Enforcing Security and Assurance Properties in Cloud Environment

Aline Bousquet, Jérémy Briffaut, Eddy Caron, Eva María Dominguez, Javier Franco, Arnaud Lefray, Oscar López, Saioa Ros, Jonathan Rouzaud-Cornabas, Christian Toinard, et al.

▶ To cite this version:

Aline Bousquet, Jérémy Briffaut, Eddy Caron, Eva María Dominguez, Javier Franco, et al.. Enforcing Security and Assurance Properties in Cloud Environment. 8th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2015), University of Cyprus, Dec 2015, Limassol, Cyprus. hal-01240557

HAL Id: hal-01240557
https://hal.inria.fr/hal-01240557

Submitted on 9 Dec 2015

Enforcing Security and Assurance Properties in Cloud Environment

Aline Bousquet*, Jérémy Briffaut*, Eddy Caron†, Eva María Dominguez¶, Javier Franco‡, Arnaud Lefray*†,
Oscar López§, Saioa Ros§, Jonathan Rouzaud-Cornabas‖, Christian Toinard* and Mikel Uriarte§
*INSA Centre Val de Loire, Univ. Orléans, LIFO EA 4022, Bourges, France
†University of Lyon - LIP, CNRS - ENS Lyon - Inria - UCB Lyon, France
‡Industry and Advanced Manufacturing department, Vicomtech-IK4, Spain
§Research and Development department, Nextel S.A., Spain
¶Transport - Technology and Development, IKUSI, Spain
‖Universiy of Lyon, CNRS, Inria, INSA-Lyon, LIRIS, UMR5205, F-69621, France
Email: {aline.bousquet, jeremy.briffaut, christian.toinard}@insa-cvl.fr, {eddy.caron, arnaud.lefray}@ens-lyon.fr,
jonathan.rouzaud-cornabas@inria.fr, jfranco@vicomtech.org, {sros, olopez, muriarte}@nextel.es, eva.dominguez@ikusi.com

Abstract—Before deploying their infrastructure (resources, data, communications, ...) on a Cloud computing platform, companies want to be sure that it will be properly secured. At deployment time, the company provides a security policy describing its security requirements through a set of properties. Once its infrastructure deployed, the company want to be assured that this policy is applied and enforced. But describing and enforcing security properties and getting strong evidences of it is a complex task.

To address this issue, in [1], we have proposed a language that can be used to express both security and assurance properties on distributed resources. Then, we have shown how these global properties can be cut into a set of properties to be enforced locally. In this paper, we show how these local properties can be used to automatically configure security mechanisms. Our language is context-based which allows it to be easily adapted to any resource naming systems *e.g.,* Linux and Android (with SELinux) or PostgreSQL. Moreover, by abstracting low-level functionalities (*e.g.,* deny write to a file) through capabilities, our language remains independent from the security mechanisms. These capabilities can then be combined into security and assurance properties in order to provide high-level functionalities, such as confidentiality or integrity. Furthermore, we propose a global architecture that receives these properties and automatically configures the security and assurance mechanisms accordingly. Finally, we express the security and assurance policies of an industrial environment for a commercialized product and show how its security is enforced.

Keywords—Security, Cloud, Assurance, Enforcement, Use-case

I. INTRODUCTION

In security, three main concepts commonly known as the CIA-triad (not to be confused with the US agency) has been widely used for decades: Confidentiality, Integrity and Availability. Both the Departement Of Defense guidelines (TCSEC/Orange Book) [2] edited in 1985 and the more recent Common Criteria (ISO/IEC 15408) international standard define security as an integration of availability, confidentiality and integrity.

In a survey [3] on Cloud adoption practices, the Cloud Security Alliance (CSA) indicates that 73% of the participating industries are concerned about the security of their data. Thus, while many companies are transitioning to Cloud computing, they are also worried about the security risk. But Cloud platforms lack of reliable security [4]. Furthermore, Halpern et al. [5] state that security policies described in a natural language have quite ambiguous semantics. To answer these problems, we need to provide a way (a language) to let the Cloud tenants (*e.g.,* companies) express their security requirements *i.e.,* through a security policy. This security policy must them be enforced on the Cloud platform and assurance reports (*i.e.,* proofs) of this enforcement must be given to the tenants.

A single security mechanism cannot protect a heterogeneous and multi-layer system such as a Cloud [6]. Consequently, it is a set of uncoupled (and already existing) mechanisms that will be used to enforce the security. However, even if the mechanisms are uncoupled, it is mandatory to carefully take into account their capabilities (*i.e.,* what they are able to enforce) and configure all of them at once to provide the wanted security. But, each of these mechanisms also comes with its own configuration language.

In [1], we have defined a specification language for global security properties (*i.e.,* properties that involve distributed resources). We have shown how these global properties can be automatically cut into a set of local properties. These local properties can be used to automatically configure security mechanisms. Moreover, our common independent language abstracting low-level capabilities can be used to provide proofs of security enforcement (*i.e.,* assurance).

As said previously, the tenants also require to receive a proof that their security is indeed enforced during the whole life cycle of the infrastructure. Accordingly, using our language, the tenants can expressed their security assurance requirements.

Once the security and assurance policies has described a set of properties to enforce, an architecture is required to automatically configure distributed and heterogeneous mechanisms. Furthermore, this architecture must also send back assurance reports to the tenants.

To show the usability and capacities of our solution, we

describe how our language has been used to define the security policy of a complete industrial application. Then, we show how our architecture is used to automatically enforce the policy and generate assurance reports of it.

This paper is organized as follows. In Section II, we present a set of existing security mechanisms that could be used to provide security in Clouds, and the related work around Cloud security and assurance. Section III describes the language and the architecture we use for the security policy enforcement and assurance. Section IV details an industrial use-case and the whole process to secure it, and Section V concludes this paper.

II. RELATED WORK

The solution proposed in this paper aims to both enforce and assure security properties, such as confidentiality or integrity. Hence, this section first describes the works related to the definition of security properties and their enforcement. Then, we quickly present the security mechanisms we will use in Section IV. Finally, we present some existing solutions for assurance.

A. Security Policy and Enforcement

Because of the ever increasing adoption of Cloud Computing platforms, many researches have been done to improve their security. As stated in [6], a security policy language is required to allow the tenants to express their security requirements. Indeed, this makes sense that the tenants define their security as they are the ones that know the best their infrastructure and its security requirements.

Some of works related to security policy languages are specific to a programming language and require the modification of the application sources. Ponder2 [7] is a distributed object management system. The Ponder2 language can express security and management policies for distributed systems. It is declarative and object-oriented and can be used to declare different types of policies. Consequently, it can only be used on Java application augmented with the Ponder2 solution. Same for the A4Cloud project described in [8] and its associated language, A-PPL. Furthermore, this solution focuses on privacy and accountability, but does not address other classes of security, such as isolation.

