A revised version of this manuscript will appear in the Proceedings of the 31st ACM Symposium on Applied Computing (SAC 2016) to be held in Pisa, Italy, 4-8 April 2016. If you wish to refer to this work, please use the following citation: M. Anisetti, C. A. Ardagna, E. Damiani, F. Gaudenzi, "A Certification Framework for Cloud-based Services," in Proc. of the 31st ACM Symposium on Applied Computing (SAC 2016), Pisa, Italy, April 2016.

A Certification Process for Cloud-based Services

Marco Anisetti, Claudio A. Ardagna, Ernesto Damiani, Filippo Gaudenzi
Dipartimento di Informatica
Università degli Studi di Milano
Crema, Italy, 26013
Email: *firstname.lastname@unimi.it*

***Abstract*—Lack of trust and transparency are among the main reasons hindering adoption of cloud computing. Users in fact cannot inspect neither their applications nor the treatment of their data, and have little or no guarantees about their security. In this context, there is a pressing need for assurance techniques supporting some key properties of cloud services and applications. Cloud security certification is a major assurance technique that has been proposed to increase cloud security, trust, and transparency. However, certification is a tedious, costly, and time-consuming process for the provider that wants to certify one of its services/applications. In this paper, we propose a cloud certification framework implementing a certification process and a cloud engineering methodology based on it, which supports providers in the design and development of ready-to-be-certified services/applications.**

***Index Terms*—Certification-aware cloud engineering, Security certification, Testing**

I. Introduction

The widespread diffusion of the cloud computing paradigm is radically changing traditional IT provisioning and procurement [1]. Users can remotely access services on demand at application (SaaS), platform (PaaS), and infrastructure (IaaS) layers, benefiting from a simplified IT management and an increased service flexibility. Despite the clear benefits provided by cloud computing, cloud customers lose full control of their data and services.

Recently, the research community has focused on increasing the trust and transparency of cloud and service-based systems by defining cloud-specific security assurance techniques [2], [3], [4], [5], [6], [7], [8]. However, while several assurance techniques have been defined for such systems, the market lacks a complete security assurance framework. Among assurance techniques, certification-based assurance has received a lot of attention in the last few years, resulting in the definition of a number of certification schemes [2], [3], [9], [10], [11], [12]. Certification schemes provide an independent evaluation process of systems, which results in a signed certificate including a set of claims and the evidence supporting them. Evidence comes from testing, monitoring, and formal proofs. However, certification is a tedious, costly, and time-consuming process, which requires a lot of post-implementation activities, including document writing and code adaptation.

In this paper, we present a test-based security certification framework for cloud-based processes. Our framework has a twofold value: *i)* it provides a set of modules and components managing the certification process of a generic cloud-based system; *ii)* it defines a methodology for certification-aware cloud engineering, including a set of guidelines to design and develop certification-ready services/applications. The framework relies on XML-based documents, namely, Certification Model (CM) Template and Instance [13]. CM Template specifies abstract configurations and security mechanisms needed to certify a given class of systems for a given property. It represents a source of requirements for a provider that wants to certify its services/applications according to the template. CM Instance is a refinement of the CM Template including all information on configurations and evaluation activities. The contribution of our framework is threefold: *i)* it supports online and automatic test-based certification of services/applications; *ii)* it defines an environment and methodology where providers are able to develop certification-aware services/applications; *iii)* it provides a set of tools supporting the certification-based chain of trust for cloud environments introduced in [13].

This paper is organized as follows. Section II presents a certification process for the cloud. Section III discusses requirements for the certification process in Section II and for certification-aware cloud engineering. Section IV describes the role and structure of CM Template and CM Instance. Section V presents the architecture of our certification framework and different deployment strategies. Section VI illustrates some aspects on the implementation. Section VII presents the working of the proposed framework in a real certification scenario. Section VIII presents related work and Section IX draws our conclusions.

II. Certification Process

A certification process for cloud-based services is a collaborative effort involving four main actors: *i)* a *service provider* developing cloud-based applications to be certified; *ii)* a *cloud provider* that gives access to its backend to support application certification; *iii)* a *certification authority* managing the certification process; *iv)* an *accredited lab*, delegated by the certification authority, responsible for cloud system evaluation. In our cloud certification process, the term accredited lab refers to both an online actor that uses a certification framework to produce evidence supporting the certification of a cloud-based service, and to the framework that automatically verifies existing certificates and awards new ones following guidelines by the certification authority.

A certification process aims at producing or maintaining a security certificate for a target of certification (i.e., the system –

or a portion thereof – under evaluation), including all evidence supporting a given property for it. Differently from traditional certification processes for software and services [2], [9], [10], our cloud certification process follows two steps. First, the accredited lab statically evaluates the target of certification in a lab environment and collects the evidence supporting the issuing of a certificate (*pre-deployment evaluation*). Then, it continuously checks the validity of the certificate in the real deployment environment (*in-production evaluation*). We note that, when pre-deployment evaluation is not possible (e.g., a suitable lab environment cannot be provided), evidence collection and certificate issuing can be done during in-production evaluation. When in-production evaluation is not possible (e.g., to test robustness against DDoS attacks), a static certificate is issued during pre-deployment evaluation and never checked for validity at runtime. In the following of this section, for simplicity, we focus on the two-step certification process.

As discussed in [13], our certification process is driven by *i)* the *security property* to be certified, defined as an abstract property (e.g., confidentiality, integrity, and availability) refined using attributes on security mechanisms and/or security attacks; *ii)* the *target of certification* (ToC), which defines the certification perimeter as a set of mechanisms, belonging to one or more cloud layers, and the layer of certificate binding; *iii)* a description of *evaluation activities*, as a set of configurations for the validation of the property for the ToC. It then returns the evidence produced by evaluation activities, a description of the collection process, and a reference to the evaluated ToC. Security property, ToC, evaluation activities, and evidence are part of more general documents representing the inputs (I) and outputs (O) of the certification process [13] as follows.

