2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks

Generating Cloud Monitors from Models to Secure Clouds

Irum Rauf
Åbo Akademi University,
Turku, Finland
Email: irum.rauf@abo.fi

Elena Troubitsyna
KTH – Royal Institute of Technology,
Stockholm, Sweden
Email: elenatro@kth.se

***Abstract*—Authorization is an important security concern in cloud computing environments. It aims at regulating an access of the users to system resources. A large number of resources associated with REST APIs typical in cloud makes an implementation of security requirements challenging and error-prone. To alleviate this problem, in this paper we propose an implementation of security cloud monitor. We rely on model-driven approach to represent the functional and security requirements. Models are then used to generate cloud monitors. The cloud monitors contain contracts used to automatically verify the implementation. We use Django web framework to implement cloud monitor and OpenStack to validate our implementation.**

I. Introduction

In many companies, private clouds are considered to be an important element of data center transformations. Private clouds are dedicated cloud environments created for the internal use by a single organization [20]. According to the Cloud Survey 2017 [3], private clouds are adopted by 72% of the cloud users, while the hybrid cloud adoption (both public and private) accounts for 67%. The companies, adopting private clouds, vary in size from 500 to more than 2000 employees. Therefore, designing and developing secure private cloud environments for such a large number of users constitutes a major engineering challenge.

Usually, cloud computing services offer REST APIs (REpresentational State Transfer Application Programming Interface) [39] to their consumers. REST APIs, e.g., AWS[1], Windows Azure [11], OpenStack [37], define software interfaces allowing for the use of their resources in various ways. The REST architectural style exposes each piece of information with a URI, which results in a large number of URIs that can access the system. Data breach and loss of critical data are among the top cloud security threats [25]. The large number of URIs further complicates the task of the security experts, who should ensure that each URI, providing access to their system, is safeguarded to avoid data breaches or privilege escalation attacks.

Since the source code of the Open Source clouds is often developed in a collaborative manner, it is a subject of frequent updates. The updates might introduce or remove a variety of features and hence, violate the security properties of the previous releases. It makes it rather unfeasible to manually check correctness of the APIs access control implementation and calls for enhanced monitoring mechanisms.

In this paper, we present a cloud monitoring framework that supports a semi-automated approach to monitoring a private cloud implementation with respect to its conformance to the functional requirements and API access control policy. Our work uses UML (Unified Modeling Language) [38] models with OCL (Object Constraint Language) [31] to specify the behavioral interface with security constraints for the cloud implementation.

The behavioral interface of the REST API provides an information regarding the methods that can be invoked on it and pre- and post-conditions of the methods. In the current practice, the pre- and post-conditions are usually given as the textual descriptions associated with the API methods. In our work, we rely on the *Design by Contract (DbC)* framework [28], which allows us to define security and functional requirements as verifiable contracts. Our methodology enables creating a (stateful) wrapper that emulates the usage scenarios and defines security-enriched behavioural contracts to monitor cloud. Moreover, the proposed approach also facilitates the requirements traceability by ensuring the propagation of the security specifications into the code. This also allows the security experts to observe the coverage of the security requirements during the testing phase.

The approach is implemented as a semi-automatic code generation tool in Django – a Python web framework [21] – and validated using OpenStack as a case study. OpenStack is an open source cloud computing framework providing IaaS (Infrastructure as a Service) [37]. The validation using OpenStack has shown promising results and motivates us to continue the tool development described in this paper.

The paper is organized as follows: section II motivates our work. Section III gives an overview of our cloud monitoring framework. In section IV, we present our design approach to modelling stateful REST services. The contract generation mechanism is described in section V. Section VI presents the tool architecture and our work with monitoring OpenStack. The related work and the conclusion are presented in sections VII and VIII correspondingly.

II. Private Clouds with REST principles

Cloud computing promises to improve agility, achieve scalability, and shorten time to market [27] of software devel-

2158-3927/18/$31.00 Â©2018 IEEE
DOI 10.1109/DSN.2018.00060

opment. This vision relies on the use of REST APIs, which enable extensibility and scalability of any cloud framework.

To facilitate extensibility, REST APIs expose their functionality as resources with unique Uniform Resource Identifiers (URIs). In complex systems, like the cloud frameworks, this results in a large number of URIs. Since the same set of HTTP methods (GET, PUT, POST, DELETE) can be invoked on them, with different authorization rules, safeguarding such a large number of URIs is a challenging task.

For example, let us consider a *volume* resource that is offered by the Cinder API of OpenStack [8]. Cinder is one of the services that is a part of the modular architecture of OpenStack. It provides storage resources (*volume*) to the end users, which can be consumed by the virtual servers [8]. A volume is a detachable block storage device that acts like a hard disk. Cinder API exposes the *volume* resource via ($/\{projectid\}/volumes/$). Any user of the project (e.g., *project administrator, service architect* or *business analyst*) with the right credentials can invoke the GET method on *volume* to learn its details. However, only the *project administrator* and *service architect* can update the existing volumes or add new volumes, and only the *project administrator* can delete a volume.

