the communication channel has occurred and a security alert is raised.

[3]Request identifiers are uniquely generated by-design by the access control system.

**Algorithm 1** Access Logs Comparison Algorithm

**Input:** a smart-contract logging tuple *ac*
**Output:** raise an alert if a discrepancy is found;
1: **if** *ac*.pdpReq.$getHash()$ != *ac*.pepReq.$getHash()$ **then**
2: raise an alert
3: **end if**
4: **if** *ac*.pdpRes.$getHash()$ != *ac*.pepRes.$getHash()$ **then**
5: raise an alert
6: **end if**
7: **return**

*2) T2 - Compromised Policy Evaluation:* Relatively to threat **T2**, the Analyser checks if the responses of an access log tuple *ac* are semantically correct with respect to the given request and the policy in force. To this aim, the SMT-based policy representation is exploited.

Given $p$ the policy currently in force[4], its SMT-based representation corresponds to a 4-tuples of SMT constraints, one for each possible XACML decision (see Section II-A). The tuple is thus defined as follows

$$\langle \texttt{permit} : c_p \qquad \texttt{deny} : c_d$$
$$\texttt{not-applicable} : c_n \qquad \texttt{indeterminate} : c_i \rangle$$

where $c$ is an SMT constraint. As formally proved in [16], these constraints ensure that the corresponding FACPL (and, hence, XACML) policy evaluates an request to a certain decision *dec* if and only if the constraint $c$ corresponding to *dec* is satisfiable with respect to the request.

Therefore, once the Analyser receives the sensed request *req* and its response *res*, it first translates *req* into a set of SMT assertions, say $c_{req}$, modelling the attributes forming the requests. Then, it chooses the SMT policy constraint corresponding the access decision *dec* reported in *res*. Being the latter constraint $c_{dec}$, the Analyser checks the satisfiability of the following SMT assertion

$$c_{dec} \ \wedge \ c_{req}$$

If the assertion is satisfiable, the access decision calculated by the PDP is correct, hence the PDP has not been violated. Otherwise, a security alert is raised.

## V. Implementation

To testify the effectiveness of DRAMS, we have implemented an early prototype of all its components and deployed them on a simplified cloud federation scenario.

More specifically, the considered federation consists of two tenants, $T_1$ and $T_2$: $T_1$ offers three federated services $S_1$, $S_2$ and $S_3$; $T_2$ plays the role of the infrastructure tenant. Consequently, the access control system consists of two PEPs, one for each tenant, and of a PDP and PRP in tenant $T_2$. The DRAMS components are deployed as in Figure 2: a probing agent and a LI in each tenant, the Analyser in tenant $T_2$.

[4]Notably, due to the root-based structure of the XACML evaluation process, we can always assume that only a single policy is in force at a moment. For the cognitive, a PDP is compliant with XACML only if it features a top-level combining algorithm composing the available policies into a single policy set.

Table I: Machine Configurations for tenant simulation (*graphics card details reported due to the GUI-based PoW algorithm*)

| Tenant | Configuration |
|---|---|
| Tenant 1 | Intel Core i7 Processor i7-4790 3.60GHz, Quad Core with 8MB Cache, 32GB DDR3 Memory, AMD Radeon R390 8GB PCIe Graphics card, 500GB Serial ATA Hard Drive |
| Tenant 2 | Intel Core i7 Processor i7-4790 3.60GHz, Quad Core with 8MB Cache, 32GB DDR3 Memory, AMD Radeon R7 240 2GB PCI Express graphics card, 500GB Serial ATA Hard Drive |

On top of this setting, we have then conducted several experiments to assess the feasibility of deploying DRAMS on an Ethereum blockchain. The hardware configuration of the two machines simulating the tenants is reported in Table I.

In the following, we first report the implementation details (Section V-A), then further information on the considered test-case (Section V-B), and finally the experiment results (Section V-C).

### A. Implementation Details

First of all, to implement DRAMS, we need a XACML implementation of PDP and PEP components, thus to regulate accesses to the considered services. To this aim, we have used WSO2 Balana [18] to implement the PDP (and its PRP) at $T_2$, while the PEPs, as well as the three shared services, are implemented by using the Go programming language [19].