- *Certification Model (CM) Template (I).* It is an abstract representation of the process inputs, which specifies high-level configurations and activities for the certification of a property for a given class of ToCs. It is signed by a certification authority.
- *Certification Model (CM) Instance (I).* It is a refinement of the CM Template, driving the certification process. It includes specific information on configurations and activities to be executed on the ToC under evaluation. Checks of consistency between CM Template and CM Instance, and between the deployed ToC and its modeling in the CM Instance are manually done by the accredited lab during pre-deployment evaluation.
- *Certificate (O).* It contains all information and results of the evaluation process, including the evidence supporting the validity of the certificate for the ToC. The certificate is bound to the ToC supporting certification-aware service/application procurement.

Figure 1 shows the conceptual framework at the basis of our certification process [13]. Upon cloud/service providers deploy their system (i.e., a cloud service or application) in the cloud (ToC), they engage with the certification authority to certify it for a given property by selecting a CM Template. A

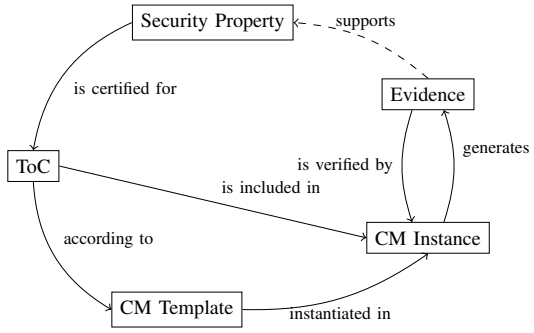

Fig. 1. Conceptual framework

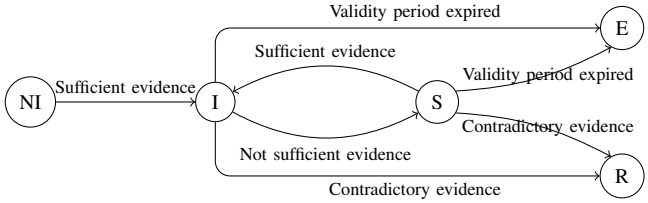

Fig. 2. Life cycle with states Not Issued (NI), Issued (I), Suspended (S), Expired (E), Revoked (R)

CM instance is then produced by the accredited lab, with the assistance of the provider, according to the selected template. By executing the CM instance, the accredited lab can generate evidence during the pre-deployment and in-production evaluation. If evidence is sufficient, a security certificate is awarded to the service/application, which includes the certified property and ToC, and the supporting evidence. Otherwise, the evidence is used to either release a new certificate or evaluate the validity of an existing certificate according to its life cycle.

Figure 2 presents an example of certificate life cycle from its issuing to expiration or revocation [13]. It is modeled as a deterministic finite state automaton, where each vertex is a certificate state (e.g., issued, revoked) and each edge is a transition between two states, regulated by a condition over certificate's evidence. The life cycle in Figure 2 specifies abstract conditions that can be mapped on specific events triggering transitions on the basis of the considered scenario.

III. Requirements

We claim that a security certification process for the cloud (Section II) should support the following basic requirements.[1]

- *Distributed deployment*: the framework *MUST* be developed in a way that allows its deployment (all or in part) over the cloud.
- *Multi-layer and multi-target certification*: the framework *SHOULD* be able to certify security properties of services and applications that can span different layers of the cloud stack (i.e., service, platform, and infrastructure layers). For instance, the confidentiality of data exchanged and

[1]For the sake of clarity, requirements in this section are prioritized according to modalities *MUST*, *SHOULD*, *MAY*.

stored by a cloud application depends also on the security mechanisms implemented at platform and infrastructure layers (e.g., the security mechanism protecting the virtual storage).
- *Property-driven certification*: the framework *SHOULD* implement a certification process driven by the property to be certified for the system under certification.
- *Evidence-based certification*: the framework *MUST* support a certification process that provides a set of evidence proving a security property and included in the security certificate. The evidence *SHOULD* be collected in a standardized way (e.g., by agents and probes deployed within the system under certification), according to different collection mechanisms (e.g., testing, monitoring).
- *Incremental certification*: the framework *SHOULD* provide an incremental approach, meaning that evidence and corresponding certificates can be re-validated at runtime using an automatic and independent process. The security certificate awarded to a given system as the result of a certification process execution should undergo a continuous validation according to a specific life cycle (see Section II).
- *Fully automatic configuration*: the framework *MAY* provide auto-configuration of agents and probes collecting evidence, thus supporting automatic execution of certification activities and collection of incremental evidence.
- *Extendible deployment*: the framework *MUST* be fully customizable and extendible to adapt to changing conditions.
- *Trusted implementation*: the framework and its activities *MUST* be trusted, increasing the confidence of the users in the correctness of the process and corresponding results.

Additional requirements supporting certified service engineering can be added as follows.

- *Certification-aware cloud engineering*: the framework *SHOULD* provide a certification-aware cloud engineering approach that can be integrated with traditional development methodologies with limited effort.
- *Guided security mechanism development*: the framework *MUST* guide the development of those security mechanisms needed to certify a given property for a given system.
- *Step-by-step deployment*: the framework *MUST* drive the deployment of all components needed to support the execution of the certification process.
- *Certification process independent*: the framework *SHOULD* implement a methodology for certification-aware cloud engineering that is independent by the corresponding certification process.

We now describe the implementation of our framework supporting the above sets of requirements. For simplicity, we will focus on in-production evaluation only.

IV. CM Template and CM Instance

As said previously, in our framework CM Template and CM Instance drive the certification process from the initial description of assurance activities (*CM Template*) to the incarnation of these activities into a concrete certification process (*CM Instance*).[2] More concretely, CM Template is a formal guide on how to design and develop a cloud service/application, such that, *i)* a given security property holds and *ii)* the evidence for this property can be collected by the certification framework in a straightforward way. CM Instance is used as a concrete description of the certification process and is consumed by the certification framework to configure all collection activities. It targets the cloud service/application under certification (ToC), defines the framework deployment strategy, and manages the certificate life cycle. Below, we provide a detailed description of the main elements of CM Template and CM Instance.

A. CM Template

CM Template is composed of the following main elements.