Works such as [9], [10] also strength the need of combing multiple security mechanisms to provide an end-to-end and cross-layer security. VESPA [10] is one of such architecture for protecting cloud infrastructures using a policy-based management approach. However, this work is oriented toward the use of automatic computing to create self-protection loops. Consequently, they lack a language allowing the tenants to express their security. Nonetheless, a combination with our works could be a a way for future work. In [11], authors present MEERKATS, a mission-oriented Cloud architecture dedicated to security. It is composed of several components that aim to address several types of attacks and seek to provide high flexibility in the use of the protection mechanisms. Nevertheless, MEERKATS lacks a simple way of expressing the security requirements of an infrastructure. In [12], the authors present a policy-based security framework. Their ASPF policy consists of an attribute map (that links system elements to their attributes) and a set of rules (indicating which actions are allowed). While the ASPF framework can enforce security policy, only low-level security properties can be expressed, which makes the definition of the security policy complex.

Finally, several works such as [13] have been done around the use of XACML to define security policies. But they focus on a specific type of mechanisms, access control. Moreover, XACML is a complex language [14] that requires the verification of the policy conformance regarding its syntax and its semantic. Furthermore, XACML does not express high level security requirements such as integrity but rather it expresses the policy directly using low-level capabilities. Accordingly, the size of the security policy is larger and thus the risk of making mistakes increases as these policies are written by human versus generated ones. Nevertheless, it could be possible to automatically generate such policy from our security policy language. Thus, such work could be used as a security mechanisms by our solution. [15] presents a privacy-aware access control system since privacy is an important concern for most users. However, the PRIME architecture is based on XACML and therefore presents the same limitations.

Each of these solutions either focuses on one kind of protection (mainly access control) or uses low-level security policy (tedious to define). To the best of our knowledge, there is no current research on the configuration of existing security mechanisms through a common abstract language.

B. Security Mechanisms

Many different security mechanisms exist, providing a wide range of features. We present some of them that we use in our solution. SELinux [16] is a Linux Security Module (LSM) providing MAC (Mandatory Access Control). PAM [17] provides authentication management support for Linux. Iptables [18] is a standard Linux firewall. We use the tunnel functionality provided by OpenSSH [19] to secure the communications between the machines.

Even if not used in this paper, cryptographic solution (such the one presented in [20] for the security of medical records or even homomorphic encryption [21]) could be added to the list of the security mechanisms that our solution takes into account. Indeed, the solution we propose is mechanism-agnostic.

C. Assurance

Operational Security Assurance [22] provides the ground for confidence that deployed security mechanisms are running as expected. Some researches have been done to evaluate security assurance. For instance, Common Criteria [23] evaluates security functionality and assurance by means of tests conducted by users. However, this process is static and time consuming. Consequently, it cannot be directly applied for a continuous evaluation of security assurance. Furthermore, Common Criteria focuses on the implementation phase of the product rather than on the operation phase, when the product is used.

Assurance Profile [24] is a formalized document that defines a common set of security assurance measurement requirements for a service infrastructure and facilitates a future evaluation against these needs. This is the approach selected for the assurance framework development.

XCCDF (eXtensible Configuration Checklist Description Format) is a standard that can perform assurance checks. It belongs to SCAP [25], a set of specifications from NIST to standardize the format and the naming of information reporting concerning specific security configurations. XCCDF provides security checklists and benchmarks to support an automated compliance testing over a set of target systems. OpenSCAP [26] is an auditing tool implementing SCAP and XCCDF.

III. ARCHITECTURE AND LANGUAGE

As we have seen, many security mechanisms are efficient but are focused toward a specific issue and/or type of protection. It is important to understand that we do not propose any new or more secure mechanism, but we rather consider the existing ones and automatically configure/coordinate them in order to enforce high-level security properties. In this section, we first present our functional architecture. Then, we present our language and how it is used to enforce a security policy. Eventually, we show how it is possible to automatically assess the correctness of the enforcement.

A. Functional Architecture

As depicted in Fig. 1, our solution consists in a 3-steps cycle. First, the tenant specifies its security policy using the language detailed in Section III-B. Then, from this high-level policy specification, the policy is enforced by firstly selecting security mechanisms and secondly configuring them. At the end, the policy specification and the list of selected mechanisms are used to generate the assurance profile. The assurance part will verify whether or not the security properties (from the policy) are duly enforced. These assurance checks are sent as feedbacks to the specification step to notify the tenant if the enforcement is correct/incorrect but also if the available mechanisms are sufficient (or not) to enforce the policy.
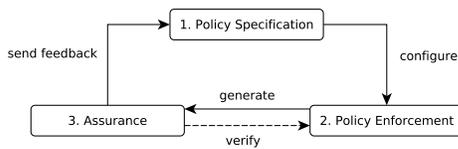

Fig. 1: Functional Architecture

B. Policy Specification

During the policy specification step, a knowledgeable tenant (i.e., a security expert) expresses a set of security properties to enforce e.g., confidentiality, integrity.

In [1], we have defined the Cloud Security Property Language (CSPL) that allows to specify security properties. In particular, we have shown how to automatically transform a property on a set of distributed objects (that we refer as a global property) into a set of properties on local objects (referred as local properties). In this paper, we are focusing on the enforcement of local properties with a given security mechanism and verifying the correct enforcement of this property. Therefore, in the following we consider only local properties.

CSPL is a context-based language. A **context** is a set of attributes where each attribute characterizes an entity or a set of entities. At the highest level, entities can be classified into 2 categories: subject (i.e., the active resources such as users and processes) and object (i.e., the passive resources such as files). For instance, the context `configApp = (File="Configuration")` `:(Domain="App")` identifies the configuration files (attribute `File`) of an application (attribute `Domain`). Therefore, it is possible to use the same set of contexts on different systems. A specific mapping file is required for each system to associate the resources (e.g., files, users, sockets, processes) name (e.g., the full path of a file, the user id, IP addresses, processes name) and the context which they belong to. For example, to associate the application's configuration to the corresponding files, the following line is added to the mapping file ("o" for "object"):

```
o /opt/dbhook/dbhook.conf configApp
```

Using contexts to identify resources (or set of resources), CSPL allows to define **security properties** and by relying on contexts to address entities, the expression of security properties is independent from the resources naming of the target system. These properties are independent from any security mechanism; in fact, multiple mechanisms can realize the same security property. Our proposition is to select a security mechanism able to enforce a given property from a pool of available mechanisms.

For instance, the property `P1` expresses that the context `SCInt` integrity has to be guaranteed with the exception to `SCAuth` contexts that are allowed to go against this property i.e., no one is allowed to modify it except the resources with the context `SCAuth`.

```
P1:= Integrity (Context SCInt, Context SCAuth);
```

Then, `P1` must be instantiated. For example, to specify that the integrity of the application's configuration files with the context `configApp` should be protected and only be the user with the context `adminRoot = (Username="appAdmin"):(Role="StandardUser|appAdmin")` is allowed to go against, the following property is instantiated: `Integrity(configApp, adminRoot)`.

