DOI 10.12694/scpe.v15i1.965
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 15, Number 1, pp. 49–62. http://www.scpe.org

DEVELOPING SECURE CLOUD APPLICATIONS

MASSIMILIANO RAK, MASSIMO FICCO* AND ERMANNO BATTISTA, VALENTINA CASOLA, NICOLA MAZZOCCA†

Abstract. Today the main limit to Cloud adoption is related to the perception of a security loss the users have. Indeed, the existing solutions to provide security are mainly focused on Cloud service provider prospective in order to securely integrate frameworks and Infrastructures as a Services in a Cloud datacenter. Customer could not monitor and evaluate the security mechanisms enforced by service provider. Service Level Agreements mainly focus on performance related terms and no guarantees are given for security mechanisms. Customers are interested in tools to verify and monitor the implemented security requirements. On the other hand, developers need tools to deploy Cloud applications offering measurable security grants to end users. In this paper, we propose an approach to implement security mechanisms as components in the application design process. We modeled security interactions according to the specific threat, the specific security requirements and user/application capabilities trying to improve security. It enables a Service Provider to offer security guarantees to customers. The approach has been designed to fit with different Cloud platforms, but to demonstrate its applicability, we will present a case study on the mOSAIC Platform.

Key words: Cloud security, security engineering, secure application design

62

1. Introduction. The most desirable Cloud feature is the self-service on demand approach. Cloud customers are able to access and use Cloud services without interacting with system administrators. Resources and services are charged on a pay-per-use base. The clear advantage for customers is the chance to have access to virtually infinite amount of resources, paying for them only when effectively used. As a side effect, end users do not own a real infrastructure: servers and data reside in the Cloud. In such scenario, the main limit to Cloud adoption is related to the perception of a loss of security the users have: is it possible to control where data reside? Are the customers granted about who can access their data? Is the provider reliable? Cloud customers can buy resources from different providers on the basis of the grants they offer, usually expressed in terms of Service Level Agreements (SLAs), i.e., contracts among the customer and the provider that specify what is granted [1, 2]. At state of art, even if clear innovative solutions exist, like the one proposed in the FP7 projects (SLA@SOI [3, 4], Contrail [5], mOSAIC [6]) or with standard languages like WS-Agreement ([7]), SLAs are commonly expressed in natural language and usually focus on performance related terms.

Existing solutions to provide security are mainly focused on Cloud service provider prospective in order to securely integrate frameworks and Infrastructures as a Services (IaaS) in Cloud datacenter. So, they focus on how to enable a Cloud service provider to offer security grants. Up to date very few results are available that consider end customers' perspectives. However, customers are interested in tools to verify and monitor the implemented security requirements. Nevertheless, application developers should be taken in consideration, too. They need tools to deploy Cloud application offering measurable security grants to end users [8, 9].

In this paper we will focus on these latter perspectives. In particular, we propose a methodology that enables the correct usage of a Platform as a Service (PaaS), in order to develop applications which clearly expose security grants to end users. The methodology has been designed to fit with different Cloud platforms, but to demonstrate its applicability we present a case study on the mOSAIC Platform [6].

The remainder of the paper is organized as follows: the next section aims at defining the problem of Cloud application security, outlining actors and roles. Section 3.1 describes the technology we focused on for application development and introduces a simple application we will use as a running example. Some related works are presented in Section 3.2. Section 4 describes the proposed methodology to take into account security requirements in Cloud application design. Section 5 shows how to apply the methodology. Finally, in Section 6 some conclusions and future work are presented.

2. Problem Description. The Cloud Computing paradigm implies to move from a local ownership of resources, services and infrastructures to a different approach in deploying, accessing and managing them. This leads to the multiplication of interactions among providers and Cloud users (actors), involved in the system and, consequently, to the perception of *loss of control* over data and offered services.

*Department of Industrial and Information Engineering, Second University of Naples, Aversa, Italia
†Department of Electrical Engineering and Information Technologies, University of Naples Federico II, Napoli, Italia

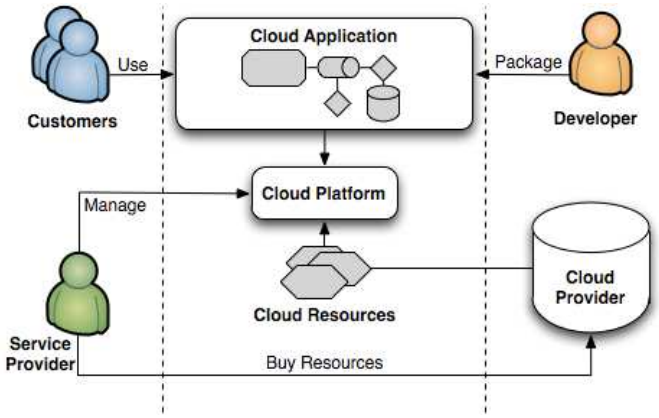

FIG. 2.1. *Actors and interactions*

Figure 2.1 briefly describes the interactions among Cloud actors. In particular, we identified three main actors: the *customer*, which uses Cloud applications and has specific security requirements; the *Cloud service provider* that deploys applications over Cloud resources, and provides application features to customers; *developer* that builds up the applications, and delivers them in terms of software packages offered to the service provider. Moreover, resources may be hosted over independent *Cloud providers*, which may sell them to the service provider under different conditions and with different guarantees. Service providers can then negotiate with their customers the adoption of such resources and services. In such context, a *Cloud application* runs over a *Platform* and it is offered in terms of a *Cloudware* (i.e., a software package that can be executed over Cloud resources in order to offer a platform able to run Cloud applications).

Customers have security requirements that should be granted by the service provider, but many security issues can arise in the described service supply chain. The service provider's task is to grant the overall security for the offered applications, but: (i) it uses resources bought from an independent Cloud provider, and (ii) it offers applications developed by third-party *developers*. Both Cloud providers and developers can enforce proper security mechanisms, but the interaction among them may not be secure at all.

We can summarize the problems to address, as follows: the customers have their security requirements, the developers have to implement them and service providers have to guarantee them in spite of the complex supply chain that can be effectively used. Therefore: (1) developers need to understand the requirements and enforce them into the application; (2) Service providers should be able to control the applications and their security features in order to guarantee them; (3) Customers need to negotiate and monitor their enforcement.

2.1. Contribution. Our first goal, presented in this paper, is to propose a methodology that enables developers to map user security requirements into security mechanisms to enforce; the enforcement is performed by adding security components to existing applications in the design phase and taking into account user requirements and possible application threats. Our long term goal, in future work, will be to enable the *automatic* negotiation, evaluation and enforcement of security features on developed applications.