- **ToC**: it describes the service/application under certification (see Section II), specifying its class and the relevant cloud layer.
- **Collectors**: it contains a set of elements, called **AbstractCollector** and **Collector**, whose goal is to describe the test-based evidence collection process for a given property and ToC. CM Templates only specify elements **AbstractCollector**, that is, the flows at the basis of the evidence collection process, using a model-based approach. Each **AbstractCollector** describes testing activities without the definition of the real test cases to be executed on the ToC (see [2] for a description of how test cases can be automatically generated from the specified model). Its scope is to define a set of testing flows for a specific test type (e.g., random input, input partitioning) and test category (e.g., functional, robustness, penetration).
- **Signature**: it provides the signature of the certification authority who signed (and took responsibility over) the template. The signature is fundamental for establishing the trust of the consumer in the framework implementing the certification process. The trust model is described in [13] and relies on the CM Instance refining the CM Template.

B. CM Instance

Once a CM Template is defined and signed by a certification authority, it can be used as the basis to define a CM Instance. CM Instance includes all elements of CM Template, and extends it by adding new elements and/or by specializing the values of the ones in common, according to a specific certification scenario. It is composed of the following main elements.

[2]As already said, both CM Template and CM Instance are XML-based documents and their schema can be downloaded at [14].

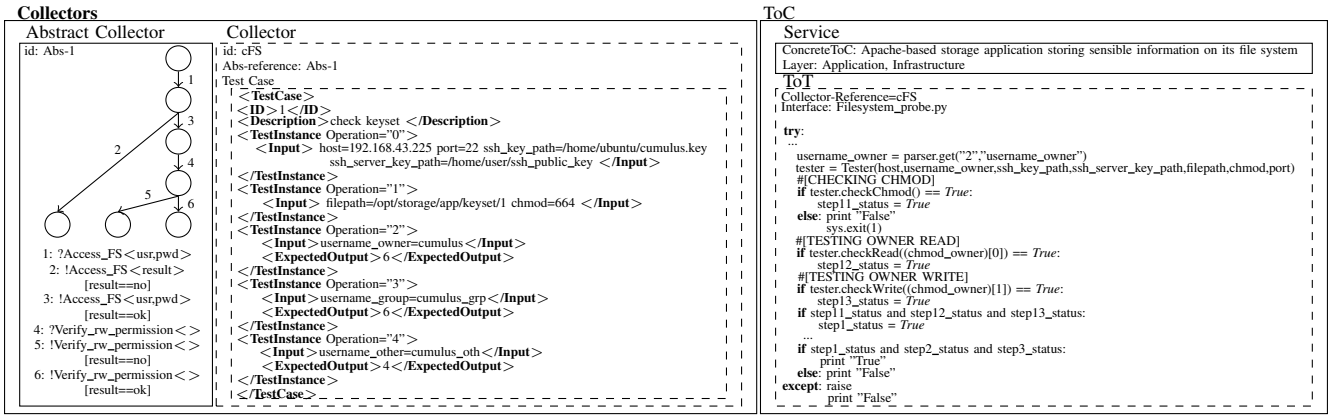

Fig. 3. CM Template and CM Instance

- **ToC**: it refines element **ToC** in CM Template by defining a set of elements **ToT** (Target of Test). Each element **ToT** first specifies the reference to the end points (hooks) needed to evaluate the ToC. It then drives the execution of test cases through probes and the collection of the results of their execution out of the specified hooks.
- **Collectors**: it extends element **Collectors** in the CM Template with a set of elements **Collector** and, if needed, by refining elements **Abstract Collector**. Each **Collector** is defined according to an element **AbstractCollector** and specifies the real test cases to be executed. We note that a **ToT** manages testing activities following a testing flow specified by **AbstractCollector** and the test cases in **Collector**.
- **Aggregator**: Each **Collector** is associated with an element **Aggregator** that is used to calculate the status of the entire testing activity carried out according to the collector itself. The status is a boolean value, where *success* (1) and *failure* (0) are defined as a metric over the testing results produced by each **Collector**. As an example, *success* is returned if a given percentage of the test cases specified in an element **Collector** are successful.
- **LifeCycle**: it describes the certificate life cycle (see Section II and Figure 2). It specifies all conditions regulating transitions in terms of boolean expressions of elements **Aggregator**.
- **Signature**: it provides the signature of the certification framework who signed the CM Instance via delegation from the certification authority. This signature is necessary to establish a chain of trust grounded on the produced certificates.

Figure 3 presents an example of CM Template and CM Instance. Black rectangles represent elements that are defined in both CM Template and CM Instance, while dashed rectangles represent elements that are only defined in CM Instance. Our example considers a CM template for a generic service deployed in a service container (ToC) and security property *confidentiality* based on *file system access control*. **AbstractCollector**, which defines the testing flow as an automaton evaluating access grants on a specific file, requires to first log into the service and then checks the correctness of read/write (*rw*) permissions when the target file is accessed by a user. CM Instance in Figure 3 refines CM Template by specifying a *storage application* deployed in an *Apache* container as our ToC. Following the flow in **AbstractCollector**, **ToT** executes all testing activities according to the inputs (elements **Input**) in the **Collector**. The life cycle in the CM Instance (not shown in Figure 3 for simplicity) includes a transition from *issued* to *suspended* when, according to element **Aggregator**, *rw* validity verification fails. For instance, let us assume that the **Aggregator** defines *success* as the case where all test instances (elements **TestInstance**) in the **Collector** are successful, and that operation 3 in the test case of collector *cFS* returns a *failure*. In this case, the certificate status changes from *issued* to *suspended*. A more complete use case example is presented in Section VII.