From the tenant's point of view, the security properties are abstract i.e., the tenant only considers the semantics of the properties, and not the underlying security mechanisms. However, the properties need to be precisely defined in order to be enforced. Thus, we introduce the concept of **capabilities**. A capability is an elementary function provided by one or several security mechanisms. For instance, `C1:= deny_all_write_accesses(Context)` is a capability that can be provided by access control mechanisms (e.g., Unix's DAC permissions or SELinux) but also by other security mechanisms. It can be used to enforce an integrity property. Consequently, the integrity property `P1` can be defined as follows:

```
P1:=Integrity (Context SCInt, Context SCauth) {
  deny_all_write_accesses(SCInt);
  allow_write_access(SCInt, SCauth);
}
```

The context `SCInt` represents an (set of) object(s) to secure, while the context `SCauth` is the identity of a (set of) subject(s)

that can write to *i.e.,* modify, `SCInt`. Two capabilities are involved in this property. The capability `deny_all_write_accesses` denies all write accesses to the context while the capability `allow_write_access` allows the context `SCauth` to counter the previous capability.

In a Cloud environment, a security property can address multiple machines (*e.g.,* Virtual Machines). In our language, a context mapped to an IP address refers to a machine. For example, a mapping file can include the following lines ("`c`" for "computer"):

```
c 192.168.30.8   hostClient
c 172.22.11.181 hostReverseProxy
```

The user can use particular attributes to include network-specific metadata such as port *e.g.,* `tunServer:=hostReverseProxy :(Port="5900")`. These kinds of contexts can be used in typed security properties *i.e.,* properties accepting only specific types of contexts (here IP:Port). For example, `Confidentiality_Tunnel (hostClient, tunServer)` creates a tunnel between two machines.

In this paper and especially in our use-case (see Section IV), we will use the following security properties. Each of the parameter (*i.e.,* contexts) of the properties can be a single context or a set of contexts.

- **Isolation**(Context SC1): Isolate a context `SC1` from other resources and vice-versa (*e.g.,* it isolates an application and its resources from the rest of the system)

- **Confidentiality**(Context SC1, Context SC2): Deny every contexts from reading a context `SC1` with the exception of the context `SC2`

- **Integrity**(Context SC1, Context SC2): Deny every contexts from modifying `SC1` with the exception of the context `SC2`

- **Confidentiality_Tunnel**(IP:Port SC1, IP:Port SC2): Create a secure tunnel between 2 contexts `SC1` and `SC2` of types IP:Port.

- **Access**(Context SC1, Context SC2): Allow connections from `SC2` to `SC1`

- **Authentication**(Context SC1, Context SC2, Context SC3): Upon successful connection from `SC1` on `SC2`, modify the context of `SC1` to `SC3`

Finally, we enrich the set of security properties with the following assurance property :

- **Assurance**(int secs): Run assurance checks at frequency `secs` (*i.e.,* every `secs` seconds) for every security property.

C. Policy enforcement

The second step of our solution, the policy enforcement, must first automatically select a set of suitable security mechanisms enabling the enforcement of each security property. And then, it must automatically configure them accordingly. The component in charge of the policy enforcement step is call the Secure Element Extended (SE^E). In the mean time, the assurance property triggers the generation of an assurance profile based on the security properties and the selected mechanisms.

Let's first present the selection and enforcement of the security properties. Figure 2 presents the architecture of the SE^E and how it can enforce the security policy. The SE^E takes as input the security properties and the contexts/resources mapping. The SE^E also considers the security mechanisms that are available on the system and their capabilities. Then, it selects the right security mechanisms and enforces the properties by configuration.
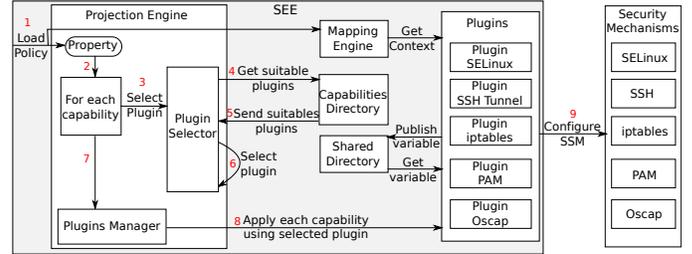

Fig. 2: Architecture of the SE^E.

First, the SE^E loads the security policy *i.e.,* the contexts, the properties, and the contexts/resources mapping (step 1 on Figure 2). Then, the SE^E iterates over security properties of the policy. For each capability of each property (step 2), the SE^E selects a plug-in (*i.e.,* a security mechanism) that can apply it (steps 3 to 6). This is done by querying the Capabilities Directory that contains the association between the capabilities and the security mechanisms (steps 4 and 5). Once the Plugin Selector has the list of matching mechanisms, it selects one of them (best-effort algorithm, step 6). When a capability/plug-in mapping has been found for the property, it is sent to the Plugins Manager that controls the plug-ins (step 7). Then, the Plugins Manager contacts each plug-in that needs to perform some actions (step 8) and the plug-ins configure their associated security mechanism (step 9).

The use of plug-ins offers a modular model: new mechanisms can be easily added by developing the associated plug-in. Plug-ins implement a simple interface to communicate with the SE^E, but the way they interact with their mechanisms is up to the plug-in's developer.

For instance, if the SE^E receives the integrity property `Integrity`(configApp, adminRoot) defined in the previous section, it can enforce it using several security mechanisms. If this property is enforced using SELinux, then the SE^E generates a SELinux module that forbids any write operation to the files labeled `configApp` that does not come from a user labeled `adminRoot`.

The SE^E also provides secure communication capabilities, especially for the case of properties between multiple systems. Thus, the two sides (*i.e.,* the selected mechanisms applying to the two contexts) of the communication must use compatible mechanisms to enforce a property. For instance, let us consider the property `Confidentiality_Tunnel`(hostClient, tunServer). The server allows the connection of the user through the defined port, and the client sets up the tunnel. The coordination is done by the SE^E's communication capabilities.

Now, let's present the generation of assurance files for the

assurance framework presented in Section III-D. To validate the enforcement of the security policy, multiple files are generated and given as input to the Assurance step, namely the XC-CDF and system-specific scripts. System-specific scripts are generated using a process similar to the property enforcement: each property definition includes an assurance specification, using capabilities. For instance, the assurance of the `Integrity` property is defined as follows:

```
P1:=Integrity (Context SCInt, Context SCauth) {
  assurance {
    boolean c = true;
    for (SCUserTmp IN get_all_users()) {
      if (SCUserTmp.Id == SCauth.Id) {
        c &= check_write (SCInt, SCauth);
      } else {
        c &= (NOT check_write (SCInt, SCauth)); }
    } return c; }}
```

As a result, the system-specific script will contain the implementation of the `check_write` assurance capability for the context `SCInt` and the authorized context `SCAuth`. This generated script is called a **Based Measure** (BM) as it is the lowest level of assurance measure. Therefore, the XCCDF is simply a list of Based Measures. The XCCF file and the related scripts are given as input to the assurance step.