To offer scalability, REST advocates the stateless interaction between the components. This allows the REST services to cater to a large number of clients. Without storing the state between the requests, the server frees resources rather quickly that ensures system scalability. However, to construct the advanced scenarios using a stateless protocol, we should enforce a certain sequence of steps to be followed. Hence, we can treat such a behavior as a stateful one, where the response to a method invocation depends on the state of the resource. For example, a POST request from the authorized user on the *volumes* resource would create a new *volume* resource if the project has not exceeded its share of the allowed volumes, otherwise it will not be created. Similarly, a DELETE request on the *volume* resource by an authorized user would delete the volume if it is not attached to any instance, otherwise it would be ignored.

The security requirements combined with the functional requirements specifying the conditions under which a method can be invoked and its expected output result in a large volume of information. Moreover, such information should be defined for each resource, which becomes overwhelming for any cloud developer. In addition, if an API is developed in a distributed manner, i.e., by several developers working on implementing different parts of API, then the design errors and inconsistencies become inevitable. Therefore, we should propose an *automated* approach that would facilitate implementing correct security policies for each resource of the system and assure that the right users have an access to the right resources.

III. Cloud Monitoring Framework

Figure 1 presents the overall architecture of the Cloud Monitoring Framework. A cloud developer uses IaaS to develop a private cloud for her/his organization that would be used by different cloud users within the organization. In some cases, this private cloud may be implemented by a group of developers working collaboratively on different machines. The REST API provided by IaaS is used to develop the private cloud according to the specification document and required security policy.

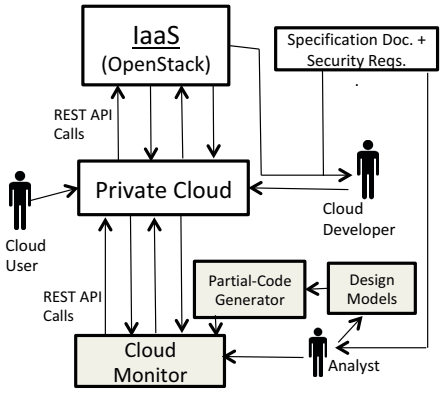

Fig. 1. Architecture of the Cloud Monitoring Framework

The cloud monitor is implemented on top of the private cloud. The main original components of our work are highlighted as grey boxes in Figure 1. The security analyst develops the required design models based on the specification document and security policies. These models define the behavioral interface for the private cloud and specify its functional and security requirements. In addition, our design models define all the information required to build the stateful scenarios using REST as the underlying stateless architecture [35].

In our approach, the construction of the design models serves several purposes: 1) the models specify the system from different viewpoints and hence, the security analysts can choose to specify in detail only those part of the system that they consider to be critical; 2) the models provide a graphical representation of the expected behavior of the system with the contracts, which can be communicated with a relative ease compared to the textual specifications; 3) the models serve as the specification document and facilitate reusability; 4) they are used to generate code skeletons with the integrated behavioral and security contracts; and finally, 5) we can use several existing model-based testing approaches to facilitate functional and security testing of private clouds.

We build on our partial code-generation tool [34] that is capable of generating the code skeletons from the design models. We extend this work by targeting the security requirements, i.e., the access rights over the resources, and propose an automated approach to representating the security requirements in the code. The generated code skeletons are then completed by the developer with the desired implementation of the methods.

A. Workflow

Our cloud monitor acts as a proxy interface on top of the private cloud implementation. It interprets the response codes

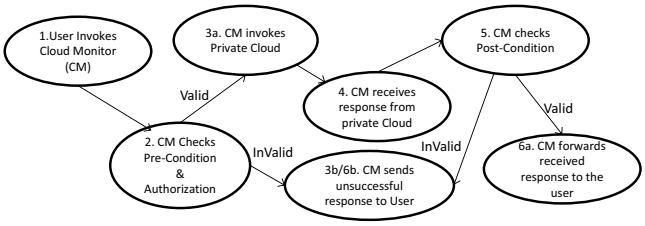

Fig. 2. Workflow in Cloud Monitor (CM)

of different resources to analyse how the request went. HTTP has a list of status codes [15]. The HTTP response code is a numeric value that informs the clients whether the request has been processed successfully. For example, the value 200 means that the request was successful, 404 means the resource was not found and 403 implies that it is forbidden to make this request on this resource.

Figure 2 presents the workflow within the Cloud Monitor (CM). The HTTP method request from CM user is forwarded to the private cloud if the pre-condition is satisfied. Similarly, upon receiving the response from the private cloud, the post-condition for the method request is verified. The request is successful if both the pre- and post-conditions evaluate to true, otherwise an invalid response specifying the faulty behavior is given to the CM user.

B. Users of Cloud Monitor

The cloud monitor can be used in a variety of ways. Namely, the users of cloud monitor can be:

1) a cloud developer, who is implementing the cloud for his/her organization and interested in validating his/her implementation during the development phase with respect to functional and security requirements.

2) a tester, who is interested in testing whether the implementation of the cloud satisfies its design specifications and security requirements.

