State-Based Robustness Testing of IaaS Cloud Platforms

Domenico Cotroneo, Flavio Frattini, Roberto Pietrantuono, Stefano Russo
Universitá degli studi di Napoli Federico II
{cotroneo, flavio.frattini, roberto.pietrantuono, sterusso}@unina.it

ABSTRACT

An uncountable number of services are deployed over cloud platforms and provided to millions of consumers. As this paradigm spreads over, the quality of provided services becomes a primary concern. Testing helps in making software reliable, but it has been overlooked for cloud. In this paper, we present a method for the robustness testing of software platforms for IaaS cloud. The method stresses the importance of considering the state for these systems, which are characterized by phase-based interactions of many software components with multiple concurrent users. Applied to a real cloud platform, the method exposes failures hard to uncover with common robustness testing approaches.

1. INTRODUCTION

Testing is a main step to assess and improve software quality through fault detection. The research on cloud computing testing focused mainly on how to sell the *testing as a service* rather than on the more basic need of testing the service provisioning mechanism itself, so far. Many techniques have been devoted to how to optimize testing *in* the cloud, rather than testing *of* the cloud. Nevertheless, when a cloud user runs up against the bad quality of a service, the unsatisfied requests translate indirectly into a cost for providers. As a consequence, before deploying a service, providers may want to assess the robustness of the underlying cloud software system. They may want to improve their own platform to minimize failures and downtime, or simply compare different platforms through benchmarking and choose the best one for deploying their services[1].

Robustness testing is adopted to assess *"the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions"* [2]. Novel robustness testing approaches are required for cloud platforms, for which existing ones may be difficult to adopt. In many cases, they consider services and servers without state, and the system under test as a *stateless* black box. They aim at breaking the system by injecting unexpected and invalid inputs at its interface in a clean initial state [3]. While easing both testing and debugging, treating a system as stateless constitutes also the theoretical and practical limit of this type of robustness testing.

More recently, researchers are focusing their effort on including the impact of the state, especially in the field of operating systems dependability assessment [4, 5, 6]. When considering the system as a whole, rather than just the provided service with whom the user interacts, there is a state, indeed. Many requests come concurrently without the system being reset and some actions need to be executed in a specific succession. Thus, resubmitting the same inputs under different states is likely to expose new and more subtle failures. This is especially true for IaaS (Infrastructure as a Service) cloud platforms, where phase-based interactions with multiple concurrent users entail very strong dependencies of a request outcome on the state. Even a concurrency-aware testing method, encompassing *stressful environmental conditions*, may not suffice. To our knowledge, there is no approach to assess the robustness of the underlying platform while accounting for its state. Field experts prefer the approximation given by a pure input-based test, since accounting for the state is difficult and expensive, and lacks of a methodical strategy.

This paper presents a systematic approach for *state-based* robustness testing of IaaS cloud computing platforms. By abstracting *statefulness* from system evolution and user interactions, the approach guides the tester to select effective test cases to explore subtle failure regions by crossing the input and state domain smartly while controlling the test space explosion. The approach is applied to Apache VCL [7], an open source cloud computing platform. Results show the effectiveness of the method in finding robustness flaws with relatively few test cases, and in unveiling failures that manifest themselves only if considering the state explicitly.

[1]We consider a failure as the manifestation of the activation of a fault (or defect), as stated in [1]. The aim of testing is to expose/force the manifestation of failures, in order to discover defects causing them and/or evaluate system reliability.

2. RELATED WORK

We first review papers related to testing in the cloud computing field, then those related to state-based testing.

Cloud Testing. In the Cloud computing field, testing is commonly seen *as a service* to provide [8]. As for the testing *of* cloud platforms, very few papers can be found, each specific for the platform under study and not considering

cloud characteristics. OakLeaf Systems started a project for testing harness of Windows Azure Storage Services [9]. In [10], it is discussed how to evaluate scalability, authorization, billing, etc., for the Google's API Infrastructure. Similarly, in [11], Iyer et al. propose a framework for testing cloud features. It includes elasticity, scalability, security and performance testing, but it is not focused on reliability evaluation, which is our main aim. In [12], authors present a testing framework as a cloud application containing plugins for testing APIs of cloud platforms. Fault injection is used in [13] and [14] for assessing the reliability of cloud systems. Adopted approaches are very interesting and reach their goal of discovering faults in a reduced time with respect to common testing methods. However, the state of the cloud platform is not considered: as it is shown in Section 4, if not considering the state, some software defects remain hidden and the coverage incomplete.
The manifestation of exceptional events and reliability for Cloud platforms are strictly related to the interaction of the numerous software components (often provided by third parties) and to the correct termination of the various stages of the cloud service (e.g., authentication, resource request, use of the virtual resource), as discussed in detail in Section 3.3. These observations drove us to consider the state of the platform for assessing its robustness.