In order to illustrate the approach in the practice, we focus on the mOSAIC Platform [10, 6], as the enabling technology for development of Cloud applications. The next section briefly introduces the mOSAIC Platform and shows how to develop an application using the mOSAIC APIs. We will use this example to outline the steps of the application development and how to improve security during the design stages.

3. Background and Related Works.

3.1. Developing applications with the mOSAIC Framework. The mOSAIC platform aims at providing a very simple way to develop Cloud applications [6, 10]. The target user for the mOSAIC solution is the application developer (*mOSAIC user*). In mOSAIC, a Cloud application is structured as a set of components running on Cloud resources (i.e., on resources leased by a Cloud provider) and able to communicate with each

other. Cloud applications may also be provided in the form of Software-as-a-Service, and can be accessed/used by users and by the mOSAIC developer (i.e., by *final users*).

Developing a mOSAIC application is very simple as it is built up as a collection of interconnected *mOSAIC components*. Furthermore a wide set of components is available. Components may be (*i*) core components, i.e., predefined tools offered by the mOSAIC platform for performing common tasks, (*ii*) COTS (commercial off-the-shelf) solutions embedded in a mOSAIC component, or (*iii*) *Cloudlets* developed using the mOSAIC APIs and running in a *Cloudlet Container*. mOSAIC Cloudlets are stateless, and developed following an event-driven asynchronous approach [10].

Among ready-to-use components it offers: (1) queuing systems used for component communications (*rabbitmq* and *zeroMQ*), (2) an *HTTP gateway*, which accepts HTTP requests and forwards them to application queues, and (3) NO-SQL storage systems (as Key-Value stores and columnar databases). mOSAIC components run on a dedicated virtual machine, named mOS (mOSAIC Operating System), which is based on a minimal Linux distribution. The mOS is enriched with a special mOSAIC component, the *Platform Manager*, which makes it possible to manage set of virtual machines hosting the mOS as a virtual cluster, on which the mOSAIC components are independently managed. It is possible to increase or to decrease the number of virtual machines dedicated to the mOSAIC application, which scales up and down automatically.

A Cloud application is described as a whole in a file named *Application Descriptor*, it lists all the required components and Cloud resources. A developer has both to develop new components, and to write the application descriptor to connect them together.

3.1.1. mOSAIC application example. To illustrate our work, in this paper, we refer to a simple Cloud application: an XML document manager (XDM) that analyzes the incoming XML files. For each received file, it counts the number of occurrences of XML tags inside the file and stores the results as a pair $<filename, tagcounts>$ (where *tagcounts* is a collection of $<tag, count>$ pairs) into the Key-Value store (KV).

The application consists of several components, which interact according to the sequence diagram shown in Fig. 3.1: an $HTTPgw$ Cloudlet manages the HTTP messages, and pushes the XML document (XML_Doc) into a queue; the XML *Analyzer* extracts and parses the queued XML documents, and stores the results (Res) in a KV store.

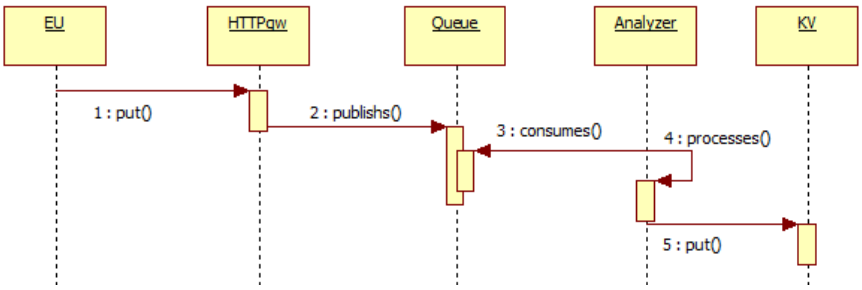

FIG. 3.1. *XDM component interactions*

Fig. 3.2 shows the architecture of such mOSAIC application, outlining the role of the different actors identified at the beginning of this section and different application components (Httpgw, Queue, Analyser and KV store).

Such application does not provide any security features and it is subjected to a number of attacks and threats. As an example, it may be subjected to XML Denial of Services attacks and it cannot guarantee mutual authentication among users and the application. The application developer well knows the components and how to connect them thanks to the application descriptor. If the developer knows - at this stage - the user requirements, is he able to enforce proper security mechanisms? Is there the possibility to design a methodology that help him in identifying and enforcing correct security solutions? At this aim, we propose an approach to implement security mechanisms as components to add in the design process. We can design

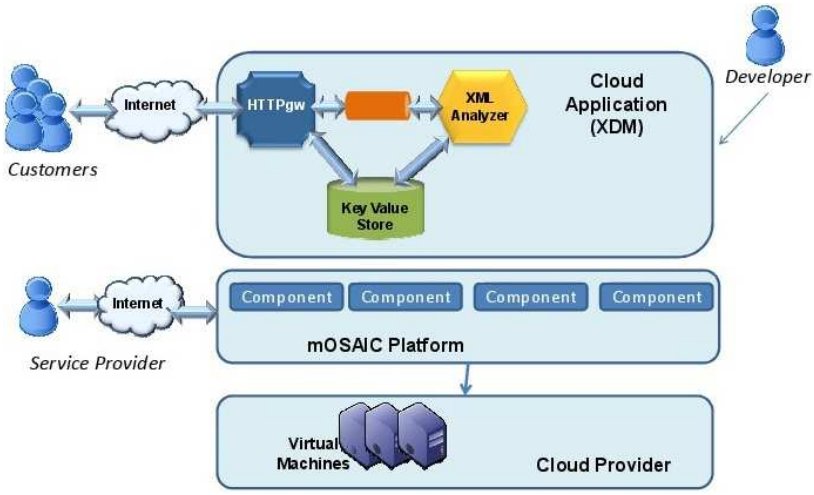

FIG. 3.2. *The XML Analyzer mOSAIC application*

component interactions according to the specific user security requirement, to its security capabilities and to known threats, trying to connect such security components with application/functional specific components.

3.2. Related Work. The protection of cloud resources is still an open problem and different approaches are available in the literature. In particular, several works propose specification languages to specify security attacks and requirements.

Abstract State Machine Language (AsmL) is a software specification language developed by Microsoft, defining abstract state machines through a fairly high-level language that can be compiled into executable code. Raihan et *al*. [11] propose a formal extension of AsmL, which describes how to model attacks, and present the design and implementation of a compiler that generates attack signatures from the attack specifications. Zulkernine et al. [12] present a method for integrating AsmL and Unified Modeling Language (UML) state charts, in order to extend finite state machine based software specification languages, with the Snort open source Intrusion Detection System (IDS). The attack scenarios are specified in AsmL or UML state charts, and then, automatically translated into Snort rules. The Snort's detection engine is modified so that it can understand the rules translated from AsmL or UML state chart scenarios.