D. Assurance

In order to be able to evaluate continuously the security assurance for a service, it is necessary to implement a process composed of several steps: modeling, measuring, aggregation, evaluation and presentation of the security assurance reports. This process is supported by a set of software components that compose our assurance framework. Fig. 3 presents our assurance framework architecture.
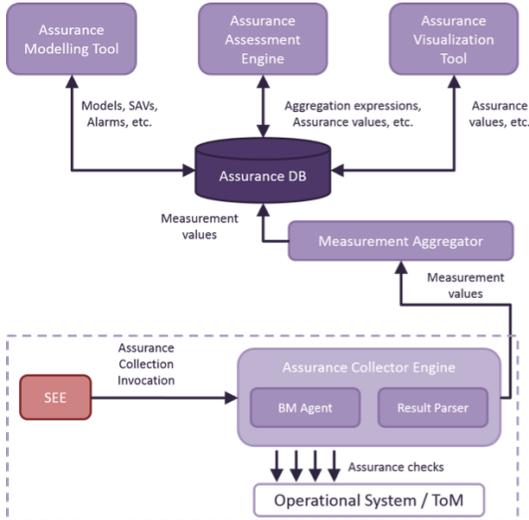

Fig. 3: Assurance Framework Architecture.

1) System Measurement Collection: As stated before, the assurance step receives an **XCCDF** file and the related system-specific scripts called **Based Measures**. The SE^E is responsible for launching the measurement collection process realized by the **Assurance Collector Engine (ACE)**. This engine includes a **BM Agent** which executes several system-specific

scripts. Script results are associated with some metadata including extra information to unequivocally identify their origins and contexts. Note that the ACE, based on OpenSCAP, is the only assurance module deployed in each virtual machine.

Next, the **Measurement Aggregator (MA)** receives these measurements from each node, validates and classifies them according to their metadata, before storing them in the **Assurance DB**.

2) Assurance Results Presentation: We have seen how to execute low-level assurance checks and collect their results. In the following, we present how to add semantics to the collected results *i.e.,* interpret them, and how to present all assurance checks to the tenant in a modular and concise manner. Our assurance model defining the entities/files relations from low-level measures to high-level views is presented in Fig. 4.
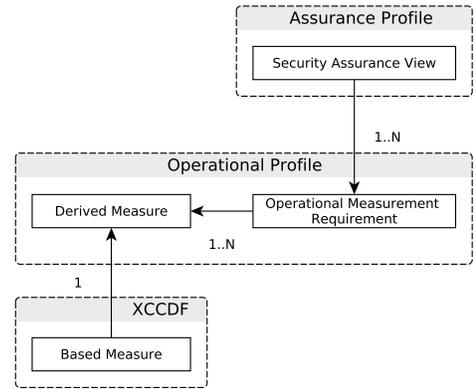

Fig. 4: Assurance Model.

First, we have not determined yet if the collected values mean a correct or faulty enforcement *i.e.,* we need to interpret them. Hence, we call **Derived Measure (DM)** the interpretation of a Based Measure.

Depending on the number of security properties and the size of the system (*i.e.,* number of objects), it is possible to have a significantly large set of assurance checks (or Derived Measures) which can be an impediment to the tenant's verification task. Our solution is to hierarchically aggregate these measurements.

Therefore, a set of Derived Measures is aggregated in an **Operational Measurement Requirement (OMR)** via an aggregation function. In particular, if all Derived Measures have successfully passed their checks, then the OMR is marked as successful. In other words, an OMR is a set of system assurance checks. The **Operational Profile (OP)** contains both the definition of OMRs (*i.e.,* the list of Derived Measures) and the definition of Derived Measures.

Our next level of abstraction is to allow the tenant to specify several **Security Assurance Views (SAVs)** where an assurance view is an aggregation of Operational Measurement Requirements. The definition of Security Assurance Views is done in the **Assurance Profile (AP)** file.

In Fig. 3, the **Assurance Modeling Tool** takes in input the Assurance Profile and the Operational Profile to maintain a Security Assurance Model. Depending on the layer of the

assurance model (*e.g.,* SAV, OMR, or DM), the **Assurance Assessment Engine** is responsible for deciding if the collected assurance values meet the expectations and for computing the aggregation results.

Finally, the **Assurance Visualization Tool** provides a Graphical User Interface for the user to monitor the assurance status. For a monitored service, the user will be able to select the different SAVs available in the Security Assurance Model. This tool presents an assurance report adapted to the tenant's concerns.

To sum up, the modeling and configuration of the assurance framework rely on the definition and refinement of 3 XML files : 1) the AP for defining the high level measurement requirements and the assurance views adapted to the tenant's concerns, 2) the OP for establishing the links with the real environment, and 3) the XCCDF for specifying how to execute the assurance measurements. Examples of these files are provided in Section IV.

IV. IKUSI'S USECASE

A. *Description of Usecase*

In this section, we present an industrial use-case based on Ikusi application : Airport Management. It aims to provide a centralized-operational management for airports management services. It involves the coordination of a group of processes, where both human and IT system interactions are required. It is a classical 3-tier web architecture *i.e.,* a HTTP frontend (tier-1), an application server (tier-2) and a database (tier-3). This architecture is deployed on top of an IaaS Cloud and is provided to end-users through a SaaS model. Moreover, one instance of the application server is launched for each client *i.e.,* for each airport.

Services provided by the architecture include the management of an operational data repository for each airport operator and passenger, the real time management of flight status updates, and the dynamic allocation and optimization of assigned resources according to data from air flight companies and airport operators.

It is based on message exchange modules, on resource allocation and on billing management airport services to provide airlines with an operational platform based on Cloud computing technology. It also incorporates enhanced security solutions based on a network of secure element developed in the SEED4C project.

The use-case is presented in Fig. 5. Four different kinds of machines or VMs are involved. First, the machine `ctseed1` is the client machine. It is the device that is used within the airport to access the airport's services. Secondly, the `reverseproxy` VM (*i.e.,* tier-1) is a proxy used by the end-user to access the airport's services. The `musik` VMs (both `musik1` and `musik2`) belong to an airport (`MAD`[1] or `EAS`[2]) and are accessed by the end-user machine through the proxy (*i.e.,* an instance of the application server, tier-2). The corresponding VM is selected based on the location of the end-user. Apart from their airport domain, these VMs are identical, so

---

[1]MAD: Madrid Airport code
[2]EAS: San Sebastian Airport code

we only consider one of them in this use-case (the security policy would be duplicated). Each of these VMs runs a `Musik` application that accesses the database (running in the `seed4c_mysql` machine) *i.e.,* tier-3.

B. *Security policy*

Based on the use case description, a security policy is defined through the graphical tool Sam4C (see Fig 5).