V. Framework Architecture

Our framework is supported by a master-slave architecture composed of two main modules: *Test Manager* (TM) and *Test Agent* (TA). *Test Manager* – the master – is the owner of the certification process. It *i)* provides external API to manage (e.g., start, stop) a certification process, and to manage, retrieve, and modify corresponding CM Instances and certificates, *ii)* processes the CM Instance to extract the configurations and test cases needed for setting up the certification process and for initializing the slaves, and *iii)* manages the life cycle of existing certificates collecting aggregated results on testing activities. *Test Agent* – the slave – is responsible for testing the ToC. It *i)* deploys the testing probes (i.e., the test scripts in the ToT described in Section IV) and connects them to the hooks (i.e., the end points needed to execute test cases described in Section IV), *ii)* runs the test cases specified in the collectors, and *iii)* returns the results of the testing activities to TM. TM deals with multiple parallel certification processes, and supports a scalable and flexible solution by deploying different TAs on the basis of available resources. The scope

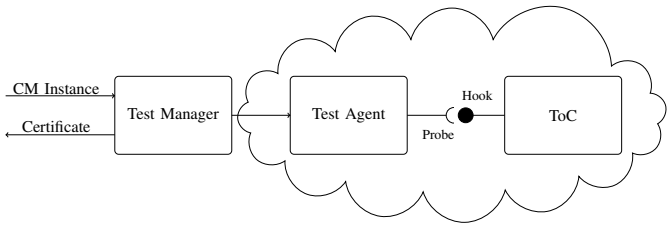

Fig. 4. A simplified view of the framework architecture

is to minimize the performance impact on the target cloud.

Figure 4 shows the relation and data flows between TM, TA, probes injecting test cases via hooks, and ToC. After a CM Instance is sent to TM, our certification process starts.[3] The CM Instance is parsed and split into pieces driving the activities of the running TA(s). If TA is not deployed yet, then TM deploys it together with the required configurations, otherwise TA updates its configurations. TA sends back all evidence to the TM when available. The hooks are provided by the cloud/service provider and are generally protected by means of access control systems. TM aggregates the results returned by TA and either releases a new certificate, if possible, or manages the life cycle of an existing certificate. In the following, we describe all its components in more detail.

A. Components

Figure 5 shows the internal components of TM and TA, and their communications. TM consists of different components that can be grouped into 3 main families: *i)* communication components (i.e., Test Management Interface, Agent Broadcaster/Receiver, Agent Deployer, and Queue Communication System), *ii)* data components (i.e., Data Manager), and *iii)* engine components (i.e., XML Parser, LifeCycle Manager, and Aggregator).

- **Test Management Interface**: it is the front-end with our certification process. It offers API to start/stop a certification process, retrieve certificates, and inspect certificate life cycle.
- **Data Manager**: it manages persistent information. It uses a database to store all artifacts including CM Instances and related certificates.
- **XML Parser**: it implements three main functionalities. First, it parses all documents, such as CM Instances and certificates, which are used by Test Management Interface. Second, it generates certificates from scratch when requested by Test Management Interface. Third, it parses CM Instance and creates commands for Test Agent.
- **Agent Broadcaster/Receiver**: it is responsible to send and collect messages from Test Agents. It is also responsible to dispatch the Test Agent messages to other TM components.

[3]We note that if the CM Instance is an updated version of a running one, the framework just updates the configurations and refreshes test cases if required.

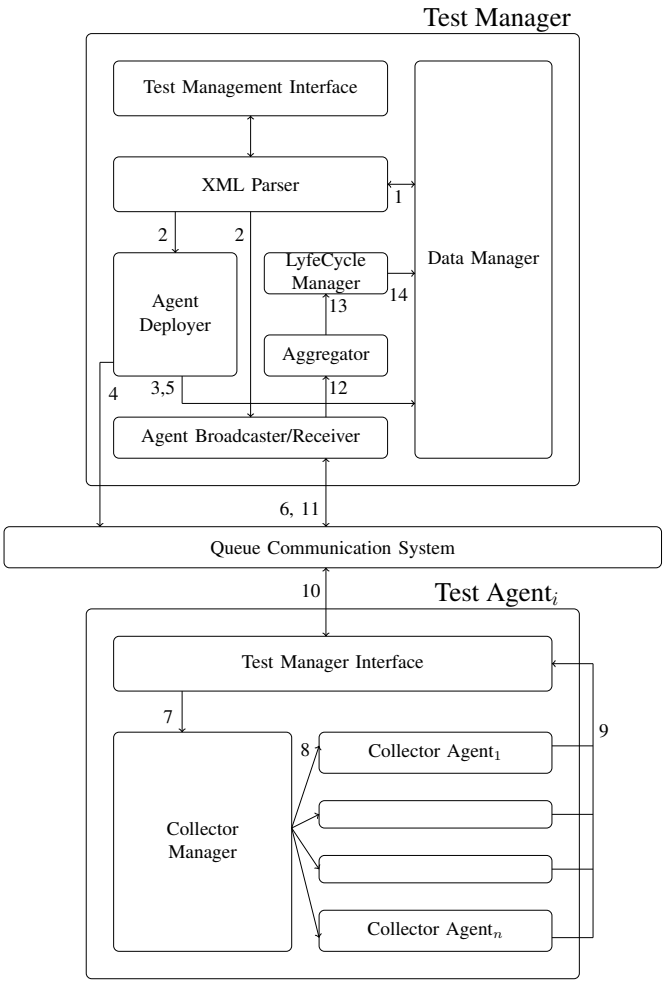

Fig. 5. Test Manager (TM) and Test Agent (TA)

- **Queue Communication System**: it forwards messages from Agents to Agent Broadcaster/Receiver and vice versa. It also manages agent authentication and message queues.
- **Agent Deployer**: it interacts with the cloud APIs to deploy a TA. We note that current solution considers OpenStack APIs, though it is general enough to support other frameworks. Agent Deployer then interacts with Queue Communication System for registering TA and making it able to authenticate with TM. TA, once deployed, can authenticate to Agent Broadcaster/Receiver through Queue Communication System. Agent Deployer also maintains a list of deployed and running TAs and identifies them in terms of running collectors and test cases.
- **Aggregator**: it evaluates the testing evidence (i.e., collector results) sent by TA via Agent Broadcaster/Receiver and notifies the LifeCycle Manager. We note that this evaluation is made by computing aggregation metrics, specified in element Aggregator of CM Instance, on the results of the testing activities.

- **LifeCycle Manager**: it receives the Aggregator evaluation and, according to the LifeCycle conditions on transitions, sets the status of the certificate.

TA executes all collectors/test cases requested by TM. Communications between TM and TA are managed by Queue Communication System that serializes the messages, authenticates the agents, and keeps TM load under control. Communications between TA and ToC depend on the hooks, and are configured according to the probes. Test agent consists of the following components.