3) a security expert, who wants to validate whether the cloud implementation has any security loopholes that may give access rights to the unauthorised users or prevent the authorised users to access the resources.

4) an automated testing script, which uses CM as a test oracle and invokes the cloud implementation through the cloud monitor to validate the authorization policy for all the resources. The invocation results can be logged for further fault localization.

In the next section, we present our design approach to specifying the behavioral interfaces for the RESTful architectures.

IV. Design Approach

Our approach focuses on modelling APIs that are REST compliant [35]. We use UML (Unified Modeling Language) [38], which is well accepted both in industry and academia, and has many associated industrial-strength automated tools. We briefly describe the construction of our resource and behavioral models using Cinder component of OpenStack introduced in section II as an example.

OpenStack services define the permitted requests based on the access rules introduces in their *policy.json* files, which follow Role Based Access Control (RBAC) paradigm [17]. Similarly to the other OpenStack services, Cinder uses Keystone service to validate the user's credentials and authorization requests [6].

The starting point of our approach is Cloud API, i.e., Cinder API [2]. It gives a textual description of which methods can be invoked on the cloud service, the request-response pairs, and the conditions for invoking those methods (if applicable).

We consider *volume* resource of Cinder Service (a part of *Volumes collection* resource) as our example because it is the central resource offered by Cinder. A *collection* resource does not have any attributes on its own. It merely contains a list of other resources. A *normal resource* has its own attributes and represents a piece of information. The users are assigned to *projects* in OpenStack. Each project in OpenStack has a corresponding quota, which limits the number of volumes that can be created in the project. The quota controls the resource consumption across the available hardware resources. Figure 3 shows the design models that elaborate on the structural and stateful behavior of a specific project.

A. Resource Model

We use the UML class diagram [38] with the additional design constraints to represent resources, their properties, and the relations between each other. We use the term *resource definition* to define a resource entity such that its instances are called resources. This is analogous to the relationship between a *class* and its *objects* in the object-oriented paradigm.

A collection *resource definition* is represented by a class with no attributes and a normal *resource definition* has one or more attributes. Each association has a name as well as minimum and maximum cardinalities showing the number of resources that can be part of the association.

Figure 3 (left) shows two collections *resource definitions – Projects* and *Volumes*. There are also four normal *resource definitions – project*, *volume*, *quota_sets* and *usergroup*. A collection *resource definition* has one outgoing transition with multiplicity of 0...* for the contained *resource definition* indicating that a collection resource can have none or many resources. The GET method on a collection resource returns a list of all the child resources that it contains.

To form URI addresses, every association should have a role name. The attributes of classes should be public since the representation of a resource should be available for the manipulation. Hence, they must have a type, because they represent a document containing an information about the resource, i.e., an XML document or a JSON serialized object.

B. Behavioral Model

The projects are created by the cloud administrator using Keystone and *users* or *usergroups* are assigned the roles in these projects. It defines the access rights of the cloud users in the project. A volume can be created, if the project has not exceeded its quota of the permitted volumes and a user

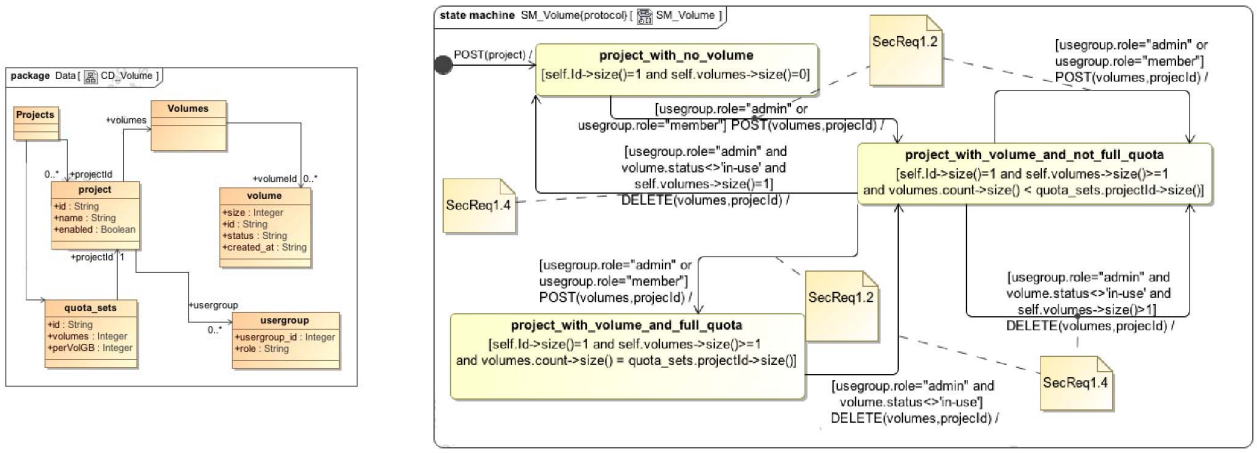

Fig. 3. (Left) Extract of Cinder Resource Model. (Right) Extract of Cinder Behavioral Model

is authorized to create a volume in the project. Similarly, a volume can be deleted, if the user of the service is authorized to do so, and the volume is not attached to any instance, i.e., its status is not *in-use*.