Hussein et *al*. [13] present a framework for developing components with intrusion detection capabilities. This framework uses *UMLintr*, an UML profile for intrusion specifications [14]. The profile allows developers to specify intrusion scenarios using UML diagrams. Specifying intrusion scenarios, using the same language that is used for specifying software behavior, eliminates the need for separate languages for describing intrusions. UML diagrams used in UMLintr are called *misuse-case* and *misuse-state-machine* diagrams. Misuse-state-machine diagrams are used to specify the detection of intrusion scenarios. Each misuse-state-machine represents the states which its corresponding attack may go through.

STATL is a language to describe security violations as sequences of actions that an attacker performs to compromise a computer system [15]. The STATL language has been successfully used in describing both network-based and host-based attacks, and it has been tailored to very different environments. By abstracting away from the details of a particular attack, it is possible to detect previously unknown variations of an attack or attacks that exploit similar mechanisms. In particular, the STATL language provides constructs to represent an attack as a composition of states and transitions.

CORAS [16] is a method for conducting security risk analysis, it presents a customized language for threat and risk modeling, and provides detailed guidelines explaining how the language should be used to capture and model relevant information during the various stages of the security analysis.

The main difference between the proposed approach and the above described solutions, is that, our solution follows a *bottom-up* approach, focusing on the application architecture and implementation, having has target

user the developer. In particular, the previous works mainly propose formalisms to specify attack and intrusion models, as well as security rules. Our approach aims at proposing a methodology to describe mechanisms as components in the cloud-based application design process, which can be used to enforce security requirements specified in SLA.

4. Modeling Secure Cloud Interactions. Traditional techniques for protecting systems have almost clear boundaries and rely on a permanent architecture, which is *hardened* through the introduction of ad-hoc security mechanisms. This is not the case for a Cloud application: it runs over an enabling platform, which consumes multiple Cloud resources dynamically acquired. In order to design an application which offers clear grants to *Customers*, grant for which the Provider is responsible, we need a way to state such requirements and a way to evaluate how they are offered.

The approach is based on the following logical steps:

1. developers identify application use cases according to a common software development methodology;
2. developers identify the interactions (i.e., every kind of data exchange among actors in a Cloud system) among components of the Cloud application under study (next sections will detail what we mean with the *interaction* terms) and describe the use cases as a workflow of different *interactions*;
3. any interaction can be implemented and offered with different modalities, taking into account security mechanisms, threats, user requirements, user security attributes and capabilities.

Basing on these assumptions, we propose to develop specific security components that are available to developers, pre-associated to user requirements and pre-evaluated in terms of security guarantees. In this way, a developer can easily add security mechanisms to their applications and, furthermore, he will be able to control the security level that he is enforcing. The pre-evaluated components also enable the Service Provider to evaluate the goodness of a secure application and have a tool to guarantee this to end-users.

In order to demonstrate this, we will introduce the main concepts of *modeling interactions* and *interaction modalities* by using an Infrastructure as a Service example, then we will apply them on a real case study, based on a Platform as a Service (the mOSAIC Platform) to develop the security components.

4.1. The proposed approach. Several Cloud providers, such as Amazon EC2, Google Compute Engine, GoGrid, offer similar services and applications to customers, but they use different technologies and techniques to guarantee security grants to end users. Our approach is to offer to developers and customers a way to identify, independently of the technologies, the features they are interested in terms of components and additional security components to enforce security requirements expressed by end-users. We introduce a model that describes Cloud applications in terms of interactions and the ways in which such interactions take place (modalities) to implement specific Use Cases. Moreover, we took into account the roles' interactions that may occur in a Use Case.

4.1.1. Cloud actors. The actors classification introduced in the first section is specialized according to their capabilities to perform an action inside or outside the Cloud:

- EU: End Users (the customers) that interact with the Cloud by means of proper Web or software interfaces.
- CSP: Cloud Service Provider.
- S: Services that provide access to applications, resources and features.
- R: Resources located in the Cloud.

Each actor can assume different roles being both providers or requester of services. According to the specific interaction with the system, the roles we located are:

- INVOKER: It initiates the interaction and provides all the necessary information to process.
- TARGET: It is the interaction object.
- PROVIDER: It provides and manages the target. It is responsible for the accessing mechanisms both in terms of functionalities (such as interfaces) and security (e.g., access control lists, policy).

4.1.2. Actor roles. Even if actors can assume any role, generally, the EU acts as an INVOKER, the R acts as a TARGET, while CSP and S assume all the three roles. The general interaction schema among actors is

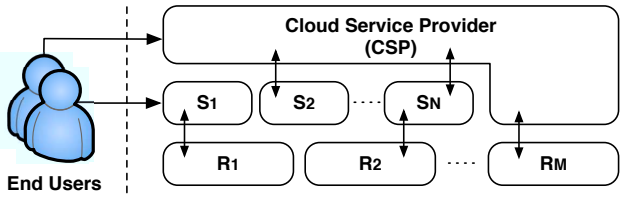

FIG. 4.1. *Interactions among actors*

reported in Fig. 4.1. For instance an EU can access a resource either using an ad-hoc service or through a CSP interface. Moreover, a CSP can offer a resource using direct access or by a service. The EU interacts with the Cloud resources and services in order to use an available *Use Case*. The term Use Case refers to the application features that an Invoker requests to the Provider. An Use Case implementation requires several interactions that can be implemented in different modalities. Therefore, we will model an Use Case as a workflow (sequence) of interactions. Each an *interaction* is characterized by the triple:

<INVOKER, PROVIDER, TARGET>.

Examples of valid interactions in which an EU has the Invoker role are reported in Tab. 4.1. Such examples help in clarifying what we mean with an *interaction Type*.

TABLE 4.1
Examples of valid Interactions Types

n.	Use Case description	Interaction Type
1	First EU's registration to CSP	<EU,CSP,CSP>
2	EU mounts a disk to an instance	<EU, S, R>
3	EU creates a virtual volume	<EU, S, S>
4	EU generates a new set of credentials for SSH	<EU,CSP,S>

The first example focuses on the actions performed by an EU when he registers himself to the Cloud provider. In this case, we outline that the *End user* acts as *Invoker*, while the CSP as *Provider*. Note that, the *Target* is the CSP itself: the interaction has the effect of changing the set of users of the CSP, enabling the EU to accept future requests for different services. From a security point of view, this operation is critical and the security mechanisms adopted to protect such interaction will be very relevant. We will take into account these mechanisms in the *Iteraction Modality*.