The next listing presents an excerpt from the security policies for the different VMs of the use-case:

```
1  // Policy for the Database VM
2  Isolation(DomainAODB);
3
4  Integrity(BinaryAODB);
5  Integrity(ConfigAODB, AdminRoot);
6  Integrity(KeyAODB, AdminRoot);
7  Integrity(LogAODB, ServiceAODB);
8
9  Confidentiality(FileAODB, ServiceAODB);
10 Confidentiality(KeyAODB, AdminRoot);
11 Confidentiality(ConfigAODB, AdminRoot);
12 Confidentiality(ConfigAODB, ServiceDB);
13 Confidentiality(LogAODB, AdminRoot);
14 Confidentiality(LogAODB,AdminOperator);
15
16 Authentication(HostReverseProxy, ServiceSSH, "SystemUser|
      CloudProvider|AdminRoot|AdminOperator|User");
17
18 Access (MysqlPort|MysqlProxyPort|SSHPort|NTPPort, AnyIP);
19
20 Assurance(Freq);
21
22 // Policy for the ReverseProxy VM
23 Integrity(BinaryModuleWeb);
24 Integrity(BinaryWeb);
25 Integrity(ConfigWeb);
26
27 Confidentiality(ConfigWeb,AdminRoot);
28 Confidentiality_Tunnel(tunClient, tunServer);
29
30 Access (SSHPort|NTPPort, AnyIP);
31
32 Authentication(anyone, ServiceSSH, "SystemUser|CloudProvider
      |AdminRoot|AdminOperator|User" );
```

The first security property (line 2) of this listing sandboxes the whole application. Lines 4 to 7 forbid anyone to edit the application's binary, but allow several write accesses to its files (configuration, keys, and logs). Lines 9 to 14 forbid read access to the application resources except for the application itself. Line 16 specifies the context evolution upon an SSH connection: a role is given to the authenticating user depending on his login data. Line 18 opens several ports for all incoming IP addresses. Line 20 defines the assurance tests to perform.

The second part of the listing describes the policy for the reverse proxy. Lines 23 to 25 guarantee the integrity of the Web application. Line 27 requests the confidentiality of the configuration files. Line 28 specifies that the network communication between the proxy and the client should be kept confidential. Line 30 opens some ports. Finally, line 32 manages the contexts evolution upon SSH connections.

The contexts used in this policy are associated to system resources. An extract from the association file is displayed in the next listing:

```
1  o /opt/dbhook(/.*)?         FileAODB
2  o /opt/dbhook/dbhook.conf   ConfigAODB
3  o /opt/dbhook/keys(/.*)?    KeyAODB
4  o /opt/dbhook/log(/.*)?     LogAODB
5  o /opt/dbhook/proxydaemon.sh BinaryAODB
```
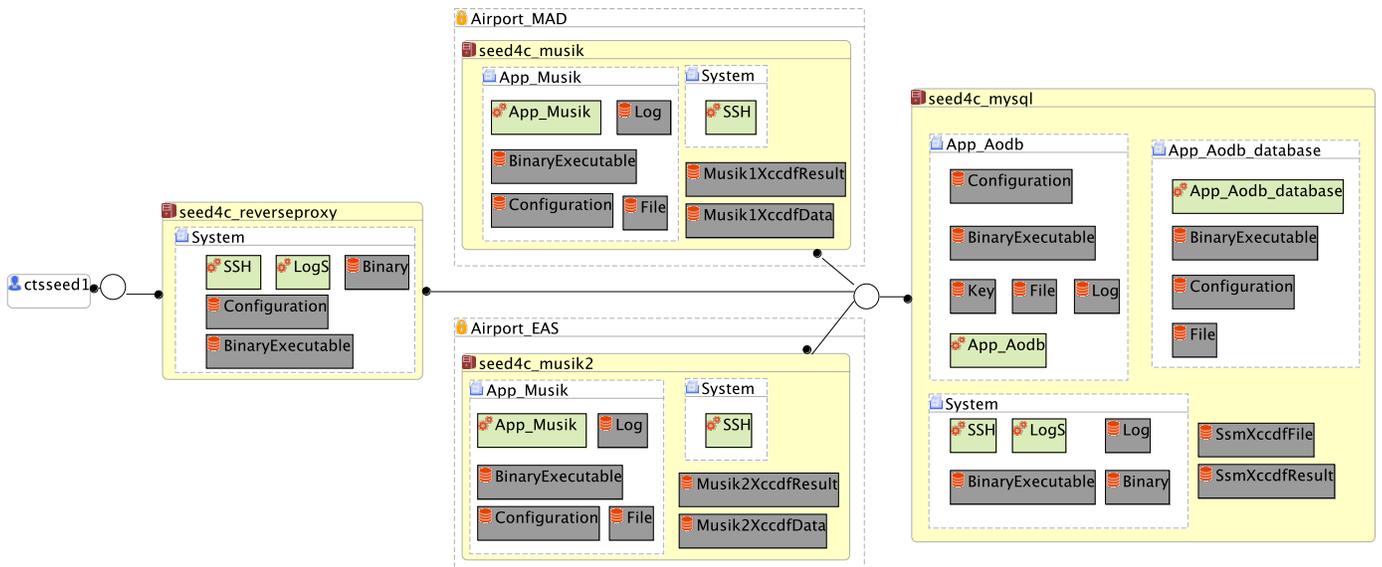
Fig. 5: Usecase Description

```
6  o /etc/rc\.d/init\.d/dbhook  BinaryAODB
7  o /opt/oscap/ssm/results/SSM-results-$date.xml SSMResultFile
8  o /opt/oscap/ssm/SSM-xccdf.xml SSMXccdfFile
9
10 p /usr/bin/mysqld_safe     ServiceDB
11 p /usr/libexec/mysqld      ServiceDB
12 p /usr/bin/mysql-proxy     ServiceAODB
13 p /usr/sbin/sshd           ServiceSSH
14
15 u cloudprovider      CloudProvider
16 u tenant-admin       AdminRoot
17 u tenant-operator    AdminOperator
18 u user               User
19
20 c 172.22.11.181    HostReverseProxy
21 c 172.22.11.178    HostServerBBDD
22 c 212.81.220.68    HostClient
```

Lines 1 to 8 of the mapping file associate the contexts to files. Lines 10 to 13 are for the processes, lines 15 to 18 for the users, and lines 20 to 22 for the computers (IP addresses).

C. Security Enforcement

The security policy is enforced by several security mechanisms. The SE^E detects what are the available mechanisms and selects those that can enforce the properties.

In this usecase, four mechanisms collaborate to enforce the whole policy.

1) SELinux: First security mechanism available is SELinux. It enforces properties from three groups: isolation, confidentiality, and integrity. To enforce them, the plug-in generates a SELinux module.