We represent the behavioral interface of the REST API by a UML state-machine [38]. Figure 3 (right) shows an excerpt from the behavioral interface of Cinder API for a project. It contains the information about the methods, which a user can invoke on the *volume* resource and the invocation conditions. In the example shown, at any given time a project can be only in one of three states. A project initially starts with no volumes attached to it. A volume is added to the project by the POST request. The request method can only be triggered, if the *user* belongs to the *usergroup admin* or *member*. As a result, the project transits to the *project_with_volume_and_not_full_quota* state. The subsequent POST requests on the project will keep it either in the same state or transfer to the *project_with_volume_and_full_quota* state, depending on the guard conditions. The DELETE method can only be invoked, if the status of the volume is not *in-use* and *user* belongs to the *usergroup admin*. The change of the project state depends on the guard conditions.

We define the invariant of a state using OCL [31] as a boolean expression over the addressable resources. In this way, the stateless nature of REST remains uncompromised because no hidden information about the state of the service gets stored between the method calls.

Figure 3 shows the state invariant for state *project_with_no_volume* written as the following OCL expression:
$project.id- > size() = 1 \quad and \quad project.volumes- > size() = 0$. It means that the project ID exists and has no volumes attached to it in this state. The condition $project.Volumes- > size() = 0$ implies that the response for invoking GET on the *Volumes* resource for the project was not 200, meaning either the resource does not exist or is not reachable to infer anything about its state. Similarly, the OCL expression $project.id- > size() = 1$ implies that the response for invoking GET on project resource was 200, i.e., the resource exists. The state invariant:
$project.id- > size() = 1 \quad and \quad project.volumes- > size() = 0$ specifies that initially a project exists but no volume is attached to it.

C. Authorization

Authorization in OpenStack, and other open source clouds (e.g., [11], [29]) is based on RBAC model [17]. In RBAC, the access rights of a user are defined by his/her role. We assume that the information about the roles and the corresponding access rights to the resources is well-defined and available for the cloud developer and security analyst.

In the current industrial practice, this information is usually given in a tabular format. We specify this information as the guards in the OCL format, which makes it amenable to an automated translation into the method contracts. In the behavioural model, each method should be labeled with a corresponding security requirement represented as a comment on a transition or state, as shown in Figure 3. When a state or transition with the requirement annotation is traversed, we get an indication which security requirement is met. This provides traceability of security requirements during the validation phase.

Table I outlines the security requirement of our example cloud implementation for *volume* resource. It specifies three types of roles implemented in the given project, namely *admin*, *member*, and *user*. Different user groups can belong to these roles.

V. Contract Generation

The interface of a cloud service advertises the operations that can be invoked on it. A cloud developer finds the cloud service API on the web and integrates it with the other services by invoking the advertised operations and providing it with the required parameters.

Resource	SecReq	Request	Role	UserGroup
Volume	1.1	GET	admin	proj_administrator
			member	service_architect
			user	business_analyst
	1.2	PUT	admin	proj_administrator
			member	service_architect
	1.3	POST	admin	proj_administrator
			member	service_architect
	1.4	DELETE	admin	proj_administrator

TABLE I
SECURITY REQUIREMENTS FOR CINDER API (EXCERPT)

These operations may imply a certain order of invocation or assume special conditions under which they can be invoked. Such conditions, i.e., pre- and post-conditions of a method, constitute contracts. This information together with the expected effect of an operation form a part of the behavioral interface of a service.

When the method *m* triggers a transition *t* in a state machine, the pre-condition for the method *m* should be true, i.e., the invariant of the source state of transition *t* and the guard on *t* evaluate to true.

For example, we are interested in generating a pre-condition to invoke the DELETE method on the *volume* resource, as shown in Figure 3 (right). DELETE on volume invokes three transitions in the behavioral model: one from the state *project_with_volume_and_full_quota* and two from the state *project_with_volume_and_not_full_quota*. We should note that while there are three different transitions triggered by DELETE(volume), the actual implementation should combine the behavior of these transitions into one method. Therefore, in order to generate the method contract, we need to combine the information stated in all the transitions triggered by a method into a pre-condition and post-condition for that method, as shown in the Listing 1.

Similarly, the post-condition states that if the pre-condition for invoking a method is true then its post-condition should also be true. We say that the post-condition of the method *m* is true if the conjunction of the state invariant of the target state of *t* and the effect on the transition *t* is true provided its pre-condition is true. Listing 1 shows the post-condition for DELETE(volume) method. The implication principle encompasses the stateful behavior since the same method can be fired from different states of the system and have different results. Thus, if the method is fired with a certain pre-condition then the corresponding post-condition for that method should also be established.