- **Test Manager Interface**: it receives commands from and sends back test evidence results, retrieved by Collector Agent via Collector Manager, to TM.
- **Collector Manager**: it manages the configuration, deployment, and execution of all relevant Collector Agents, dispatching to them the test cases and setting the relative probes for their connection to the hooks. It also starts, stops, and resets a Collector Agent sending a different or updated configuration, test cases, probes, and hooks.
- **Collector Agent**: it uses probe scripts in ToTs and corresponding test cases to evaluate ToC. Probe scripts read information from test cases and hook configurations, feed the test cases in the attached hook, and, after all test cases have been executed, notify Collector Manager with the results, which are then forwarded to TM. Each Collector Agent is assumed to be independent and thus runs in parallel, while each of its test cases runs sequentially following the testing flow dictated by the abstract collector.

TA keeps detailed information of its activities in logs, which can be retrieved later for audit or a posteriori analysis.

B. Execution flow

Figure 5 shows the execution flow of our process, assuming a CM Instance is already available and ready to be executed. CM Instance is retrieved from the database by *Data Manager* (step 1) and then parsed by *XML Parser* to generate the information to be sent to the *Test Agent* (step 2). *XML Parser* parses the whole CM Instance retrieving all the collectors with corresponding test cases and ToTs, and the information about the TA to be deployed. Then, *Agent Deployer* first checks if a suitable TA is up and running (step 3), otherwise it registers the TA to be deployed with *Queue Communication System* (step 4). This allows TA to communicate with TM. Afterwards, it deploys TA, and internally registers TA in the list of active TAs for further interactions or checks (step 5). Upon TA is up and running, all the required information is submitted to it by *Agent Broadcaster/Receiver* (step 6). TA receives the message and parses it with *Collector Manager* (step 7). Each Collector Agent is deployed in parallel and runs all test cases (step 8). Each test case result is sent back to TM via *Test Manager Interface* (steps 9 and 10). *Agent Broadcaster/Receiver* receives the messages from TA and forwards them to *Aggregator* (steps 11 and 12) that, as soon as it holds enough information, evaluates the collector status according to the corresponding element Aggregator in CM instance. *Aggregator* then notifies *LifeCycle Manager* (step 13), which temporarily saves the collector status, checks at any notification if there is at least a transition to evaluate, and, if yes, evaluates it. If a transition event occurs *Data Manager* updates the status of the corresponding certificate (step 14).

C. Deployment strategies

Of course, the deployment of our framework might be considered invasive by the cloud/service providers that need to open their system to third party Test Agents. In addition, CM Instance may contain information that can be considered confidential by the cloud/service provider, such as, internal configurations, authorization credentials, test cases, and the like. This information is generally associated with hooks that implement the interface with ToC and are used by Test Agents to inject test cases. Even if the certification authority and its certification framework are trusted, the cloud/service providers may be reluctant to open their system to the outside and to release such sensitive data. To handle this problem, we foresee two different framework deployment strategies.

- **Self-assessment test**: the accredited lab of a certification authority and the cloud/service provider collaboratively define a CM Instance containing all sensitive information, and prepare a TA pre-configured with all collectors, probes, and ToTs, which is ready to be connected to the ToC. TA is then released to the provider that deploys it within the system under certification. TM has no control on TA configurations, while it can only call TA to execute pre-configured test cases and collect the corresponding results. In the extreme case, the self-assessment test strategy allows a provider to define a CM Instance in conformance with a CM Template (abstract collectors and ToC) without revealing sensitive information. This information is added only after the TA is deployed in the provider infrastructure.
- **Certification-as-a-service**: cloud/service provider gives to the certification framework direct access to its system (e.g., it opens an SSH port) and releases all sensitive information required by TAs to execute the test cases on the given ToC. TAs are deployed in the system by Agent Deployer and are driven by TM. TM, in fact, sends CM Instance to configure a TA and execute the specified testing activities on the ToC.

Our deployment strategies require two levels of access to the cloud/service provider backend. In the first case (self-assessment test), the certification framework can only access the interface provided by the pre-configured TA and execute the pre-configured test cases. No choice of or changes to testing activities can be done at runtime. In the second case (certification-as-a-service), the certification framework can access the internal interface of the cloud/service provider and execute test cases, which can be selected and modified at runtime. The certification-as-a-service approach is more risky for the providers, since it requires access to internal structures

of the cloud. However, it is more flexible since the framework can automatically adapt to changes in the environment by re-configuring the agents. In the self-assessment test approach an increased security is achieved at a price of a reduced flexibility, since agent re-configuration is in the providers' hands. This can potentially cause a *vendor lock-in* issue and the revocation of a certificate in case the providers are not reacting properly to changes. In other words, certification-as-a-service strategy outperforms the self-assessment test one in terms of resilience, since the certification process is fully under the control of the certification framework, while in the second strategy configurations may require to be updated asynchronously.

It is important to note that CM Instance not only supports and drives the deployment of our certification framework, but it also extends our methodology to certification-aware cloud engineering. In fact, CM Instance represents a source of information that permits a certification-friendly deployment of the service/application under certification by the corresponding provider. The services/applications are configured in a way that supports direct integration with TAs and evaluation activities, following the selected deployment strategy.

VI. Implementation Guidelines

We describe some details of our TM and TA implementation.

TM is a Java-based Axis 2 web service running on Tomcat Apache version 7. External communications are managed by Test Manager Interface through standard SOAP; internal communications with TA are managed by Agent Receiver/Broadcaster through a RabbitMQ Server with security features activated. The communication over RabbitMQ is secured by enabling the mutual authentication and the encryption of the channel. Both TM and TAs have their own users and certificates, which correspond to a specific policy in the RabbitMQ Server. TM connects to multiple TAs using the *publish/subscriber* paradigm, while TA connects back to TM using the *direct queue* paradigm. All documents and building blocks (i.e., CM Instances, agents, certificates) are stored by Data Manager in a MySQL database, while XML document creation and validation are managed by XML Parser using the JAXB library. The JAXB library permits automatic binding between XML Documents and Java Objects based on corresponding XML Schemas. Finally, Agent Deployer is responsible for the deployment of TAs on different cloud providers through the jclouds library. We note that although our Agent Deployer has been developed for OpenStack infrastructure, it can be easily extended for deployment on any infrastructure supported by jclouds library.