4.1.3. Iteraction modalities. CSPs can offer the same Use Case by several modalities. In Tab. 4.2 different examples of Interaction Modalities are associated to the same interaction. The Use Case "First EU's registration to CS" can be implemented with a Web Console interface and can require a user and password authentication mechanisms, or a more secure One Time Password mechanisms; these two are equivalent from the functional point of view, but provide a different security level being the second more secure.

Each modality can be associated to a different security mechanisms, which depends on threat related to the specific interaction modality, on user requirements, on his attributes and capabilities. Finally, different modalities can provide a different security level that is pre-evaluated. The evaluation of the security level is performed by adopting an evaluation methodology as the ones presented in [17, 18, 19]. Therefore, we can characterize an *interaction modality* by <Threat, Security Requirements, Security Mechanisms, Invoker's Attributes>, where:

- threats are specific attacks or systems vulnerabilities that we want to mitigate;

TABLE 4.2
Examples of valid Interactions Modalities and Security Level

Interaction Type	Interaction Modalities	Security Level
<EU,CSP,CSP>	Web Console with usr/pwd	2
-	Web Console with OTP	3
<EU,S,S>	Web Console with usr/pwd	1
-	Web Console with OTP	2
-	HMAC SHA1	3
-	HMAC SHA2-256	4

- security requirements are user desiderata that usually are requested in the Service Level Agreement with the provider;
- security mechanisms can be enforced with both additional components or with a specific implementation of an already existing component; for example, to improve security in the communication channel, one can use components that already have cryptoghaphic primitives to encrypt the channel or, alternatively, one can add external components to provide this feature;
- invoker attributes are needed to specify invokers security capabilities, if any.

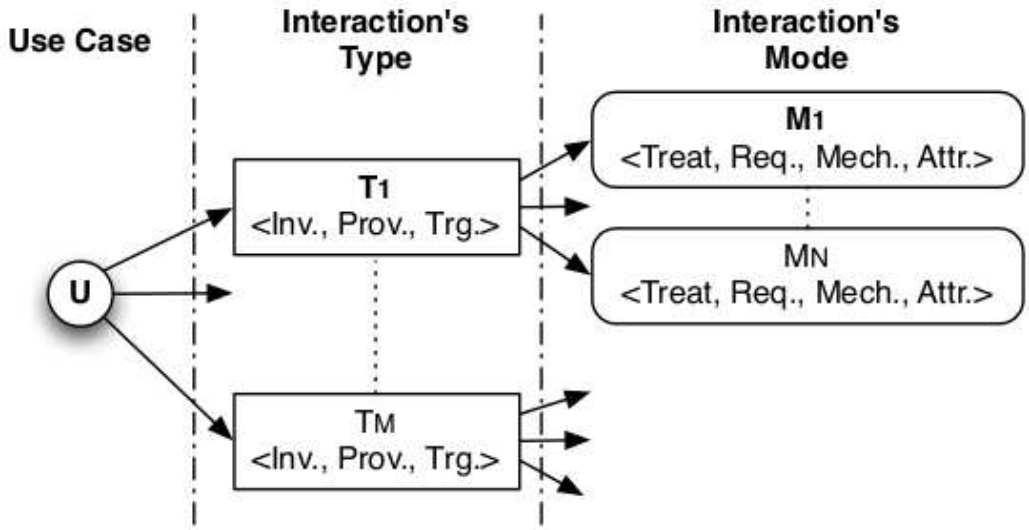

FIG. 4.2. *Modeling a Use Case with interactions*

From the developer point of view, this model helps to see a Cloud application as a composition of n different Use Cases implemented by a number of interactions T_i. In Fig. 4.2 the Use Case representation model is reported. Interactions T_i can be implemented in different modalities M_j that enforce different security mechanisms and are able to guarantee different pre-evaluated security levels. The developer has the possibility to choose one of the available application packages that best fit user requirements and provider guarantees.

Specifically, the approach will enable to add security related terms in the Service Level Agreements among Customers and Service Providers being able to easily evaluate and enforce proper security mechanisms.

4.2. Security mechanism description. Some security mechanism to be adopted can be defined by the security (detection) rules that it has to implement. A rule is specified by three basic elements: Condition, Attributes, and Reaction.

- *Condition* A rule is satisfied if its condition is true. A condition is made of a set of events. An event can be any significant occurrence detected by a probe, such as a particular string contained in a received packet, a predetermined number of messages received from the same IP or a high number of failed login attempts. It is also possible to set up a hierarchy of rules. In this case, a condition can be made of multiple rules too. If all the rules are true and the events are occurred, then the condition is satisfied. Events and rules can be combined with AND, OR, NOT logical operators.
- *Attributes* Every rule may have some attributes too. Designer has the possibility to specify the involved security probes or the monitored resource. For example, it is possible to specify the variable or the log file considered by the rule condition. It is also possible to include the eventual security protocol involved in the rule.

- *Reaction* If the specified condition occurred, the rule is true and the system may be under cyber attack. Thus, when all the events and the rules in a condition are true, an appropriate reaction is performed. The action to take is different basing on the events occurred. Actions depend also on events severity level. For example, an alert e-mail can be sent to the administrator if a low severity event occurs. Instead, in the case of a critical event extreme countermeasures must be taken.

Figure 4.3 shows an example of a rule representation. The stereotype 'Rule' characterizes the object and a numerical ID uniquely identifies the rule. The object is divided in three sections. The first one contains a short rule description and a list of all its optional attributes. In 'Condition' there are events and rules that make true the rule considered. The rule is valid if at least one of the conditions is verified. Each event must be associated to an ID and should be described in a separately table. 'Reaction' contains a short description of what the system has to do. It may contain a list of operations too.

5. A Case Study: Securing mOSAIC Applications. In order to validate the proposed approach we consider the application described in Section 3.1. We assume that the application offers two use cases:

- UC_P: a parsing service of the XML documents (XML_Doc), i.e., it performs the XML analysis (using a DOM approach), and computes the total number of nested XML tags;
- UC_Q: a query service for retrieving results (Res).

In this case study, we consider two kinds of Denial Of Service threats, the HTTP flooding and the Deeply-Nested XML:

- The HTTP flooding aims at creating a plenty of requests to the XDM application in order to exhaust resources and to inflict a denial of service to legitimate EUs.
- The Deeply-Nested XML exploits the XML message format by inserting a large number of nested XML tags in the message body. The goal is to force the XML parser (the Analyzer), to exhaust the computational resources of the host VM, by processing a large number of deeply-nested XML tags [20]. Moreover, this kind of attack is particularly harmful when using DOM based parser. A DOM based parser reads the complete message and builds an in-memory representation (called DOM tree), that is much larger than the message itself. Therefore, the application memory would be exhausted before the validator could even start the validation process.