Since an execution of a method might change the state of a resource, to evaluate the pre-condition, we need to store the resource state before the method execution. To achieve this, we save the resource state before the method execution in the local variables of the monitor implementation. The values of these variables are later used to calculate the post-condition. We believe this is not computationally expensive because we do not need to save the copy of the whole resource(s) but only the values that constitute the guards and invariants that are evaluated. Usually, this only requires a few bits of storage per method. The pre- and post-conditions generated from behavioral model and security requirements table in OCL are shown in Listing 1.

```
PreCondition(DELETE(.../v3/{project_id}/volumes)):

[(project.id ->size()=1 and project.volumes->size()>=1 and
     project.volumes < quota_sets.volume and volume.status
     <> 'in-use' and user.id.groups= 'admin') or
(project.id ->size()=1 and project.volumes->size()>=1 and
     project.volumes < quota_sets.volume and
     project.volumes->size() >1 and volume.status <>
     'in-use' and user.id.groups= 'admin') or
or
(project.id ->size()=1 and project.volumes->size()>=1 and
     project.volumes = quota_sets.volume and volume.status
     <> 'in-use' and user.id.groups= 'admin')]

PostCondition(DELETE(.../v3/{project_id}/volumes)):

[((project.id ->size()=1 and project.volumes->size()>=1
     and project.volumes < quota_sets.volume and
     volume.status <> 'in-use' and user.id.groups= 'admin')
     ==> project.id ->size()=1 and
     project.volumes->size()>=0 )or
((project.id ->size()=1 and project.volumes->size()>=1 and
     project.volumes < quota_sets.volume and
     project.volumes->size() >1 and volume.status <>
     'in-use' and user.id.groups= 'admin') ==> project.id
     ->size()=1 and project.volumes->size()>=1 and
     project.volumes < quota_sets.volume and
     project.volumes->size() < pre(project.volumes->size()))
or
((project.id ->size()=1 and project.volumes->size()>=1 and
     project.volumes = quota_sets.volume and volume.status
     <> 'in-use' and user.id.groups= 'admin')==> project.id
     ->size()=1 and project.volumes->size()>=1 and
     project.volumes < quota_sets.volume and
     project.volumes->size() <
     pre(project.volumes->size()))]
```

Listing 1. Example of pre- and post-conditions

A detailed description of how to generate contracts from UML diagrams (without REST features and security requirements) can be found in [33].

VI. TOOL ARCHITECTURE OF CLOUD MONITOR

Section III described the main concepts and architecture of the cloud monitor (CM). In this section, we present details of CM implementation. CM is implemented as a proxy interface (wrapper). We implemented our monitoring mechanism in Django web framework [22]. Figure 4 shows the tool architecture. The dotted and solid arrows show the manual and automated steps correspondigly. The starting point of our approach is the specification document along with the description of the security policy. The system analyst should create the resource and behavioral models using some of the available UML tools. In this work, we have used MagicDraw UML [7]. We generate XML Metadata Interchange (XMI) of the behavioral model from this tool and save it into a file. The XMI files are given as the input to CM.

The tool gathers the necessary information from the input models and creates appropriate data structures. Django can be understood with its three basic files that support separation of concerns, as follows: the file *models.py* contains descriptions of the database tables, *views.py* contains the business logic and *urls.py* specifies which URIs map to which view. A detailed

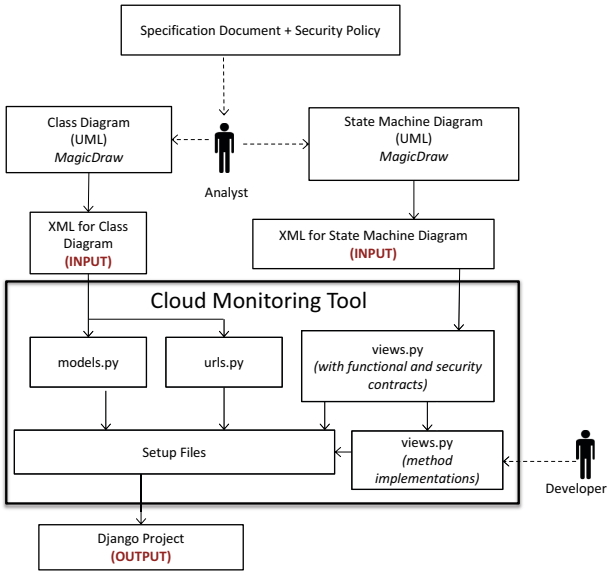

Fig. 4. Cloud Monitor

description of the Django web framework can be found in [18] and [21].

The current implementation continues our work on developing the wrapper [34]. It focuses on validation of the authorization policy and its implementation in the cloud environment.