Upon this access control infrastructure, we can thus implement the components of DRAMS. The probing agents are implemented still by using Go, while the smart-contract blockchain and the Logging Interface relies on the Ethereum toolset. More specifically, we have created a private Ethereum network formed by two nodes, one for each tenant, that also act as miners, i.e. they do PoW. Instead, the Logging Interface is implemented by means of Node.js [20], a server-side JavaScript platform, and Ethereum *web3.js*, the adapter to interact with blockchain smart contracts via a web server.

To perform the monitoring checks, Algorithm 1 has been implemented in terms of a smart-contract written in Solidity, i.e. a programming language for Ethereum. Listing 1 reports the main excerpt of the smart-contract code. The contract represents *ac_LI* logs, i.e. the single log produced by the Logging Interface, by means of the *accessLog* data structure; the 4-tuples *ac* corresponds instead to the *combinedLogs* structure. It offers also two functions, *addLog* and *checkLog*, to store and control access log, respectively. In particular, *addLog* takes in input a *ac_LI* log and stores it according to its request identifier *reqId* and type *type*. Instead, *checkLog* controls, for a given a request identifier *reqId*, whether there has been a violation during the request authorisation. If so, a security alert

Listing 1: Access Control Monitoring

```
contract Monitor{
  struct accessLog {
    string requestId;//unique identifier of the Log
    string encLog;   //encrypted access Log
    string logHash;  //Hash of the access log
    string logType;  //Type of the access log
    string timeStamp;//Timestamp of the log
    boolean fullLog;
  }
  struct combinedLogs {
    string requestId;
    accessLog pepReq;
    accessLog pdpReq;
    accessLog pdpDec;
    accessLog pepDec;
    boolean fullLog;
  }
  mapping (string => combinedLogs) accLogs;

  function addLog(...) public returns (string){
    accessLog memory tempLog;//create record
    ...//create/update record
    if (accLogs[reqId].fullLog){
        checkLog(reqId);
        //generate log <<EVENT>> for Analyser
    }
    return "Success in storing!";
  }
  function checkLog(string reqId) private returns (string){
    if (accLogs[reqId].pepReq.logHash != accLogs[reqId].
        pdpReq.logHash){
        //generate security alert <<EVENT>>
    }
    if (accLogs[reqId].pdpDec.logHash != accLogs[reqId].
        pepDec.logHash){
        //generate security alert <<EVENT>>
    }
    return "No alert!";
  }
}
```

is propagated via the event mechanism. Similarly, when a *ac* contains all the forming logs, a event is propagated to the Analyser to control the correctness of the calculated response.

Finally, the Analyser is implemented as a Java application interacting with the local Logging Interface via JSON-based invocations. The Analyser features the Java toolchain of FACPL [3], that in own turn exploits the Xtext parsing libraries [21] and the SMT solver Z3 [22].

### B. Test-Case

To simulate user interactions in the federation scenario, we created two different users, named *Alice* and *Bob*, exposing different sets of attributes. On the base of these attributes, we created an access control policy stored in the PRP. Such policy assigns different access rights to the users for the access to the available services. Namely, Alice is allowed to access only $S_1$, while Bob is allowed to access both $S_1$ and $S_2$. Instead, none of the users is allowed to access $S_3$.

Similarly, to simulate a suite of attacks, we assume that a communication, randomly chosen, for each user is violated, thus to represent threat **T1**. Similarly, the PDP is instrumented to randomly subvert the decision calculated by the PDP before returning it to the PEP, thus to represent threat **T2**. As a matter of fact, these threats are intended to allow Alice and Bob to both access the forbidden service $S_3$.