For this case study, an advantage of using mOSAIC is the ability of automatically scale the application components (the Cloudlets and queues) when the host resources are overloaded (because the attack). But, as a side effect, the application consumes an increased amount of resources, on charge of the application owner (the Service Provider). The Provider needs proper security mechanisms in order to mitigate the considered attacks, as well as to guarantee the signed SLA. In next section we will illustrate how the developer can add such security mechanisms within the already deployed application, by simply adding the set of already available Cloudlets that act as mitigation means.

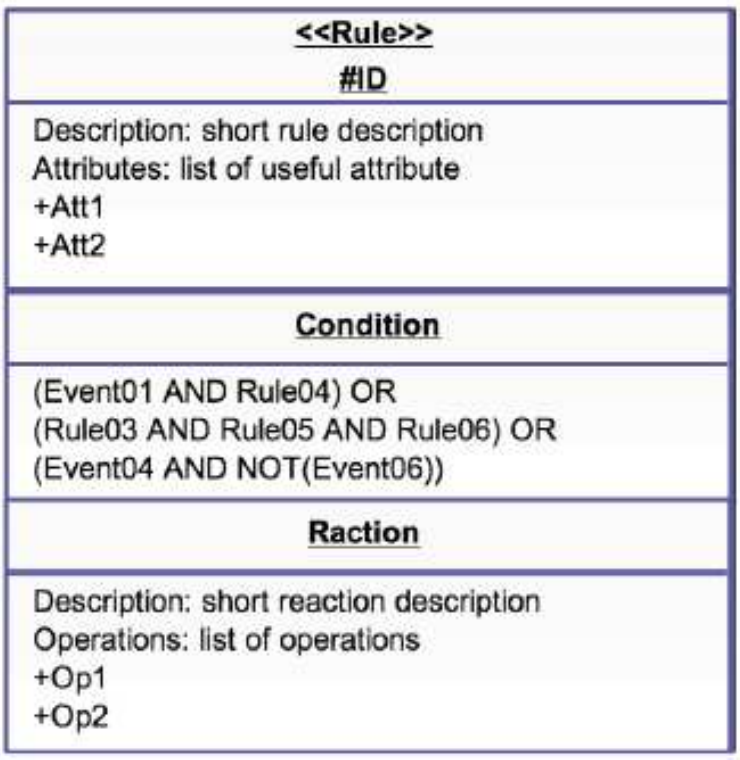

FIG. 4.3. *Example of rule representation*

5.1. Approach implementation. To understand how to apply the proposed methodology, we need to understand how to map user requirements with specific security mechanisms to add in the application. For this reason, to express user requirements, we will adopt the same tuple as described in the previous section, in this way it will be easier to identify additional components for the application developer. Fig. 5.1 expresses the security requirements and attacks to mitigate (they can be reported in a Service Level Agreement with the adoption of the same tuple [21]). In particular, they state that the User wants to protect the interaction T_P from both the attack types, and T_Q by the HTTP flooding. During the specification of requirements, the protection mechanisms and security attributes are not explicitly indicated and the values in the tuple are zero.

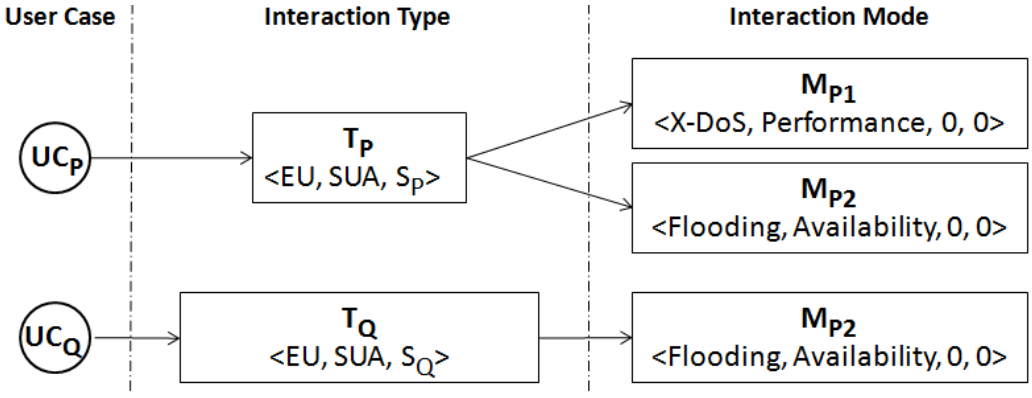

FIG. 5.1. *Security requirements expressed with the same formalism as interaction types*

According to these requirements, in the design phase, in order to protect the application from the considered

attacks, the developer has to identify/choose appropriate mitigation means (i.e., the modalities) for reducing or eliminating the potential intrusions. Specifically, for each interaction between the application components (in terms of INVOKER and PROVIDER), the developer has to identify the vulnerabilities that can be exploited by the specified attacks and, for each attack, he has to define/choose the mitigation means to be implemented, in order to mitigate or eliminate attacks exploiting an interaction vulnerability.

Therefore, the overall approach consists in mapping the high level Interaction Modes M_{p1} and M_{p2}, which express security requirements, with the low-level interactions among application components, as defined in the Service Descriptor of mOSAIC [22]. Moreover, for each interaction type, the developer has to identify vulnerabilities, possible mitigation means, needed user attributes to eventually enforce one mechanism (Can the user manage security tokens? Can the user provide a secret password?).

Looking at the sequence diagram represented in Fig. 3.1, the Use Case UC_P is implemented with four interactions (Fig. 5.2): $\langle$EU, HTTPgw, put$\rangle$, $\langle$HTTPgw, Queue,
publish$\rangle$, $\langle$Analyzer, Queue, consume$\rangle$, $\langle$Analyzer, KV, put$\rangle$ and, for each of them, the possible Interaction Modes with associated vulnerability are reported. For the sake of simplicity, in figure we just reported the threat and the requirement without reporting the mechanisms and user attributes that we will illustrate later together with a description of the located threats.