Upon receiving an isolation property for a domain, the plug-in creates a SELinux module to isolate all elements of this domain from the rest of the system. Then, the plug-in will allow some interactions corresponding to confidentiality and integrity properties. To enforce the properties **Isolation**(DomainAODB), **Integrity**(ConfigAODB, AdminRoot), and **Confidentiality**(ConfigAODB, "ServiceDB|AdminRoot") from policy, the following module is generated (see next listing). Lines 2-5 define the domain

and SELinux contexts, while lines 7-8 give authorization rules. Lines 12-13 associate SELinux contexts to resources.

```
1  $ cat Aodb.te
2  policy_module(Aodb,1.0.0)
3  see_create_service_domain(Aodb)
4  see_create_files_type(Aodb_conf_t)
5  see_create_files_type(Aodb_file_t)
6
7  see_files_type_read_write(Aodb_t,Aodb_conf_t)
8  see_files_type_read(idAodbAdmin_t,Aodb_conf_t)
9  [...]
10
11 $ cat Aodb.fc
12 /opt/dbhook/dbhook.conf   gen_context(system_u:object_r:
     Aodb_conf_t,s0)
13 /usr/bin/mysql-proxy     gen_context(system_u:object_r:
     Aodb_exec_t,s0)
14 [...]
```

2) PAM: The PAM plug-in enforces authentication properties. Indeed, such property specifies how contexts can evolve to have correct properties applied. Moreover, it controls the authentication rights and allows or denies a user authentication. Upon encountering the property **Authentication**(anyone, ServiceSSH, "SystemUser| CloudProvider|AdminRoot|AdminOperator|User"), PAM plug-in adds a rule to PAM configuration in order to detect a successful login:

```
session  required  pam_exec.so  /etc/see/scripts/notifyLogin
```

When a successful authentication occurs, PAM executes the script notifyLogin (see next listing), which informs the SE^E (through Ncat) of a connection and sends data, such as the user name, the remote host, or the date.

```
1  $ cat notifyLogin
2  #!/bin/sh
3  [ "$PAM_TYPE" = "open_session" ] || exit 0
4  {echo "User: $PAM_USER"
5   echo "Ruser: $PAM_RUSER"
6   echo "Rhost: $PAM_RHOST"
7   echo "Service: $PAM_SERVICE"
8   echo "TTY: $PAM_TTY"
9   echo "Date: `date`"
10  echo "Server: `uname -a`"
11  echo "PID: $$"
```

```
12  echo "PPID: $PPID"
13  } | ncat -U --send-only /var/run/seePam
```

3) iptables: The iptables plug-in enforces the network access properties. For instance, the iptables plug-in can allow network communications on a specific port or from a given IP address.

The Access properties in the use-case's policy are used to open some ports. For instance, the access property `Access(`
`SSHPort, AnyIP)` is enforced using the following iptables rule:

```
iptables -I INPUT -m state --state NEW -p tcp --dport 22 -j
    ACCEPT
```

This plug-in can be requested to apply additional capabilities by other plug-ins. For instance, the SSH tunneling plug-in can dynamically request a specific port to be opened by the iptables plug-in.

4) SSH Tunneling: The SSH Tunneling plug-in enforces the creation of confidential tunnels between machines inside and outside the Cloud. Apart from the infrastructure in the Cloud, the Airport Management use-case includes physical machines located in the airport. As part of the use-case, we need to monitor what is displayed on the machines from the Cloud application. The remote port forwarding process comes with a solution to this issue allowing flows redirection.

The remote port forwarding process ensures the confidentiality of the communication, because SSH is an encrypted protocol. Furthermore, thank to the public key cryptography, both sides of the communication channel are authenticated.

The communication between the machines is essential since the SSH server machine has to allocate its own local ports. They will be assigned to a SSH client in order to allow it to do the port forwarding. The enforcement is made of 3 steps: 1) the SSH server machine allocates a local port for the client to set up the tunnel, 2) the SSH client gets the allocated port, and 3) the SSH client creates the remote tunnel.

Fig. 6 shows this process for a remote port forwarding tunnel creation using the port 5900 as the port on client machine.
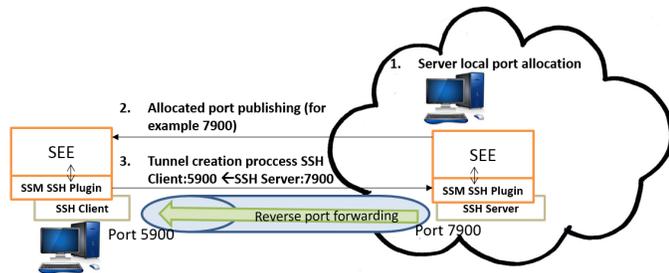

Fig. 6: Tunnel creation process example

To enforce the tunnel, the command `ssh -R 7900:0.0.0.0:5900 ctseed1@reverse-seed4c` is executed, where 7900 is the allocated port for the SSH server, 5900 is the objective port for the SSH client, `reverse-seed4c` is the server's hostname, and `ctseed1` is the user on the SSH server machine used by the client machine.

D. Assurance

The Assurance Model used in the airport management use-case is based on the security policy and focuses on monitoring the effectiveness of the security mechanisms. The model checks that deployed security mechanisms (*e.g.,* SELinux, Iptables, and OpenSSH tunnel) are running as expected. It also checks that the security properties are fulfilled, in terms of data integrity, data confidentiality, and data availability.

For instance, the enforcement of the property `Integrity (ConfigAODB, AdminRoot)` (line 5 of the security policy in Section IV-B) can be checked using the script from listing 1. This script is generated by the SE^E during the enforcement step (Section IV-C), depending on the properties of the security policy.

```
1  $ cat BM_fileInt-1.1.sh
2  #!/bin/bash
3  RET=$XCCDF_RESULT_PASS
4  check_write(){su -c "test -w "$1"" $2; return $?;}
5  FILES=[...]     # list of files in integrity property
6  USERS=[...]     # list of all users
7  OK_USERS=[...]  # list of authorized users
8
9  for file in "${FILES[@]}" ; do
10   for user in "${USERS[@]}" ; do
11    check_write $file $user
12    WRITE_OK=$?
13
14    if [[ " ${OK_USERS[@]} " =~ " $user " ]] ; then
15     if [[ $WRITE_OK -ne "0" ]] ; then
16      RET=$XCCDF_RESULT_FAIL
17      echo "Unexpected access denial: $user->$file"
18     fi
19    else
20     if  [[ $WRITE_OK -eq "0" ]] ; then
21      RET=$XCCDF_RESULT_FAIL
22      echo "Unauthorized access: $user->$file"
23     fi
24    fi
25   done
26  done
27  exit $RET
```

Listing 1: Script checking the integrity of a file

The script `BM_fileInt-1.1.sh` checks the integrity of a file by testing which users are allowed to write it. Line 4 defines a function to check if a file can be written by a specific user. Lines 5 to 7 get the files and users involved in the property (not detailed here due to lack of space). Then, the script loops over the files (line 9) and the users (line 10) and tries to open the files for writing (line 11). If the property and the test result do not match (lines 15 and 20), the return value is set to `XCCDF_RESULT_FAIL` (lines 16 and 21), so that the script will exit with a failure. Otherwise, the script exits with the return value `XCCDF_RESULT_PASS`, indicating that the integrity property has been properly enforced.