To dynamically modify the user load to which DRAMS is exposed we have used JMeter [23], a widely used open-source application to load functional behaviours and measure their performance on web-based applications. Indeed, JMeter has been configured to define, for each user, the following functional behaviour: first submitting an access request to $S_1$, then to $S_2$ and finally to $S_3$. It follows that each JMeter iteration, for each user, generates three access requests and three corresponding access responses to be stored in the smart contract. This functional behaviour has been then simulated by JMeter to an increasing number of active users (i.e., 5, 10, 25, 50, 100) within a period of 3 minutes, 5 minutes, 10 minutes, 20 minutes and 30 minutes, respectively.

### C. Feasibility Experiment

The test-case just identified has been applied to the considered federation scenario to inspect the following aspects

**a)** *Resiliency*: the ability of detecting an access control violations concerning both threats **T1** and **T2**.

**b)** *Cost*: the Ether needed for storing the gathered logs in a (public) Ethereum blockchain.

**c)** *Latency*: the time taken to store a log in a block in the blockchain.

The analysis results of these aspects assess the feasibility of deploying DRAMS upon Ethereum.

*a) Resiliency:* The resiliency test has evaluated if the architecture can detect any discrepancy under a considerable amount of usage load. As described in Section V-A, each interaction flow of a user simulates one attack scenario. Hence, the number of simulated attacks corresponds to the number of users simulated in a flow, e.g. with 5 simulated users we expect to have 5 attacks, for 50 users we expect to have 50 attacks and so on. To this aim, we have counted the number of generated alerts. Our evaluation suggests that the algorithm has been able to detect all simulated attacks under the described usage load.

*b) Cost:* Here, the associated cost corresponds to the total cost, in Ether, for storing access requests and other related information for each user. We compute the cost of executing an iteration while increasing the number of users. Before and after each iteration, we measure the current balance of the Ethereum accounts associated with tenants $T_1$ and $T_2$. The cost is then calculated by adding the subtracted value (the existing balance minus the current balance) for each account. As reported in Figure 4, the associated cost linearly increases with the number of users. The cost goes from around 0.38 ETH for 1 user to around 38 ETH for 100 users.

To compare the associated cost with the size, we have also calculated the size of the stored logs. In an average, the size of stored access logs for a user is around 25KB which increases linearly to around 2500KB for 100 users. Therefore, it costs around *0.0152* (0.38/25 = 0.0152 ETH/KB) ETH to store 1 KB of data in Ethereum at the time of this experiment had been carried out which is currently equivalent to 0.145 USD.

*c) Latency:* The latency has been evaluated by calculating the time it took to store a particular tuple in the blockchain. The tuple itself is stored via the smart-contract

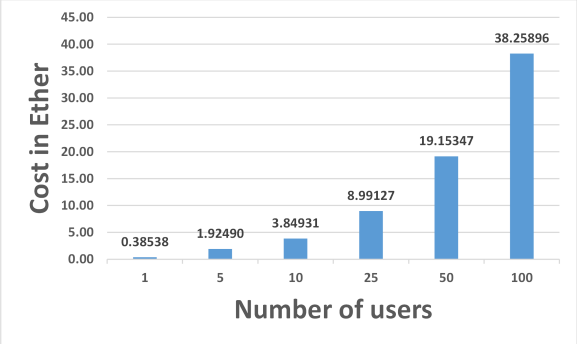


Figure 4: Number of users vs Cost

which is invoked by a transaction embedded inside a block. Since there is a latency involved in creating and validating a block, it is useful to examine the associated latency for storing tuples under a considerable amount of usage load. The average latency (in seconds) for each iteration group is presented in Figure 5. The important observation to make from Figure 5 is that the latency increases considerably (from 56.3s for 1 user to 6572.21s for 100 users) as more users are emulated to submit access requests. This is because access requests generate tuples which are then stored using the corresponding Ethereum client. The Ethereum client buffers the request in a transaction pool which are then released to the Peer-2-Peer network for broadcasting among other peers. The rate at which the tenant releases the buffered transaction is lower than the rate the tuples are generated when many users simultaneously submit access requests. Furthermore, the *gasLimit* value of 0x4c4b40 used to initialise the private blockchain allowed only one transaction to be included in each block even though there are numerous transactions in the pool. These two particular behaviours attribute to the increase in the latency.