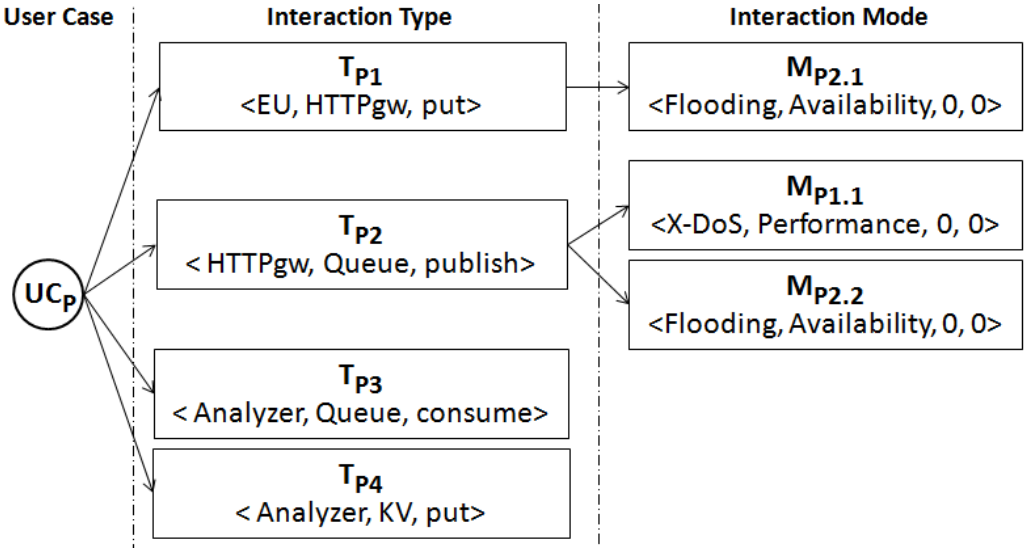

FIG. 5.2. *Use Case P: interaction types and Modes*

5.1.1. Interaction Mode implementation. As for the Interaction Mode $M_{p1.1}$ related to X-DOS threat, the schema validation is a good countermeasure against attacks based on XML messages that are not conform to the Web service description. These schema can be enforced by validating the incoming messages against specific restrictions [23]. Sophisticated solutions can be implemented by restricting the length of each XML element or limiting the list of XML elements contained within each message. However, although efficient XML validation engines (that can operate completely on the fly) have been proposed [24], in the current Web server frameworks, they are not used or not activated by default. This is mainly due to performance reasons, since schema validation is expensive regarding CPU load and memory consumption. This mechanism can be used to verify the presence of an attack, and to identify the malicious flows. If an attack is detected, a reaction can be triggered to filter malicious requests, or to block the attack sources [25].

The detection rule consists in monitoring the overall CPU consumption of the target application deployed over the distributed VMs. We can adopt a window-based state monitoring that triggers state alerts only when the normal state is continuously violated for a time window W [26]. W is essentially the tolerable time in abnormal state, which must be established by the service provider. Given the threshold T_R, the size W of the

monitoring window, and n monitored VMs with CPU values $CPU_i(t)$ at time t, an alert is triggered only when $\sum_{i=1}^{n} CPU_i(t-j) > T$, $\forall j \in [0, W]$ at any t. In order to validate the adopted technique, we assume that the alarm must be triggered when the overall CPU load on the involved VMs exceeds 80% for a time window $W = 10$ minutes. The considered rule is represented in Figure 5.3.

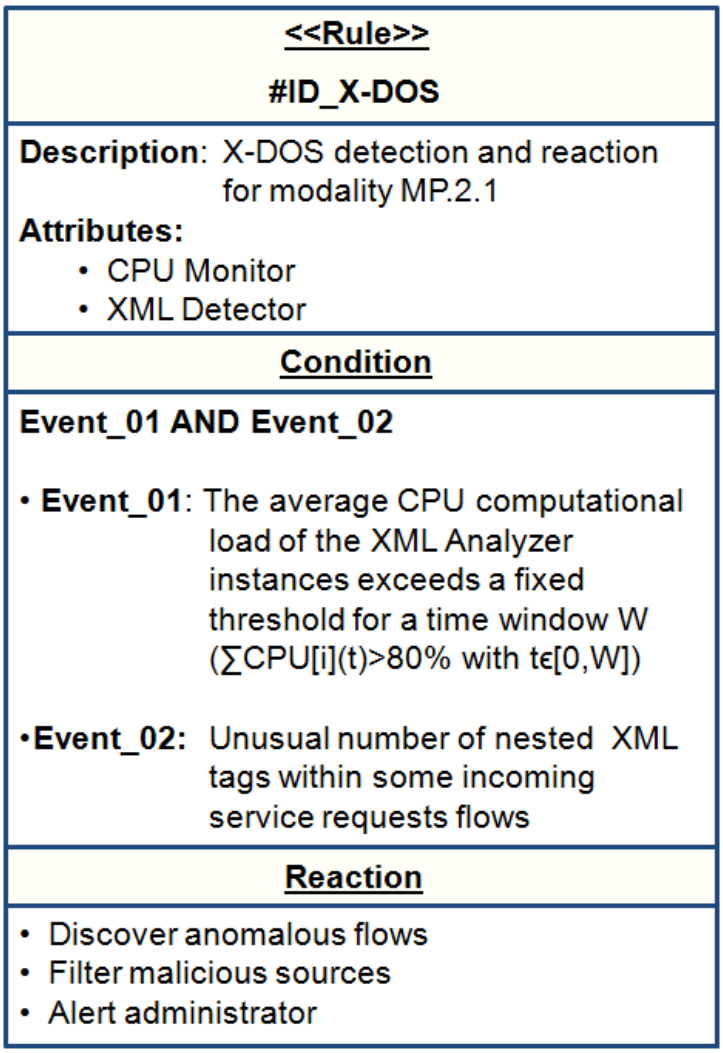

FIG. 5.3. *Detection rule description for modality* $M_{P1.1}$

As for the Interaction Modes $M_{p2.1}$ and $M_{p2.2}$ related to HTTP flooding threat, a different consideration can be done. In the process of visiting a Web site, the user sends requests, which have different properties in idle time, memory computational and computational load. On the contrary, the HTTP flooding sources send out HTTP requests with another pattern. In particular, several known worms' attack pattern use parallel threads to send GET requests in order to exhaust the server's resource. They might use different numbers of threads or different delays, but it's hard to simulate human user's request patterns. Therefore, the fact that the human user and attacker have different request patterns can be used in detecting the attackers, for example, by exploiting the idle time between each request from the same user [27].