TA is implemented as a tamper-resistant virtual machine installing a minimal Ubuntu 14.04 Server distribution, where a RabbitMQ client is configured to communicate with TM. It implements all functionalities to execute the testing activities on a given ToC. In particular, it is composed of: *i)* a daemon in python (implementing Test Manager Interface) listening for commands from TM, *ii)* a thread manager as a python script (implementing Collector Manager) running a given set of Collector Agents in parallel, *iii)* one or more Collector Agents as generic python scripts parsing the collector and executing it via ToC-specific probe scripts. While parsing a collector (see Figure 3), the Collector Agent extracts relevant test cases and organizes them in one or more input files. A single file is generated for each testing flow (i.e., linear independent path) in the abstract collector automaton. The input file is given as input to a probe, that is, a script which implements the testing activities specified in the collector. Collector Agent runs a probe script for every input file ($[probescript].py - i[inputfile] - o[outputfile]$) and collects a set of output files with the results (evidence) of the testing activities. The output files are then returned to TM.

Listing 1 shows an example of a probe script used to test a ToC for property *confidentiality* based on *file system access policy*. We note that probe scripts share the same structure: *i)* a common part for interface purposes (lines 9-31), *ii)* a custom part for parsing test cases in the input files (lines 34-44, 65, 80), *iii)* a custom part launching the test cases (lines 45-64, 66-79, 81-101). We note that probes with fixed-structure help developers in writing their own probes, making them directly executable by the Collector Agent.

```
1  #!/usr/bin/env python
   import sys
3  import string
   import json
5  import ConfigParser, getopt
   from time import sleep
7  from patlib_old import convertChmodToBinary
   from tests import Tester
9  #begin common part#
   def usage():
11  print """\033[1m\033[91mCheck file permissions
        \033[0m\033[0m
    Usage: %s <input> <output>""" % __file__[__file__.
        rfind('/')+1:]
13 if __name__ == '__main__':
    try:
15   opts, args = getopt.getopt(sys.argv[1:], "i:o", ["
        init=","output=","help"])
    except getopt.GetoptError:
17   usage()
     sys.exit(2)
19  for o, a in opts:
     if o in  ("-o","--output"):
21    output_folder = a+".log"
     elif o in  ("-i","--init"):
23    config_file = a
     else:
25    assert False, "unhandled option"
    config = config_file
27  #Parser for inputfile
    parser =  ConfigParser.RawConfigParser(
        allow_no_value=True)
29  with open(config, 'r') as g:
     parser.readfp(g)
31  #end common part
    #begin parsing input and testing
33  try:
     host   = parser.get("0","host")
35   port    = int(parser.get("0","port")) or 22
     if parser.has_option("0","ssh_key_path"):
37    ssh_key_path = parser.get("0","ssh_key_path")
     else:
39    ssh_key_path = "~/.ssh/id_rsa"
     ssh_server_key_path  = parser.get("0","
        ssh_server_key_path")
41   filepath  = parser.get("1","filepath")
     chmod   = parser.get("1","chmod")
43   chmod_owner, chmod_group, chmod_other =
```

```
    convertChmodToBinary(chmod)
username_owner = parser.get("2","username_owner")
tester = Tester(host,username_owner,ssh_key_path,
    ssh_server_key_path,filepath,chmod,port)
#begin testing flow
  step11_status = False
step12_status = False
step13_status = False
step1_status = False
#[CHECKING CHMOD]
if tester.checkChmod() == True:
 step11_status = True
else:
 print "False"
 sys.exit(1)
#[TESTING OWNER READ]
if tester.checkRead((chmod_owner)[0]) == True:
 step12_status = True
#[TESTING OWNER WRITE]
if tester.checkWrite((chmod_owner)[1]) == True:
 step13_status = True
if step11_status and step12_status and
    step13_status:
 step1_status = True
  username_group = parser.get("3","username_group"
    )
tester = Tester(host,username_group,ssh_key_path,
    ssh_server_key_path,filepath,chmod,port)
step21_status = False
step22_status = False
step2_status = False
#[TESTING GROUP READ]
sleep(1)
if tester.checkRead((chmod_group)[0]) == True:
 step21_status = True
#[TESTING GROUP WRITE]
sleep(1)
if tester.checkWrite((chmod_group)[1]) == True:
 step22_status = True
if step21_status and step22_status:
 step2_status = True
username_other = parser.get("4","username_other")
tester = Tester(host,username_other,ssh_key_path,
    ssh_server_key_path,filepath,chmod,port)
#[TESTING OTHER READ]"
step31_status = False
step32_status = False
step3_status = False
sleep(1)
if tester.checkRead((chmod_other)[0]) == True:
 step31_status  = True
#[TESTING OTHER WRITE]
sleep(1)
if tester.checkWrite((chmod_other)[1]) == True:
 step32_status = True
if step31_status and step32_status:
 step3_status = True
if step1_status and step2_status and step3_status:
 print "True"
else:
 print "False"
except:
 raise
 print "False"
```

Listing 1. Probe script for the evaluation of an access control mechanism protecting a file system

We developed a web application implementing an editor for the definition of CM Templates and CM Instances, based on the Model View Control paradigm. The code of TM, TA, and editor, together with some examples of CM Templates and CM Instances are available at [14].

VII. Running Example

We now describe a use case and running example of our engineering methodology and certification process (Section V-C). We assume a service provider that wants to *i)* develop a storage application, providing functionality to store and retrieve files from the cloud, and *ii)* certify it for security property *end-to-end confidentiality*.