State-based testing. Ballista [3] was the first approach for evaluating and benchmarking the robustness of commercial operating systems with respect to the POSIX system call interface. OS robustness testing evolved in dependability benchmarks in the framework of the DBench European project [4]. DBench is a dependability benchmark proposed to assess OS robustness in terms of OS failures, reaction time (i.e., mean time to respond to a system call in presence of faults) and restart time (i.e., mean time to restart the OS after a test). Experimentations show that the workload should be representative of the expected usage profile (e.g., database or mail server) in order to assess robustness under common circumstances. This has been further studied in [15] and [16], where the authors assess the influence of the workload on the performance and memory consumption of several software systems by means of stress testing. Apart from OSs, the state is considered for other software systems, too. A common approach is to use state-based models. In [17], several models are used to represent the behavior of space software under the presence of both normal inputs and external faults in communication, processor, and memory. Authors show that a large number of errors are found when also considering the state. In [18], it is discussed an approach based on state machine for state base robustness testing. Tests are performed by considering the paths of the transitions to cover most of the system states and examine more transitions than stateless testing.

3. TEST CASE GENERATION APPROACH

We focus on testing of cloud platforms providing IaaS services. Even though the approach can be extended to other services, here we neglect functionalities and arguments that may be specific to a platform (e.g., software to be included, in the case of a Platform as a Service) or to a software (e.g., input parameters of an application provided as a service, in the case of a Software as a Service).
Our approach is based on four high-level steps to specify test cases: **1. Decomposition**, i.e. the identification of the set of independently testable functionalities, **2. Input Modeling**, **3. State Modeling**, **4. Constraints** application to eliminate potentially useless or redundant combinations.

3.1 Decomposition

The first step towards the testing of the platform is the identification of the provided functionalities. They can be identified by means of decomposition of the requirements specification. We model each functionality as: *[values] = F(par1, par2, . . . , parN)*, namely by a function *F* representing a generic *request*, an *output* representing the outcome as a set of one or more values, and a set of *input parameters*. A specific value of an input parameter is also called *argument*.

3.2 Input Modeling

Input modeling considers the input parameters of the functionalities to test as derived from requirements specification; for the arguments, we adopt a *data type* based approach [3], and consider three *classes* of values: *VALID*, *BOUNDARY VALID*, and *INVALID*, representing, respectively, the cases in which a value is as expected, is a boundary valid value (i.e. valid but uncommon), or is invalid with respect to what specified. The latter two classes of *exceptional values* are used to generate tests with at least one parameter having an exceptional value. Valid values are considered to prevent the masking of robustness failures: exceptional values correctly handled for one argument might mask non-robust handling of exceptional values for other arguments [3].

3.3 State Modeling

The *state* needs an appropriate modeling for IaaS cloud. We examined five open source systems to identify commonalities of a generic cloud computing platform: Apache VCL, CloudStack[2], Nimbus[3], OpenNebula[4], and OpenStack[5]. Developers built these platforms by integrating existing software components and libraries (e.g., hypervisors, distributed message brokers, storage systems) with new components responsible for accepting and orchestrating requests. We focus on functionalities related to IaaS cloud.
The architectural description of the cloud is derived from [19]. A platform is intended to serve **end user**s, requesting for resources (e.g., a Virtual Machine (VM)), and **administrator**s, who manage, modify, and repair the system.
A generic cloud system consists of these components:
Front End (FE): the interface of the system toward users. It allows customers to request for or access to a resource, or administrators to perform management operations.
Resource Manager (RM): responsible for the allocation of available resources to a request (i.e., the scheduling).
Provisioning Engine (PE): a PE instantiates VMs using the *hypervisor*, provides storage using the *Storage System*, or network by means of the *Network Manager*. It also frees resources once the service period expires.
Hypervisor (HV): provides virtualized abstraction of physical resources.
Storage System (SS): a SS provides storage capabilities to VMs and stores the images of the VMs.

[2]http://cloudstack.apache.org
[3]http://www.nimbusproject.org
[4]http://opennebula.org
[5]http://www.openstack.org