The main steps in our implementation are as follows:

- We look for the resources in the class diagram to implement database tables in *models.py*. For each resource we create a table in the database, and analyze its associations to define their relationships with their keys. This creates a local copy of the resource structures as required by our monitor.
- *urls.py* contains the relative URLs of each resource and ways to access their respective views. This information is fully defined in the class diagram. By traversing the tags on the associations between the resources, we compose the paths of each resource. We always start from the corresponding collection, especially if we are referencing an item in the collection.
- The *views.py* file contains the main functionality of the system, i.e., the code that will run when accessing a resource through its URL according to the request (GET, PUT, POST or DELETE). These concepts are defined in the state machine diagram. The population of *views.py* is done in four steps: 1) add information regarding the permitted methods over the resources; 2) extract the functional contracts from the behavioral model as explained in section V and add them to the appropriate views; 3) add the authorization information from the guards into the appropriate views; 4) read security requirements from the comments on the transitions and add them as the corresponding variables in the code.
- Export to code all the information, i.e., create the file structure needed to run the system for the Django web framework.

In the final result, we obtain the necessary Django project files. As mentioned above, we need the intervention of the user in some parts of the *views.py* file. The user should fill in the skeleton code of the functions with the desired method implementation code. The use of the cloud monitor is very simple:

```
uml2django ProjectName DiagramsFileinXML
```

where ProjectName denotes the name of our project in Django and DiagramsFileinXML will contain the diagrams required in XML format.

The users can rely on cURL to invoke URIs if they want to use the cloud monitor. The cURL tool runs an HTTP client and supports most of the HTTP methods, authentication mechanisms, headers etc. [4]. For invoking a DELETE method on *volume* resource on a local server, the following command can be used in cURL:

$$curl - X\ DELETE - d\ id\ =\ 4\ http\ :\ //127.0.0.1\ :\ 8000/\ cmonitor/volumes/4$$

Alternatively, the users can also use REST clients available as plugins for different browsers or write test scripts.

A. Used Technologies

We have used the standard UML to model REST behavioral interface of private cloud. The UML standard provides many benefits to the system developers supported by a large user base and mature and sophisticated tools that constantly improve with time. In addition, these models can serve as a part of the specification document. We use MagicDraw to define our diagrams. Our work uses XML 2.1 and UML 2.0. Our compiler is implemented in Python programming language. It is written in Python 2.7 and requires lxml [5] module. In order to invoke methods on private cloud via code-skeletons, we use urllib2, a Python module that is used to fetch URLs [10]. Finally, to run the cloud monitor modeled, we have chosen Django web framework that uses Python too.

B. Limitations

Our approach is not fully automatic because we require the user intervention in filling in the missing lines of code in the generated code skeletons. This has been done to keep the models readable and avoid cluttering them with too many low-level details. Such an approach can make it harder to establish an equivalence between the specification models and the implementation code. The alternative approaches would be either to use the transformation languages or to enable fully automatic generation by deteriorating readability of the models. In our implementation, we chose to create a Python compiler with a greater capacity for compilation and processing of data structures enabling an analysis of the different parts of the code.

Another limitation, rather typical to the model-driven approaches, is the need to maintain the correspondence between the models and the code, i.e., whenever the changes are introduced, they should be done in both the models and the code. However, this is the price to be paid for enabling an automated model-based verification of functional and security requirements.

Ensuring scalability and managing complexity of the models is also a challenge associated with any modelling approach. Indeed, our models are enriched with fairly complex behavioral information defined in terms of state invariants. To reduce the complexity, we do not require the system analyst to model the entire functional and security behavior of private cloud in one diagram. Our approach can be used to represent and validate only those scenarios that are considered to be critical by the experts. Similarly, the number of methods that are selected for the implementation in the cloud monitor can be reduced by focusing only on those resources that are considered to be the main assets. We are planning to address these limitations in our future work by proposing a support for splitting the models into several parts via slicing or aspect-oriented approaches.

C. Implementation

For Cinder service, we exemplify implementation of DELETE method on *volume* resource. The Listing 2 shows the view for *volume* and $volume\_delete$.

```
def volume(request, volume_id):
    if not request.method in ["GET", "POST", "DELETE"]:
      return HttpResponseNotAllowed(["GET", "POST",
          "DELETE")
    if request.method == "DELETE":
        return volume_delete(request, volume_id)

def volume_delete(request, id):
    # add pre-condition - we skip detailed code for space
        limit
    if ((projectId and proj_volumes==1 and proj_volumes <
         proj_quota and vol_status != 'in-use' and
         usergroup=='admin') or ...):
        #redirect the DELETE request to volume resource in
             cloud implemetnation for project with Id:4
        url = 'http://130.232.85.9/v3/4/volumes/%s' % (id)
        opener = urllib2.build_opener(urllib2.HTTPHandler)
        req = urllib2.Request(url)
        req = RequestWithMethod(url, method='DELETE')
        return opener.open(request)
        response = urllib2.urlopen(req)
        the_page = response.read()
#check PostCondition
    if ((pre_projectId and pre_proj_volumes==1 and
         pre_proj_volumes < pre_proj_quota and
         pre_vol_status != 'in-use' and
         pre_usergroup=='admin') or ... ):
        if response.code == 204 # resource deleted
                response = HttpResponse(the_page)
                return response
```

Listing 2. Excerpt of DELETE view in Cloud Monitor for DELETE on *volume* resource on Cinder API

We can get the relative URI for each resource directly from Figure 3 (left) which is then mapped to the respective views as shown in Listing 3.

```
urlpatterns = patterns('',
          (r'^cmonitor/volumes/(\d{1,3})$', volume) ,
          (r'^cmonitor/projects/$', projects),
          (r'^cmonitor/projects/(\d{1,3})/volume/$',
               project_volume),)
```

Listing 3. URIs and views mapping for Cloud Monitor

Since *models.py* simply serves a local copy of the cloud resources being monitored, for brevity, we omit showing it.