First of all, the service provider selects a signed CM Template defined for property *end-to-end confidentiality* and a class of applications that includes storage applications. Let us assume that the selected CM Template specifies requirements on the application against an attacker with internal and external capabilities. In particular, it requires the application to implement the following security mechanisms for being certified:[4] *i)* an encrypted HTTPS channel between the application and the external world, *ii)* an authentication mechanism at service layer, *iii)* an access control mechanism at infrastructure layer protecting access to the file system. Mechanism *i)* protects the application against external attackers intercepting all or part of the communications; mechanism *ii)* guarantees authorized access only at service layer; mechanisms *iii)* protects the application against internal users trying to access personal information in the file system. Upon selecting CM Template, the provider designs and develops the application according to the above requirements (i.e., implementing the above mechanisms). The provider also contributes to the definition of a CM Instance that is compatible with the selected CM Template, and deploys the application according to it. In other words, it configures the probes and connect them to the hooks. Such CM Instance contains: *i)* one element **Collectors**, including three elements **Collector** with the test cases to be executed on each specific mechanism (an example of **Collectors** for mechanism *iii)* is presented in Figure 3 and Listing 1), *ii)* three elements **Aggregator** (one for each **Collector**) returning a *success* when all test cases are satisfied, *iii)* a **Life Cycle** automaton expressing the transition conditions in terms of the above **Aggregator**. More in detail, denoting A_1 the aggregator corresponding to the HTTPS collector, A_2 the aggregator corresponding to the authentication collector, A_3 the aggregator corresponding to the file system access control collector, the transition conditions in the **LifeCycle** in Figure 2 are as follows: *i)* $NI \rightarrow I : A_1 \wedge A_2 \wedge A_3$, *ii)* $I \rightarrow S : \neg(A_1 \wedge A_2 \wedge A_3)$ with $t<t_m$, *iii)* $S \rightarrow I : A_1 \wedge A_2 \wedge A_3$, *iv)* $I \rightarrow E : t_{exp}$, *v)* $S \rightarrow E : t_{exp}$, *vi)* $S \rightarrow R : \neg(A_1 \wedge A_2 \wedge A_3)$ with $t \geq t_m$, *vii)* $I \rightarrow R : \emptyset$, where t_{exp} is the expiration time of the certificate, t_m is the time after which the certificate becomes revoked according to condition *vi)*, and $\emptyset$ models a not-existing transition. We note that transition $I \rightarrow R$ is not implemented because all certificate revocations must pass through the suspended state in our example. Finally, the certification framework is composed of: *i)* a TM running the CM Instance and *ii)* one TA with three Collector Agents executing the corresponding **Collector** for each mechanism to be tested. As soon as the testing results are collected by the three Collector Agents, they are aggregated in a boolean value in A_1, A_2, and A_3 and returned to the LifeCycle Manager for updating certificate status. Our running example together

[4]We note that there may exist many CM Templates that can be used as a starting point to certify *end-to-end confidentiality* and require different sets of mechanisms.

with all relevant artifacts are available at [14], where interested readers can simulate a certification process. In this context, users can also simulate malfunctioning in one of the three mechanisms needed for property certification, causing a state transition in the certificate life cycle and a change in the certificate itself.

VIII. Related Work

Traditional certification approaches have been designed for static and monolithic software, and applied at deployment and installation time [9]. The success of service-based paradigm called for advanced certification schemes dealing with service-based system peculiarities (e.g., [2], [15], [16], [17]). Today, the cloud and its dynamic, flexible, multi-layer, and hybrid nature is fostering a new step in the evolution of certification [18], [19]. Spanoudakis et al. [12] proposed a concrete approach to security certification in the cloud. Anisetti et al. [13] proposed a chain of trust for certification in the cloud, which is at the basis of the framework in this paper. Some work [3], [11] focused on improving the trust of a cloud certification process by means of trusted computing platforms. Another relevant line of research focused on mechanisms providing the evidence at the basis of a certification process. While different types of evidence can be collected (e.g., formal proof and monitoring), in this paper we considered test-based evidence. A lot of work has been done on testing the cloud and its services (e.g., [20], [21], [22], [23]). All this work provided different approaches aiming to verify the behavior of a system at different layers of the cloud, supporting the dynamics and low transparency of the cloud, and increasing the quality of the testing activities. More in detail, Bai et al. [20] presented a survey of approaches and tools for cloud testing working at every layer of the cloud stack. Gao et al. [21] provided a tutorial focused on challenges and needs in SaaS layer testing. Pham et al. [22] presented CloudVal, a framework to validate the reliability of virtualization environments in a cloud computing infrastructure. Zech et al. [23] focused on the definition of negative requirements derived from risk analysis in security testing. This work proposed a model-driven risk-aware methodology for the security testing of cloud environments. Recently, much in line with the work in this paper, some engineering methodologies for designing security-aware system and software have been proposed based on requirements definition [24], [25], on threats modeling [26], and on the concepts of patterns [27], [28]. Haley et al. [24] presented a formal framework for security requirements analysis. It defines a context for the system as a set of constraints modeling security requirements, and provides formal and structured informal (satisfaction) arguments for checking feasibility of security requirements. Luckey et al. [25] tackled the problem of an efficient refinement of security requirements by extending the Activity-Based Quality Model (ABQM). Harjani et al. [28] focused on certification-oriented design and development, and presented an initial view on a certified-by-design engineering process. The main difference with the framework in this paper is that our methodology supports the development of ready-to-be-certified systems, while the work in [28] presents a solution where systems are designed a priori based on specific security models.

IX. Conclusions

We presented a certification framework that supports a security certification process for the cloud. Our framework supports pre-deployment and in-production evaluation of cloud systems at different layers of the cloud stack. It also provides a certification-aware cloud engineering methodology that drives cloud/service providers in the design and development of ready-to-be-certified systems. Our approach reduces the overhead, in terms of cost and time, which are introduced by traditional certification schemes.

Our future work will extend the framework to be deployable on different cloud infrastructures. Then, it will enrich our certification process with the management of hybrid evidence (e.g., a mix of test- and monitoring-based evidence) and incremental certification. Finally, it will provide an algorithm supporting the automatic check of consistency between CM Template and Instance.

Acknowledgments

This work was partly supported by the Italian MIUR project SecurityHorizons (c.n. 2010XSEMLC) and by the EU-funded project CUMULUS (contract n. FP7-318580).

References

[1] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the clouds: A berkeley view of cloud computing," in *Tech. Rep. UCB/EECS-2009-28*, EECS Department, U.C. Berkeley, February 2009.

[2] M. Anisetti, C. A. Ardagna, E. Damiani, and F. Saonara, "A test-based security certification scheme for web services," *ACM Transactions on the Web*, vol. 7, no. 2, pp. 1–41, May 2013.