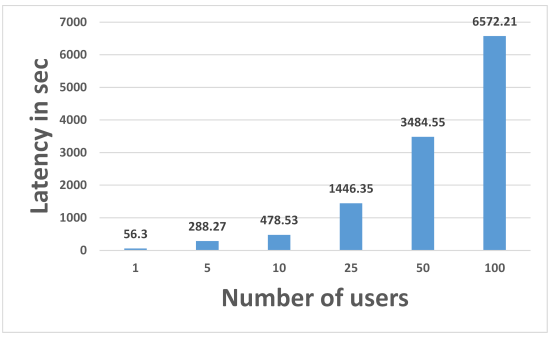


Figure 5: Number of users vs Latency

## VI. Discussion

In this section, we explore different aspects of DRAMS. We first analyse the security of DRAMS (Section VI-A) and then discuss some of the challenges in deploying the system on a public blockchain (Section VI-B).

### A. Security Considerations

Here, we comment on the security guarantees of DRAMS.

*Integrity.* DRAMS ensures the integrity of both the algorithm to detect discrepancies between access control decision taken and enforced and of the logs (i.e., threat **T3**). The integrity of the algorithm for checking discrepancies is ensured because the algorithm has been implemented as a smart contract that is deployed and executed on the blockchain and therefore is immutable. The integrity of the logs is also preserved for the same reason. An attacker would need to subvert the security of blockchain consensus protocol in order to be able to modify the smart contract and the associated logs.

*Availability.* DRMAS is fully decentralised exposing no single point of failure. If a single logging agent or the corresponding Logging Interface is under a Denial-of-Service attack (i.e., threat **T4**) or is compromised in a tenant allowing an attacker to alter an access request/response, the other components in other tenants remain functional. DRAMS, leveraging the blockchain, also provides a strong guarantee of availability of the stored logs as well as the smart-contract.

*Confidentiality.* Everything that is stored on the blockchain is public. However, logs may contain sensitive information about the access control policies enforced that could be used by an attacker to gain access on a federated resource. Therefore, DRAMS protects the confidentiality of the logs by storing them encrypted with AES 256.

*Other security concerns.* Finally, we report a few security considerations on the Analyser and its exploitation. The FACPL-based framework ensures compelling properties on the analysis on XACML policies, hence on detecting PDP violations. Differently from other analysis framework (e.g., those in [14], [15], [16]), it ensures, experimental proves on its semantic compliance with XACML; the proves have been defined by using the XACML implementation Balana and the validation tool X-CREATE [24]. On the other hand, it ensures that the FACPL translation procedure to SMT constraints is formally proved correct according to the FACPL semantics. Furthermore, notice that to ensure the correctness of the Analyser check, it is crucial to protect the integrity of the SMT representation of the policy in force. To this aim, we can exploit an hardware trusted platform to opportunely protect the integrity of the SMT constraints.

### B. Practicalities

Even though the Ethereum-based DRAMS offers compelling properties, there are a few practicalities to take into account for adopting such a novel system. We explore them in what follows.

*System Integrity.* The first practicality concerns the integrity of the monitoring system. In fact, even though the smart-contract of DRAMS is immutable, the integrity of the other components, e.g. the Logging Interface, cannot be guaranteed by-design, because they are deployed off-chain. Similarly, as all the Logging Interface instances share a symmetric key $K$, its management is of paramount importance. To mitigate both difficulties, we can introduce a trusted hardware platform (e.g., Trusted Platform Module) within the system. On the

one hand, it can be leveraged to store the symmetric keys by increasing the overall system security. On the other hand, this platform can be utilised to guarantee the integrity of the off-chain components.

*Log Size.* The key parameter highly affecting the monitoring system is the size of the log. In fact, the bigger the size is, the higher is the time (and the cost) to spend to store the log on the blockchain. To mitigate this practicality, different proposals are available. By relying on a private blockchain, where all PoW parameters can be dynamically tuned according to the needs, the latency can be maintained under control. However, due to the limited size of the network and a possibly lightweight PoW, this solution does not ensure strong integrity guarantees. Alternatively, a hybrid approach combining classical database with blockchain system should offer an adequate flexibility to find a tradeoff between latency, integrity guarantees and, in case of public chain, cost. A preliminary design to such a system is presented in [25].