5.1.2. Interaction Mode enforcement. In order to implement the identified solutions, we used specific mOSAIC components that are available and can be added by the Service Provider in a transparent way with respect to the application. For example, as for the X-DoS attack the following additional security components

can be used to implement the iteration modality:

- *Monitor*: By using the features offered by the mOSAIC Platform, the Monitor monitors the CPU consumption of the application. In particular, if the average computational load of the XML Analyzer instances exceeds a fixed threshold for a long time period, the Monitor enables the Detector.
- *Detector*: It consists of several sub components: the *Parser*, the *Validator*, and the *Engine* [8, 28]. The first two parse and validate the incoming messages, by using a schema validation approach (e.g., it is verified whether a hunge number of XML tags included in the incoming messages is below a fixed threshold). The Engine uses an anomaly based methods to identify the malicious sources (e.g., for each flow of messages, it counts the number of malformed XML documents received by the application). If a flow of requests is identified as malicious, it sends a message to the HTTPgw, reporting the malicious source of the messages (IP address).
- *HTTPgw*: It is enabled to operate as a filter. It discards the messages flows that are identify as malicious. This action is applied until the system performance returns to a normal state (e.g., the average CPU consumption of the observed Cloudlets is below the fixed threshold).

Fig. 5.4 shows how the architecture of the identified solution changes respect to the original one in Fig. 3.2, with all connected components to enforce the chosen interaction modalities.

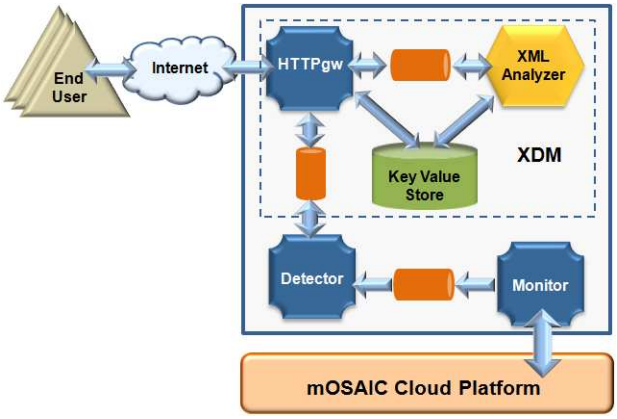

FIG. 5.4. *Application components to enforce chosen Interaction modes*

According to the proposed solution, the validation process of the incoming messages is enabled only when an attack symptom is detected. Moreover, in order to prevent resource exhaustion, the Detector works on an event-based approach, using a SAX parser. For each XML document parsed, an event is sent to the Validator, which directly operates on this event to validate the document. There is no need, neither for the Parser nor the Validator, to reconstruct the whole document in memory. In fact the Validator has constant memory consumption (only depending on the schema size) and linear run-time. The Detector can therefore easily process very large documents. If the Validator finds a schema violation inside a message, the Detector has read the document only up to that particular element, the remaining document is not read and therefore there is not impact on the Detector performance. If the Engine detects a potential attack, the message source is tagged as malicious, and a policy is set on the HTTPgw to discard messages from that source.

As for the HTTP flooding, two interaction modes can be enabled, but they cover different requirements (Fig. 5.2). The first mode can be used to prevent the attack. In particular, secure sockets layer (SSL) and transport layer security (TLS) encryption protocol can be used to secure Web communications via HTTPS. In particular, we can assume that a flooding attack is likely in progress, if the XDM application is overloaded with HTTP requests to be processed, and a huge number of incoming HTTP requests are discarded. In such a case, a detection method must be enabled in order to identify and filter malicious sources. The second mode can be applied for detecting and mitigating the attack. The associated detection rule is represented in Fig. 5.5

The implemented solution is similar to that presented in Fig. 5.4. By using some mOSAIC Platform features, a Monitor can monitor the state the XDM application's queue, as well as the number of incoming

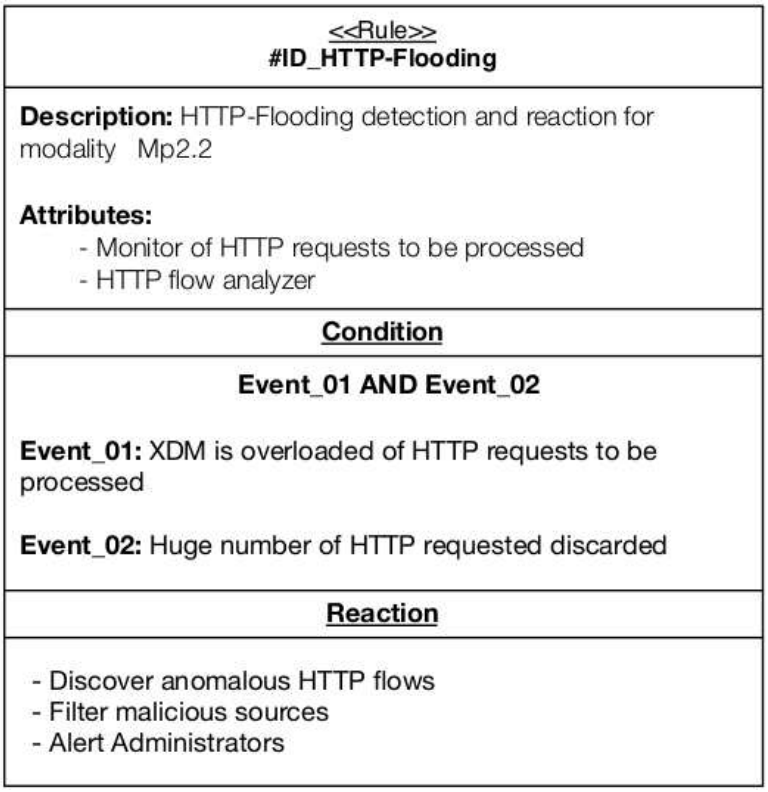

FIG. 5.5. *Detection rule description for modality* $M_{P1.2}$

HTTP requests. If the queue is full for a long time and many HTTP requests are discarded, a specific Detector is enabled. The Detector analyzes each incoming flow of requests. If a flow is identified as anomalous (it does not fit the model estimated during the training phase), the source of the flow is considered malicious and filtered by the HTTPgw. The detection method is based on the assumption that attackers and normal users have different properties in the request patterns. Therefore, a large volume of normal users' request sequences can be used to train a normal model, and then use the model to check incoming users. A Hidden Semi-Markov Model can be used to describe the process of an user visiting a website [27]. If the user's request sequence fits the model, the user will be considered as normal user. Otherwise, it will be considered as a potential attacker source and should be filtered.