[3] B. Bertholon, S. Varrette, and P. Bouvry, "Certicloud: A novel TPM-based approach to ensure cloud IaaS security," in *Proc. of the 4th IEEE International Conference on Cloud Computing (CLOUD 2014)*, Washington, DC, USA, July 2011.

[4] F. Doelitzscher, T. Ruebsamen, T. Karbe, M. Knahl, C. Reich, and N. Clarke, "Sun behind clouds - on automatic cloud security audits and a cloud audit policy language," *International Journal on Advances in Networks and Services*, vol. 6, no. 1–2, pp. 1–16, 2013.

[5] S. Pearson, "Toward accountability in the cloud," *IEEE Internet Computing*, vol. 15, no. 4, pp. 64–69, 2011.

[6] H. Rasheed, "Data and infrastructure security auditing in cloud computing environments," *International Journal of Information Management*, vol. 34, no. 2, pp. 364–368, June 2014.

[7] A. Schröter, T. Zimmermann, and A. Zeller, "Predicting component failures at design time," in *Proc. of the ACM/IEEE International Symposium on Empirical Software Engineering (ESEM 2006)*, Lund, Sweden, September 2006.

[8] L. Ye, H. Zhang, J. Shi, and X. Du, "Verifying cloud service level agreement," in *Proc. of the IEEE Global Communications Conference (GLOBECOM 2012)*, Anaheim, CA, USA, December 2012.

[9] E. Damiani, C. Ardagna, and N. E. Ioini, *Open source systems security certification*. New York, NY, USA: Springer, 2009.

[10] D. Herrmann, *Using the Common Criteria for IT security evaluation*. Auerbach Publications, 2002.

[11] A. Muñoz and A. Maña, "Bridging the gap between software certification and trusted computing for securing cloud computing," in *Proc. of the IEEE 9th World Congress on Services (SERVICES 2013)*, Santa Clara, CA, USA, June 2013.

[12] G. Spanoudakis, E. Damiani, and A. Maña, "Certifying services in cloud: The case for a hybrid, incremental and multi-layer approach," in *Proc. of the 14th IEEE International Symposium on High-Assurance Systems Engineering (HASE 2012)*, Omaha, NE, USA, October 2012.

[13] M. Anisetti, C. Ardagna, and E. Damiani, "A certification-based trust model for autonomic cloud computing systems," in *Proc. of the IEEE Conference on Cloud Autonomic Computing (CAC 2014)*, London, UK, September 2014.

[14] M. Anisetti, C. Ardagna, E. Damiani, and F. Gaudenzi, *A Methodology and Framework for Certification-Aware Cloud Engineering - Code and Artifacts*, https://github.com/fgaudenzi/testManager/tree/master/CertificationFramework.

[15] M. Anisetti, C. Ardagna, and E. Damiani, "Security certification of composite services: A test-based approach," in *Proc. of the 20th IEEE International Conference on Web Services (ICWS 2013)*, San Francisco, CA, USA, June-July 2013.

[16] D. Kourtesis, E. Ramollari, D. Dranidis, and I. Paraskakis, "Increased reliability in SOA environments through registry-based conformance testing of web services," *Production Planning & Control*, vol. 21, no. 2, pp. 130–144, 2010.

[17] V. Lotz, S. P. Kaluvuri, F. D. Cerbo, and A. Sabetta, "Towards security certification schemas for the internet of services," in *Proc. of the 5th International Conference on New Technologies, Mobility and Security (NTMS 2012)*, Istanbul, Turkey, May 2012.

[18] B. Grobauer, T. Walloschek, and E. Stocker, "Understanding cloud computing vulnerabilities," *IEEE Security & Privacy*, vol. 9, no. 2, pp. 50–57, March-April 2011.

[19] A. Sunyaev and S. Schneider, "Cloud services certification," *Communications of the ACM*, vol. 56, no. 2, pp. 33–36, February 2013.

[20] X. Bai, M. Li, B. Chen, W.-T. Tsai, and J. Gao, "Cloud testing tools," in *Proc. of the 6th IEEE International Symposium on Service Oriented System Engineering (SOSE 2011)*, Irvine, CA, USA, December 2011.

[21] J. Gao, X. Bai, W.-T. Tsai, and T. Uehara, "Saas testing on clouds - issues, challenges and needs," in *Proc. of the 8th IEEE International Symposium on Service Oriented System Engineering (SOSE 2013)*, San Francisco, CA, USA, March 2013.

[22] C. Pham, D. Chen, Z. Kalbarczyk, and R. Iyer, "Cloudval: A framework for validation of virtualization environment in cloud infrastructure," in *Proc of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2011)*, Hong Kong, China, June 2011.

[23] P. Zech, "Risk-based security testing in cloud computing environments," in *Proc. of the 4th IEEE International Conference on Software Testing, Verification and Validation (ICST 2011)*, Berlin, Germany, March 2011.

[24] C. B. Haley, R. Laney, J. D. Moffett, and B. Nuseibeh, "Security requirements engineering: A framework for representation and analysis," *IEEE Transactions on Software Engineering*, vol. 34, no. 1, pp. 133–153, 2008.

[25] M. Luckey, A. Baumann, D. Méndez, and S. Wagner, "Reusing security requirements using an extended quality model," in *Proc. of the 6th International Workshop on Software Engineering for Secure Systems (SESS 2010)*, Cape Town, South Africa, May 2010.

[26] A. Marback, H. Do, K. He, S. Kondamarri, and D. Xu, "Security test generation using threat trees," in *Proc. of the 4th International Workshop on the Automation of Software Test (AST 2009)*, Vancouver, Canada, May 2009.

[27] A. Maña, J. Ruiz, and M. Arjona, "Engineering secure and private systems using cossps," in *Proc. of the 21st International Conference on Pattern Languages of Programs (PLoP 2014)*, Monticello, IL, USA, September 2014.

[28] R. Harjani, M. Arjona, A. Muñoz, and A. Maña, "Towards an engineering process for certified multilayer cloud services," in *Proc. of the 8th Layered Assurance Workshop (LAW 2014)*, New Orleans, LA, USA, December 2013.