*Private vs. Public.* Our early prototype of DRAMS is based on a private Ethereum blockchain where, as just mentioned, we can tune PoW parameters freely. Instead, to enjoy the strong integrity guarantees of public blockchain, we must deal with higher latency and, most of all, with the cost of the consumed Ether. Besides the cost, which is however an hopeless obstacle, the latency of public chains is currently around 14s per new block. Hence, DRAMS cannot be deployed upon a public chain, unless the number of interactions to be monitored are not frequent.

## VII. Related Work

This section discusses the related work in the areas of system monitoring tools (Section VII-A), decentralised access control (Section VII-B) and blockchain-based access control (Section VII-C).

### A. Monitoring Tools

There has been a long history of utilising monitoring tools for observing resource utilisation and tracking system performance, which predate well beyond the cloud paradigm. With the advent of distributed technologies such as cloud computing, new demand has emerged that requires monitoring tools to function in a seamless-distributed manner and to track new types of system properties. To satisfy these requirements, existing and new monitoring tools have been adopted and proposed for such distributed cloud environments.

A number of studies investigate the desirable properties that a cloud monitoring tool should possess [26], [27], [28]. The identified properties include scalability, adaptability, comprehensiveness, extensibility, resiliency, reliability, availability, accuracy, usability, affordability and achievablity. Then, different tools are analysed against these properties to identify which tool satisfies which properties. However, the analysis in these studies consider limited number of security properties (i.e., reliability and availability), and most of the tools lack in satisfying these security properties which undermine their usage to guarantee security assurance.

Most existing monitoring tools, such as Zabbix [29] and Nagios [30], maintain a client-server architecture where a client is a monitoring agent deployed to monitor specific resources in a specific layer and communicates with the server to transfer monitoring data. The server is responsible to collect and store monitoring data from the deployed agents, analyse such data, visualise them using a graphical user interface and raise an alert if it identifies a fault. Being based on the client-server architecture, these tools suffer from the most well-known weakness of the client-server architecture: the single point of failure. Indeed, these tools cease to function once the respective server is taken down by an attacker.

Another drawback of these tools is that they do not support any external access control framework such as XACML and thus cannot support discrepancy detection over logs created in different nodes in cloud environment. The solution proposed in this paper, instead, focuses on a decentralised monitoring architecture that supports XACML and allows heterogeneous nodes to have different access control policies.

### B. Decentralised Access Control

There have been several consensus maintaining approaches to dealing with consensus agreement among different (possibly colluding) entities in different tenants.

The Paxos-style protocol [31] focuses on reaching an agreement with respect to the state of an access request/response among nodes who may not maintain the same value, due to some error conditions. For instance, a node may have failed and hence lost the state value. The consensus is achieved using a concept of majority and is suitable for scenarios where fault-tolerance is required. However, in a security-critical system, a discrepancy regarding an access request/response among participating entities indicates a high possibility that some entities act maliciously, or collude with each other, or the respective access request/response reaching these entities has been altered, possibly by an attacker.

Sundareswaran et al. [32] present a decentralised approach for information accountability in the cloud environment utilising the programmable capability of Java Archive (JAR) files. Their approach allows the owner of a data to define access as well as usage control policy and attach the policy with the corresponding data within a JAR file in order to ensure accountability, securely log each data access incidence and enforce access/usage control policy in every single domain the data is stored within a cloud environment. This approach is quite interesting, particularly how the security of access logs have been ensured. However, their approach is unsuitable for any XACML-based external access control framework

### C. Blockchain-Based Access Control

There are interesting proposals that connect the blockchain technology with access control systems. Enigma [33], for instance, has proposed a decentralised blockchain-based platform for managing data storing and computation among parties. In that platform, the blockchain is utilised as the controller of access control and serves as a tamper-free event

logs. The difference between this approach with our solution is that we detail the security threats in the context of cloud federations and integrate important algorithms for detecting policy violations of access control.