As presented before, the assurance framework is steered by 3 files, namely the Assurance Profile (AP), the Operational Profile (OP) and the XCCDF.

The excerpt of the **Assurance Profile** presented in Listing 2 defines one Security Assurance View (SAV) with two Operational Measurement Requirements (OMRs), `OMR_1` and `OMR_3` (lines 10-11), needed for the evaluation of data integrity (lines 7-13).

```
1  [...]
2  <SecurityAssuranceView id="SAV_1">
3   <Statement>Security Functions effectiveness</Statement>
```

```
4  <SAVObject id="1_Data_Int">
5   <Description>Data Integrity</Description>
6   <MetricsAggregFunction>#%t</MetricsAggregFunction>
7   <Metric id="SF_Int_Active">
8    <Description>Availability of security functions affecting
          data integrity</Description>
9    <ReqAggregFunction>#t==##</ReqAggregFunction>
10   <ConcernedMeasurementReq>OMR_1</ConcernedMeasurementReq>
11   <ConcernedMeasurementReq>OMR_3</ConcernedMeasurementReq>
12   [...]
13  </Metric>
14  [...]
15 </SAVObject>
16 [...]
17 </SecurityAssuranceView>
18 [...]
```

Listing 2: AP file for the Airport Management use case.

The **XCCDF** file in Listing 3 defines the last step of the measurement chain. It specifies the assurance checks (with their related scripts, for example `BM_fileInt-1.1.sh`, line 14) that have to be executed to collect the base measures (here, `BM-fileInt-1.1`, lines 9 to 16) needed to evaluate upper levels of the assurance model.

```
1  [...]
2  <Profile id="properties_IO">
3   <description>Properties Assurance</description>
4   <select idref="BM-fileInt-1.1" selected="true" />
5   <select idref="BM-fileConf-1.1" selected="true" />
6   <select idref="BM-netConf-1.1" selected="true" />
7  </Profile>
8  <Group id="properties_group">
9   <Rule id="BM-fileInt-1.1" selected="true">
10   <title>File Integrity</title>
11   <description>Check that file integrity is enforced</
          description>
12   <check system="http://open-scap.org/page/SCE">
13    <check-import import-name="stdout" />
14    <check-content-ref href="BM_fileInt-1.1.sh"/>
15   </check>
16  </Rule>
17  [...]
18 </Group>
19 [...]
```

Listing 3: XCCDF file for the Airport Management use case.

In order for the Assurance Profile and the XCCDF file to inter-operate, the **Operational Profile** (Listing 4) links the Operational Measurement Requirements `OMR_3` of the Assurance Profile (lines 13 to 18) with the Based Measures `BM-fileInt-1.1` of the XCCDF file (lines 3 to 9). It also specifies the machine from which to collect this data (line 7).

```
1  [...]
2  <DerivedMeasures>
3   <DerivedMeasure id="DM-fileInt-1.1-musik1">
4    <Description>Check that file integrity is effective</
          Description>
5    <InterpretFunction>"pass".equals($0)</InterpretFunction>
6    <ConcernedBaseMeasure>BM-fileInt-1.1</ConcernedBaseMeasure
          >
7    <ConcernedDevice>Musik1</ConcernedDevice>
8    <Periodicity>180000</Periodicity>
9   </DerivedMeasure>
10   [...]
11 </DerivedMeasures>
12 <MeasurementRequirements>
13  <MeasurementRequirement id="OMR_3">
14   <MRAggregFunction>#t==##</MRAggregFunction>
15   <DerivedMeasure>DM-fileInt-1.1-musik1</DerivedMeasure>
16   <DerivedMeasure>DM-fileInt-1.1-musik2</DerivedMeasure>
17   <DerivedMeasure>DM-fileInt-1.1-db</DerivedMeasure>
18  </MeasurementRequirement>
19   [...]
20 </MeasurementRequirements>
21 [...]
```

Listing 4: OP file for the Airport Management use case.

The Assurance Collector Engine executes the script `BM_fileInt-1.1.sh` (listing 1) in order to check to enforcement of integrity properties in the security policy (here, the property on line 5 of the policy).

Both the Assurance Profile and the Operational Profile are imported in the Assurance Modeling Tool and derived into the Airport Management Assurance Model, displayed in Fig. 7 by the Assurance Visualization Tool. The model shows the Security Assurance Views defined in the Assurance Profile, in this case the Security Functions effectiveness view, with its corresponding measurement requirements fed by the assurance checks. The left framework of the model allows the navigation by the model structure and shows the assurance compliance in a colour basis. The right framework shows the details on the selected model component. In this case it shows the base measures corresponding to SELinux (MAC) mechanisms status, but the results obtained from the integrity property verification can also be displayed.
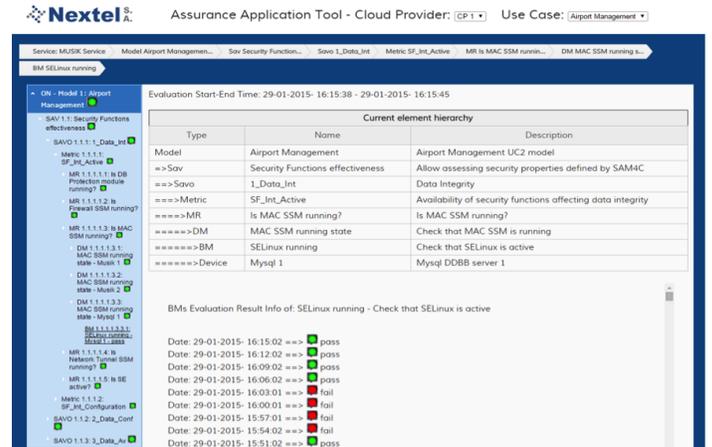

Fig. 7: Airport Management Assurance Model Evaluation and Visualization.

E. Results

Table I presents some statistics concerning the security policy for this usecase.

Number of	
Templates	8
Properties	47
For the client node	1
For the proxy VM	7
For the application VM	12
For the database VM	27
SSMs collaborating to enforce the security properties (SELinux, iptables, PAM, SSH, Oscap)	5
Assurance scripts for the properties	8
Assurance scripts for the SSMs	4

TABLE I: Use-case Policy Statistics

As we can see, the policy for this use-case uses only 8 different properties templates, since our high-level properties cover a wide range of security needs. The policy itself uses

50 contexts and 47 properties for the protection of the whole use-case, which is a very low number considering all the security functionalities covered. Moreover, this policy is entirely generated from a GUI, so the Cloud tenant does not have to write these contexts and properties himself. Besides, this policy manages both the enforcement and the assurance, so that Cloud tenant has information about the status of the enforcement, through a graphical dashboard.