D. Monitoring OpenStack

We installed OpenStack on a virtual machine (VM) with an ISO file on MacBook Pro with an i7 processor and 16 GB RAM. We used Oracle VM virtual box and Ubuntu 16.04 LTS ISO file. We deployed Newton version of OpenStack as two node architecture, i.e., controller and compute [9]. All the other needed services were configured on these VM nodes. The controller node was configured with 8 GB RAM, 40 GB hard disk, and 2 VCPUs. The compute node also had 40 GB hard disk with 2 VCPUs but 4 GB RAM. We implemented our private cloud with three user groups and roles shown above. We set up the cloud infrastructure for the project named *myProject*. The cloud monitor was run from Mac's terminal using *curl* commands [4] that invoked private cloud implementation running in Oracle VirtualBox. During validation, we were able to kill all three mutants (errors) systematically introduced in the cloud implementation to detect wrong authorization on resources. Due to the space limit, we omit the detailed discussion of technical details of the validation procedure.

VII. Related Work

There are three strands of research related to our work: model-driven security engineering, generation of executable code with contracts from models and security of private clouds. Research on using models to develop and analyze secure systems has been an active area of research for more than a decade. Nguyen et al. [30] provide a comprehensive overview of the research done in this area. The works specifically focusing on using UML to model authorization include such frameworks as SECTET [12], *UMLsec* [23], *MDSE@R* [13] and [14]. These works introduce the dedicated UML profiles to model security requirements. In our work, we have chosen to rely on the standard UML without employing a profiling mechanism. Such an approach allows us to use a variety of industrial-strength tools for UML, which promotes an industrial adoption of our method. Moreover, our model-driven approach is specifically tailored for the stateless nature of REST APIs, which is also novel comparing to the existing works.

A large body of work exists in generating executable conditions from various modelling notations. Gordon and Harel [19] generate executable code from structured requirements. Their work translates controlled natural language requirements into live sequence charts that are made fully executable. The work of Lohman [26] uses the visual contracts to specify software interfaces. In this work, the visual contracts and class diagrams define the dynamic and static aspects of a service correspondingly. The approach relies on graph transformations of the underlying modelling to define pre- and post-conditions of the operations. The assertions generated from the visual contracts are incorporated in Java Modeling Language and used to check software behavior at runtime. The general idea

behind our work is close to Lohman's approach. The main distinction of our approach is its focus on validating the advanced scenarios of REST APIs, which are widely used in cloud frameworks. Moreover, in addition to pre- and post-conditions, we also use the invariants as defined by design-by-contract paradigm [28].

Cloud security research has resulted in creating a variety of monitoring tools. Among them are the tools aiming at risk evaluation (e.g., [36]), enhanced encryption (e.g., [16]), continuous checking of security requirements according to the SLA document (e.g, [24]) as well as approaches that detect suspicious user behavior (e.g, [32]). Our monitoring tool addresses security from a different perspective. Namely, we aim at checking whether the implemented security policy complies to its specification. Our goal is to detect errors made by the developers during the implementation of private clouds. While there are many technical solutions to safeguard cloud APIs, in our work we aim at validating security at the application level. Moreover, to the best of our knowledge, there are no model-based design and monitoring approaches that address behavioral interfaces to properly represent the stateless nature of REST APIs.

VIII. Conclusions

In this paper, we have presented an approach and associated tool for monitoring security in cloud. We have relied on the model-driven approach to design APIs that exhibit REST interface features. The cloud monitors, generated from the models, enable an automated contract-based verification of correctness of functional and security requirements, which are implemented by a private cloud infrastructure. The proposed semi-automated approach aimed at helping the cloud developers and security experts to identify the security loopholes in the implementation by relying on modelling rather than manual code inspection or testing. It helps to spot the errors that might be exploited in data breaches or privilege escalation attacks. Since open source cloud frameworks usually undergo frequent changes, the automated nature of our approach allows the developers to relatively easily check whether functional and security requirements have been preserved in new releases.

References

[1] Amazon Web Services. https://aws.amazon.com/. Accessed: 30.11.2017.

[2] Block Storage API V3 . https://developer.openstack.org/api-ref/block-storage/v3/. retrieved: 126.2017.

[3] Cloud Computing Trends: 2017 State of the Cloud Survey. https://www.rightscale.com/blog/cloud-industry-insights/. Accessed: 30.11.2017.

[4] cURL. http://curl.haxx.se/. Accessed: 20.08.2013.

[5] Extensible markup language (xml). https://www.w3.org/XML/. Accessed: 27.03.2018.