6. Conclusions and Future Work. Security is one of the main limits in the adoption of Cloud infrastructures and applications. Users and providers do not have enough tools to evaluate and monitor the real enforcement of implemented security solutions. Indeed, security parameters are not included in Service Level Agreements as they cannot be negotiated and easily monitored. In this paper, we propose an approach to cope with this problem, by giving to cloud application developers the possibility to choose different security mechanisms that can be easily integrated in the application and have been pre-evaluated from the security point of view. We propose to model components' interactions from a security point of view in order to enable a developer to choose from different solutions which best fits the user requirements. The methodology we propose does not aim at proposing a new language to specify security attacks and requirements, indeed at the state of the art many solutions to this problem are based on this kind of approach. We aim at demonstrating the applicability of a simple methodology to implement security mechanisms in a transparent way with respect to the user applications, and to enable a Service Provider to easily offer security guarantees to customer. This is a first work in this direction, and we presented first results through a simple case study that introduces new components in an existing application to improve the overall security and meet user requirements. In future

work, we are going to implement different security packages and truly offer different secure interaction modalities. Our goal is to design a full and easy-to-use methodology to develop secure Cloud applications, such that the provided security is measurable and can be monitored by both end users and providers.

Acknowledgments. This work has been partially supported by the FP7-ICT-2013-10-610795 (SPECS).

REFERENCES

[1] M. Ficco, L. Tasquier, and B. Di Martino, "Interconnection of federated clouds," *Studies in Computational Intelligence*, vol. 511, pp. 243–248, 2014.
[2] C. Esposito, M. Ficco, F. Palmieri, and A. Castiglione, "Interconnecting federated clouds by using publish-subscribe service," *Cluster Computing*, vol. 6, no. 4, pp. 887–903, 2013.
[3] W. Theilmann, R. Yahyapour, and J. Butler, "Multi-level sla management for service-oriented infrastructures," vol. 5377, pp. 324–335, 2008. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-89897-9_28
[4] M. Comuzzi, C. Kotsokalis, C. Rathfelder, W. Theilmann, U. Winkler, and G. Zacco, "A framework for multi-level sla management," vol. 6275, pp. 187–196, 2010. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-16132-2_18
[5] CONTRAIL, "Contrail: Open computing infrastructres for elastic computing," http://contrail-project.eu/, 2010.
[6] mOSAIC Project, "mosaic: Open source api and platform for multiple clouds," http://www.mosaic-cloud.eu, 2010.
[7] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu, "Web services agreement specification (ws-agreement)," in *Global Grid Forum*. The Global Grid Forum (GGF), 2004.
[8] M. Ficco, "Security event correlation approach for cloud computing," *High Performance Computing and Networking*, vol. 7, no. 3, pp. 173–185, 2013.
[9] M. Ficco and L. Romano, "A generic intrusion detection and diagnoser system based on complex event processing," in *Proc. of the IEEE Int. Conf. on Data Compression, Communications and Processing (CCP'11)*. IEEE CPS, 2011, pp. 275–284.
[10] I. Ivanov, M. van Sinderen F. Leymann, B. S. SciTePress, and T. Publications, Eds., *Towards a cross platform Cloud API. Components for Cloud Federation*, 2011.
[11] M. Raihan and M. Zulkernine, "Asmlsec: An extension of abstract state machine language for attack scenario specification," *Proc. of the 2nd Int. Confe. on Availability, Reliability and Security (ARES'07)*, pp. 775–782, 2007.
[12] M. Zulkernine, M. Graves, and M. Khan, "Integrating software specification into intrusion detection," *Journal on Information Security*, vol. 6, pp. 345–357, 2007.
[13] M. Hussein and M. Zulkernine, "Intrusion detection aware component-based system: A specification-based framework," *Journal of System and Software*, vol. 80, no. 5, pp. 700–710, 2007.
[14] Hussein and M. Zulkernine, "Umlintr - a uml profile for specifying intrusions," *Proc. of the 13th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, pp. 279–286, 2006.
[15] S. Eckmann, G. Vigna, and R. Kemmerer, "Statl - an attack language for state-based intrusion detection," *Journal of Computer Security*, vol. 10, no. 1-2, pp. 71–104, 2002.
[16] F. den Braber, I. Hogganvik, M. Lund, K. Stølen, and F. Vraalsen, "Model-based security analysis in seven steps - a guided tour to the coras method," *BT Technology Journal*, vol. 25, no. 1, pp. 101–117, 2007.
[17] V. Casola, A. Mazzeo, N. Mazzocca, and V. Vittorini, "A security metric for public key infrastructures," *Journal of Computer Security*, vol. 15, no. 2, 2007.
[18] R. Preziosi, M. Rak, L. Troiano, and V. Casola, "A reference model for security level evaluation: Policy and fuzzy techniques," *Journal of Universal Computer Science*, January 2005.
[19] V. Casola, A. Mazzeo, N. Mazzocca, and M. Rak, "A sla evaluation methodology in service oriented architectures," *Advances in Information Security*, vol. 23, pp. 119–130, 2006.
[20] M. Ficco and M. Rak, "Intrusion tolerant approach for denial of service attacks to web services," *Proc. of the IEEE Int. Conf. on Data Compression, Communications and Processing (CCP'11)*, pp. 285–292, 2011.
[21] M. Ficco, M. Rak, and B. Di Martino, "An intrusion detection framework for supporting sla assessment in cloud computing," *Proc. of the 4th IEEE Int. Conf. on Computational Aspects of Social Networks*, pp. 244–249, 2012.
[22] M. Ficco, S. Venticinque, and B. Di Martino, "mosaic-based intrusion detection framework for cloud computing," *On the Move to Meaningful Internet Systems: OTM 2012*, vol. 7566, pp. 628–644, 2012.
[23] M. Jensen, N. Gruschka, R. Herkenhoner, and N. Luttenberger, "Soa and web services: new technologies, new standards - new attacks," *Proc. of the 5th European Conf. on web services*, pp. 35–44, 2007.
[24] N. Gruschka and N. Luttenberger, "Protecting web services from dos attacks by soap message validation," *IFIP Advances in Information and Communication Technology*, vol. 201, pp. 171–182, 2011.
[25] M. Ficco and M. Rak, "Intrusion tolerance of stealth dos attacks to web services," *IFIP Advances in Information and Communication Technology*, vol. 376 AICT, pp. 579–584, 2012.
[26] S. Meng, T. Wang, and L. Liu, "Monitoring continuous state violation in datacenters: Exploring the time dimension," in *IEEE 26th Int. Conf. on Data Engineering (ICDE)*. IEEE, 2010, pp. 968–979.
[27] W.-Z. Lu and S.-Z. Yu, "An http flooding detection method based on browser behavior," *Proc. of the Int. Conf. on Computational Intelligence and Security*, vol. 2, pp. 1151–1154, 2006.
[28] M. Ficco, L. Tasquier, and R. Aversa, "Intrusion detection in cloud computing," *Proc. of the 8th Int. Conf.on P2P, Parallel, Grid, Cloud and Internet Computing*, pp. 276–283, 2013.

Edited by: Teodor Florin Fortiş
Received: Mar 3, 2014
Accepted: Apr 16, 2014