V. CONCLUSION

In this paper, we have presented a solution to specify, enforce and assure security properties in a Cloud environment. Our solution handles the enforcement by re-using existing security and assurance mechanisms, such as SELinux, iptables, PAM, SSH, or Oscap. Our solution is composed of several elements: 1) a language that can express the security and assurance properties independently from the system, the resources naming, and the available mechanisms, 2) an enforcement engine, the SE^E, that receives the properties and enforces them by configuring existing mechanisms, and 3) an assurance framework that models, measures, aggregates, evaluates and presents the security assurance results. Our solution has shown its efficiency on a complete industrial use-case for airport system management: 1) the policy expressing the security requirements of the use-case has been defined, 2) the policy has been enforced using several mechanisms that collaborate to offer an end-to-end protection (across the different machines), and 3) the assurance framework has confirmed the proper enforcement of the security policy.

In our future works, we will define generic policy templates that could be used to secure the system base, in addition to the policy on the tenant's software architecture. This added protection would improve the overall security of the system. Besides, we plan to extend the language so that the results generated by the assurance framework are sent back to the enforcement engine: this would allow the enforcement engine to update the configuration of the security mechanisms to adapt the protection in case something is not working as expected.

REFERENCES

[1] L. Bobelin, A. Bousquet, J. Briffaut, J.-F. Couturier, C. Toinard, E. Caron, A. Lefray, and J. Rouzaud-Cornabas, "An advanced security-aware cloud architecture," in *High Performance Computing & Simulation (HPCS), 2014 International Conference on*. IEEE, 2014, pp. 572–579.

[2] United States Department of Defense, "Trusted Computer System Evaluation Criteria (Orange Book)," Tech. Rep., 1985.

[3] Cloud Security Alliance, "Cloud Adoption Practices and Priorities Survey Report," 2015, https://cloudsecurityalliance.org/download/cloud-adoption-practices-priorities-survey-report/.

[4] T. Jaeger and J. Schiffman, "Outlook: Cloudy with a Chance of Security Challenges and Improvements," *IEEE Security & Privacy Magazine*, vol. 8, no. 1, pp. 77–80, Jan. 2010. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5403158

[5] J. Y. Halpern and V. Weissman, "Using first-order logic to reason about policies," *ACM Trans. Inf. Syst. Secur.*, vol. 11, no. 4, pp. 21:1–21:41, Jul. 2008. [Online]. Available: http://doi.acm.org/10.1145/1380564.1380569

[6] R. Sandhu, R. Boppana, R. Krishnan, J. Reich, T. Wolff, and J. Zachry, "Towards a Discipline of Mission-Aware Cloud Computing," in *Proceedings of the 2010 ACM workshop on Cloud computing security workshop*, 2010, pp. 13–18.

[7] K. Twidle, N. Dulay, E. Lupu, and M. Sloman, "Ponder2: A policy system for autonomous pervasive environments," in *Autonomic and Autonomous Systems, 2009. ICAS'09. Fifth International Conference on*. IEEE, 2009, pp. 330–335.

[8] S. Pearson, V. Tountopoulos, D. Catteddu, M. Südholt, R. Molva, C. Reich, S. Fischer-Hübner, C. Millard, M. Lotz, M. G. Jaatun *et al.*, "Accountability for cloud and other future internet services." in *Cloud-Com*, 2012, pp. 629–632.

[9] J. Chen, Y. Wang, and X. Wang, "On-demand security architecture for cloud computing," *Computer*, vol. 45, no. 7, pp. 73–78, 2012.

[10] A. Wailly, M. Lacoste, and H. Debar, "Vespa: multi-layered self-protection for cloud resources," in *Proceedings of the 9th international conference on Autonomic computing*. ACM, 2012, pp. 155–160.

[11] A. D. Keromytis, R. Geambasu, S. Sethumadhavan, S. J. Stolfo, J. Yang, A. Benameur, M. Dacier, M. Elder, D. Kienzle, and A. Stavrou, "The meerkats cloud security architecture," in *Distributed Computing Systems Workshops (ICDCSW), 2012 32nd International Conference on*. IEEE, 2012, pp. 446–450.

[12] R. He, M. Lacoste, and J. Leneutre, "A policy management framework for self-protection of pervasive systems," in *Autonomic and Autonomous Systems (ICAS), 2010 Sixth International Conference on*. IEEE, 2010, pp. 104–109.

[13] C. Ngo, P. Membrey, Y. Demchenko, and C. de Laat, "Policy and context management in dynamically provisioned access control service for virtualized cloud infrastructures," in *Availability, Reliability and Security (ARES), 2012 Seventh International Conference on*, Aug 2012, pp. 343–349.

[14] V. C. Hu, E. Martin, J. Hwang, and T. Xie, "Conformance checking of access control policies specified in xacml," in *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, vol. 2. IEEE, 2007, pp. 275–280.

[15] C. A. Ardagna, S. De Capitani di Vimercati, S. Paraboschi, E. Pedrini, and P. Samarati, "An xacml-based privacy-centered access control system," in *Proceedings of the first ACM workshop on Information security governance*. ACM, 2009, pp. 49–58.

[16] R. Haines, *The SELinux Notebook - Third Edition*, 2012.

[17] V. Samar, "Unified login with pluggable authentication modules (pam)," in *Proceedings of the 3rd ACM conference on Computer and communications security*. ACM, 1996, pp. 1–10.

[18] O. Andreasson, "Iptables tutorial 1.2. 2," 2001.

[19] T. Ylonen and C. Lonvick, "The secure shell (ssh) connection protocol," 2006.

[20] M. Li, S. Yu, Y. Zheng, K. Ren, and W. Lou, "Scalable and secure sharing of personal health records in cloud computing using attribute-based encryption," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 24, no. 1, pp. 131–143, 2013.

[21] A. López-Alt, E. Tromer, and V. Vaikuntanathan, "On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption," in *Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing*, ser. STOC '12. New York, NY, USA: ACM, 2012, pp. 1219–1234. [Online]. Available: http://doi.acm.org/10.1145/2213977.2214086

[22] S. Haddad, S. Dubus, A. Hecker, T. Kanstrén, B. Marquet, and R. Savola, "Operational security assurance evaluation in open infrastructures," in *Risk and Security of Internet and Systems (CRiSIS), 2011 6th International Conference on*. IEEE, 2011, pp. 1–6.

[23] C. Criteria, "Common Criteria for Information Technology Evaluation v3.1 (ISO/IEC 15408)," 2012.

[24] ETSI, "ETSI TR 187 023: Security Assurance Profile for Secured Telecom Operations Statement of needs for security assurance measurement in operational telecom infrastructures," 2012, www.etsi.org/deliver/etsi_tr/187000_187099/187023/01.01.01_60/tr_187023v010101p.pdf.

[25] SCAP, "SCAP: Security Content Automation Protocol," 2014, http://scap.nist.gov/.

[26] OpenSCAP, "OpenSCAP Website," 2014, http://www.open-scap.org.