[6] Keystone Security and Architecture Review. Online at https://www.openstack.org/summit/openstack-summit-atlanta-2014/session-videos/presentation/keystone-security-and-architecture-review. retrieved: 06.2017.

[7] Nomagic MagicDraw. http://www.nomagic.com/products/magicdraw/. Accessed: 27.03.2018.

[8] OpenStack Block Storage Cinder. https://wiki.openstack.org/wiki/Cinder. Accessed: 26.03.2018.

[9] OpenStack Newton - Installation Guide. https://docs.openstack.org/newton/install-guide-ubuntu/overview.html. Accessed: 20.11.2017.

[10] urllib2 - extensible library for opening URLs. *Python Documentation*. Accessed: 18.10.2012.

[11] Windows Azure. https://azure.microsoft.com. Accessed: 30.11.2017.

[12] MM Alam et al. Model driven security for web services (mds4ws). In *Multitopic Conference, 2004. Proceedings of INMIC 2004. 8th International*, pages 498–505. IEEE, 2004.

[13] Mohamed Almorsy et al. Adaptable, model-driven security engineering for saas cloud-based applications. *Automated Software Engineering*, 21(2):187–224, 2014.

[14] Christopher Bailey et al. Run-time generation, transformation, and verification of access control models for self-protection. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 135–144. ACM, 2014.

[15] Tim Berners-Lee et al. Hypertext transfer protocol–HTTP/1.0, 1996.

[16] Gaurav Bhatnagar and QMJ Wu. Chaos-based security solution for fingerprint data during communication and transmission. *IEEE Transactions on Instrumentation and Measurement*, 61(4):876–887, 2012.

[17] David Ferraiolo et al. Role-based access control (rbac): Features and motivations. In *Proceedings of 11th annual computer security application conference*, pages 241–48, 1995.

[18] Django Software Foundation. Django Documentation. *Online Documentation of Django 2.0*, 2017. https://docs.djangoproject.com/en/2.0/.

[19] Michal Gordon and David Harel. Generating executable scenarios from natural language. In *International Conference on Intelligent Text Processing and Computational Linguistics*. Springer, 2009.

[20] Robert L Grossman. The case for cloud computing. *IT professional*, 11(2):23–27, 2009.

[21] A. Holovaty and J. Kaplan-Moss. The Django Book. *Online version of The Django Book*, 2010. http://docs.djangoproject.com/en/1.2/.

[22] Adrian Holovaty and Jacob Kaplan-Moss. *The definitive guide to Django: Web development done right*. Apress, 2009.

[23] Jan Jürjens. Towards development of secure systems using umlsec. In *International Conference on Fundamental Approaches to Software Engineering*, pages 187–200. Springer, 2001.

[24] Nesrine Kaaniche et al. Security SLA based monitoring in clouds. In *Edge Computing (EDGE), 2017 IEEE International Conference on*, pages 90–97. IEEE, 2017.

[25] Ronald L Krutz and Russell Dean Vines. *Cloud security: A comprehensive guide to secure cloud computing*. Wiley Publishing, 2010.

[26] Marc Lohmann et al. A model-driven approach to discovery, testing and monitoring of web services. In *Test and Analysis of Web Services*, pages 173–204. Springer, 2007.

[27] Sean Marston et al. Cloud computingthe business perspective. *Decision support systems*, 51(1):176–189, 2011.

[28] Bertrand Meyer. Applying'design by contract'. *Computer*, 25(10), 1992.

[29] Dejan Milojičić et al. Opennebula: A cloud management tool. *IEEE Internet Computing*, 15(2):11–14, 2011.

[30] Phu H Nguyen et al. An extensive systematic review on the model-driven development of secure systems. *Information and Software Technology*, 68:62–81, 2015.

[31] OMG. *OCL, OMG Available Specification, Version 2.0*, 2006.

[32] Alina Oprea et al. Mosaic: A platform for monitoring and security analytics in public clouds. In *Cybersecurity Development (SecDev), IEEE*, pages 69–70. IEEE, 2016.

[33] Ivan Porres and Irum Rauf. From nondeterministic uml protocol statemachines to class contracts. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 107–116. IEEE, 2010.

[34] Irum Rauf and Ivan Porres. Beyond crud. In *REST: From Research to Practice*, pages 117–135. Springer, 2011.

[35] Irum Rauf and Ivan Porres. Designing Level3 behavioral REST web service interfaces. *ACM Applied Computing Review*, 11(3), 2011.

[36] Mehmet Sahinoglu. An input–output measurable design for security meter model to quantify and manage software security risk. *IEEE Transactions on Instrumentation and Measurement*, 57(6), 2008.

[37] Omar Sefraoui et al. Openstack: toward an open-source solution for cloud computing. *International Journal of Computer Applications*, 55(3), 2012.

[38] OMG UML. 2.4. 1 superstructure specification. Technical report, document formal/2011-08-06. Technical report, OMG, 2011.

[39] Jim Webber et al. *REST in Practice: Hypermedia and Systems Architecture*. O'Reilly Media, Inc., 2010.