In a similar vein, the survey on the principles and applications of blockchain technology [34] envisions the potential of exploiting blockchain as a promising way to manage access control but restricts the scenario to role based access control frameworks. Our solution is more general that it can be applied to other access control frameworks, and provides a concrete proposal consisting of a blockchain-based monitoring infrastructure and deployment strategies for cloud federations.

## VIII. Conclusion

In this paper, we presented DRAMS, a blockchain-based decentralised monitoring infrastructure for a distributed access control system. The main motivation of DRAMS is to deploy a decentralised architecture which can detect policy violations in a distributed access control system under the assumption of a well-defined threat model. Being decentralised, DRAMS exhibits no single point of failure. In addition, its blockchain-based architecture provides a strong guarantee of integrity and availability for the stored logs. We have deployed the architecture we propose for DRAMS in a simulated setting in which we use a private Ethereum blockchain. Our evaluation suggests that the proposed architecture is resilient against the threats of the specified threat model. Considering the associated cost for deploying such a system, we believe that our proposed system will be most suitably deployed using a private blockchain system.

This work can be extended in different directions. Firstly, we are interested in studying the impact of deploying DR AMS in a public Ethereum blockchain. In recent times, the Hyperledger project has gained considerable attention in the blockchain community [35]. Hyperledger is an open-source blockchain platform that can accommodate different sets of block-validating algorithms which might pave out the way for leveraging blockchain technologies without using any crypto-currency. This can address one of the biggest challenges for any large-scale adoption of DRAMS which is the associated cost. It will be interesting to investigate how Hyperledger can be exploited for this purpose.

The ultimate idea is to integrate DRAMS in a federated cloud setting such as SUNFISH, which is designed to be used by public sector structures consisting of thousands and thousands of users. The flexibility and strong security guarantees provided by DRAMS drive the desire for such an integration. However, the latency and, especially, the associated cost might be the key motivating factors for the successful deployment of DRAMS in such a cloud setting. The work presented in this paper aims to lay out the foundation for such a vision.

## Acknowledgment

This work has been supported by the EU H2020 Programme under the SUNFISH project, grant agreement N. 644666.

## References

[1] A. Celesti, F. Tusa, M. Villari, and A. Puliafito, "How to enhance cloud architectures to enable cross-federation," in *CLOUD*. IEEE, 2010, pp. 337–345.

[2] T. Kurze, M. Klems, D. Bermbach, A. Lenk, S. Tai, and M. Kunze, "Cloud Federation," in *Cloud Computing, GRIDs, and Virtualization*, 2011, pp. 32–38.

[3] F. P. Schiavo, V. Sassone, L. Nicoletti, and A. Margheri (Eds.), "Faas: Federation-as-a-service," *CoRR*, vol. abs/1612.03937, 2016.

[4] B. Suzic, A. Reiter, F. Reimair, D. Venturi, and B. Kubo, "Secure data sharing and processing in heterogeneous clouds," *Procedia Computer Science*, vol. 68, pp. 116–126, 2015.

[5] "SecUre iNFormatIon SHaring in federated heterogeneous private clouds (SUNFISH)," Accessed on 16 January, 2017, http://www.sunfishproject.eu/.

[6] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008, available at https://bitcoin.org/bitcoin.pdf.

[7] "Ethereum," Accessed on 16 January, 2017, https://www.ethereum.org/.

[8] Bill Parducci,Hal Lockhart, "eXtensible Access Control Markup Language (XACML) Version 3.0," 22 January, 2013, http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html.

[9] V. C. Hu, D. R. Kuhn, and D. F. Ferraiolo, "Attribute-based access control," *IEEE Computer*, vol. 48, no. 2, pp. 85–88, 2015.

[10] X. Jin, R. Krishnan, and R. S. Sandhu, "A unified attribute-based access control model covering dac, MAC and RBAC," in *DBSec*. Springer, 2012, pp. 41–55.

[11] D. F. Ferraiolo and D. R. Kuhn, "Role-based access control," in *NIST-NCSC National Computer Security Conference*, 1992, pp. 554–563.

[12] B. Suzic, B. Prünster, D. Ziegler, A. Marsalek, and A. Reiter, "Balancing Utility and Security: Securing Cloud Federations of Public Entities," in *C&TC*, ser. LNCS, vol. 10033. Springer, 2016, pp. 943–961.

[13] A. Paverd, A. Martin, and I. Brown, "Modelling and automatically analysing privacy properties for honest-but-curious adversaries," Tech. Rep., 2014.[Online]. Available: https://www. cs. ox. ac. uk/people/andrew. paverd/casper/casper-privacy-report. pdf, Tech. Rep.

[14] K. Arkoudas, R. Chadha, and C. J. Chiang, "Sophisticated access control via SMT and logical frameworks," *ACM Trans. Inf. Syst. Secur.*, vol. 16, no. 4, p. 17, 2014.

[15] F. Turkmen, J. den Hartog, S. Ranise, and N. Zannone, "Analysis of XACML policies with SMT," in *POST*, ser. LNCS, vol. 9036. Springer, 2015, pp. 115–134.

[16] A. Margheri, M. Masi, R. Pugliese, and F. Tiezzi, "A rigorous framework for specification, analysis and enforcement of access control policies," *CoRR*, vol. abs/1612.09339, 2016.

[17] L. M. de Moura and N. Bjørner, "Satisfiability modulo theories: introduction and applications," *Commun. ACM*, vol. 54, no. 9, pp. 69–77, 2011.

[18] WSO2, "Balana: Open source XACML implementation," 2015, https://github.com/wso2/balana.

[19] "The Go Programing Language," Accessed on 1 November, 2016, https://golang.org/.

[20] "Node.js," Accessed on 16 January, 2017, https://nodejs.org/en/.

[21] Xtext, *Language Development Made Easy*, http://www.eclipse.org/Xtext/.

[22] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *TACAS*, ser. LNCS, vol. 4963. Springer, 2008, pp. 337–340.

[23] "Apache JMeter," Accessed on 16 January, 2017, http://jmeter.apache.org/.

[24] A. Bertolino, S. Daoudagh, F. Lonetti, and E. Marchetti, "The X-CREATE Framework - A Comparison of XACML Policy Testing Strategies," in *WEBIST*. SciTePress, 2012, pp. 155–160.

[25] E. Gaetani, L. Aniello, R. Baldoni, F. Lombardi, A. Margheri, and V. Sassone, "Blockchain-based database to ensure data integrity in cloud computing environments," in *ITA-SEC*. CEUR-WS.org, To Appear.

[26] K. Fatema, V. C. Emeakaroha, P. D. Healy, J. P. Morrison, and T. Lynn, "A survey of cloud monitoring tools: Taxonomy, capabilities and objectives," *Journal of Parallel and Distributed Computing*, vol. 74, no. 10, pp. 2918–2933, 2014.

[27] G. Aceto, A. Botta, W. De Donato, and A. Pescapè, "Cloud monitoring: A survey," *Computer Networks*, vol. 57, no. 9, pp. 2093–2115, 2013.

[28] J. S. Ward and A. Barker, "Observing the clouds: a survey and taxonomy of cloud monitoring," *Journal of Cloud Computing*, vol. 3, no. 1, p. 1, 2014.

[29] R. Olups, "Zabbix 1.8 network monitoring," 2010.
[30] W. Barth, "System and network monitoring," 2008.
[31] L. Lamport *et al.*, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.
[32] S. Sundareswaran, A. Squicciarini, and D. Lin, "Ensuring distributed accountability for data sharing in the cloud," *IEEE Transactions on Dependable and Secure Computing*, vol. 9, no. 4, pp. 556–568, 2012.
[33] G. Zyskind, O. Nathan, and A. Pentland, "Enigma: Decentralized computation platform with guaranteed privacy," *CoRR*, 2015.
[34] M. Pilkington, "Blockchain technology: Principles and applications," HAL, Tech. Rep., 2016.
[35] "Hyperledger," Accessed on 1 November, 2016, https://www.hyperledger.org/.