Network Manager (NM): it performs networking tasks, such as, the creation of VMs' virtual network interfaces, IP addresses assignment, etc.
Service Engine (SE): representing both the hardware (e.g., CPU, RAM, disks, network interfaces) and the software packages (e.g., monitoring tools) for providing a service.
Database (DB): a database is commonly used in cloud platforms to keep track of available resources, instantiated VMs, users details, etc.
Management Engine (ME): the ME component provides administrative functionalities. Examples are modification of delivered services and repair of failed components. It can interact with all the other components.
Formally, a cloud system concurrently provides services to a set of users $U = u_1, u_2, \ldots, u_l$, each one submitting (a subset of) m possible request types, $R = r_1, r_2, \ldots, r_m$. We name a complete interaction of each user with the platform as *session*. A user session (s_u) evolves through a set of n *phases* $P = p_1, p_2, \ldots, p_n$. We define the (sequential) phases which a cloud service may evolve through as:

i) **Authentication (AN)**: the first step for accessing the cloud services;
ii) **Submission (SB)**: when end user submits a resource request over a certain period;
iii) **Placement (PT)**: when the system looks for free resources to satisfy a request;
iv) **Provisioning (PV)**: the system creates the VM, allocates storage space, and sets up the request on the physical resource;
v) **Operational (OP)**: the user works with the resource until the service period expiration.
vi) **Deprovisioning (DP)**: the system de-provisions the deployed resources and frees the physical resources.

A **Management (MT)** phase encompassing all the administrative operations, such as upgrades, backup, failure/repair/ migration of components is also considered. It is not is sequence with the other phases.

The *state of the session* of a user $u \in U$ at time t ($s_u(t)$) is thus characterized by the phase ($p_{iu}(t)$, $i \in \{1, \ldots, n\}$ - the session of user u is in phase i at time t) and by the request type being served ($r_{ju}(t)$, $j \in \{1, \ldots, m\}$ - the session of user u is serving the request j at time t). The ***state of the platform*** at time t is given by the union of the states of the active user sessions: $\cup\{s_u(t)\} = \cup\{p_{iu}(t), r_{ju}(t)\}$. The number of active sessions implies the *concurrency degree* at a given time. The whole set of states is therefore given by: (i) the number of possible request types, multiplied by (ii) the maximum number of active sessions, and by (iii) the number of phases.

To abstract the *request types* users may submit, we separate them in groups depending on the weight of the operations in terms of OS resource usage and processing time. We consider the following factors and levels:

- **TypeADM**. For administrators, we distinguish between *inquiring requests* (i.e., visualization of usage information) with *LOW* weight, and *modification requests* (e.g., adding/removing host machines or images) with *HIGH* weight. This factor also includes a *NONE* type, indicating no operations by the administrator (i.e., concurrency is only among end users).
- **TypeUSER**. As for end users, we distinguish *operation requests* (e.g., connection/use of a VM) with *LOW* weight, from *reservation requests* of a resource, with *HIGH* weight. A *NONE* type applies also in this case, indicating only administrator requests in act.

We consider a specific factor indicating the level of *concurrent requests.*

- **Concurrency**. We distinguish *LOW* and *HIGH* level in terms of number of sessions currently active. The *LOW* level is expressed as a percentage of the maximum capacity (e.g., it is set to 30% of the *HIGH* level value).

The difference among the levels is to be preliminary confirmed for the platform being tested with short-running experiments.
Finally, we consider the *phase* in which the input case is submitted. In a normal flow, a request in a certain phase would be submitted only after the successful execution of all the previous phases; for the robustness testing, we are also interested in exceptional inputs in both normal and abnormal sequences (e.g., a resource deprovisioning request submitted after an unsuccessful provisioning).

- **Phase**. We consider these levels: (i) $PHASE_1$: input case submitted in the first phase; (ii) $PHASE_i$-$VALID$ and $PHASE_i$-$INVALID$, with i=2 . . . 6 (the 6 sequential phases): input case submitted in the i-th phase after *a)* a successful termination of all the phases from 1 to $i - 1$, and *b)* an unsuccessful termination of at least one of them. We end up with $5 \times 2 + 1 = 11$ levels.

3.4 Constraints

Since introducing state related factors, the number of tests one can perform may be excessively large. Constraints can discard combinations with higher likelihood to be redundant or with a reduced chance of discovering defects.
A *test case* is a vector of actual values selected for the levels (e.g., a specific VALID argument as input, an administrator operation with LOW weight, HIGH level of concurrency given by 10 users requesting for a resource). A *test frame* is a selection of the possible test cases that can be performed for a functionality. The aim is to select the tests with higher chances of discovering defects. The *testing* of a software consists in the repetition of test frames with all the possible combinations of input classes, and levels of state and concurrency related factors for each functionality.

Input Constraints. For the inputs, we can only operate on the three defined levels. We implicitly apply a constraint by discarding the combination with all *VALID* arguments. At deeper level, we can further reduce by eliminating combinations with the *VALID* class for one or more parameters that unlikely impact the discovery of defects.

State Constraints. Some constraints are related to *concurrency* factors. They focus on keeping only state classes believed to be more prone to trigger exceptional conditions.

- **Stress Reduction (Str)**: for some factors, we identified *HIGH* and *LOW* classes. Tester may choose only combinations with *HIGH* classes and discard *LOW* ones.

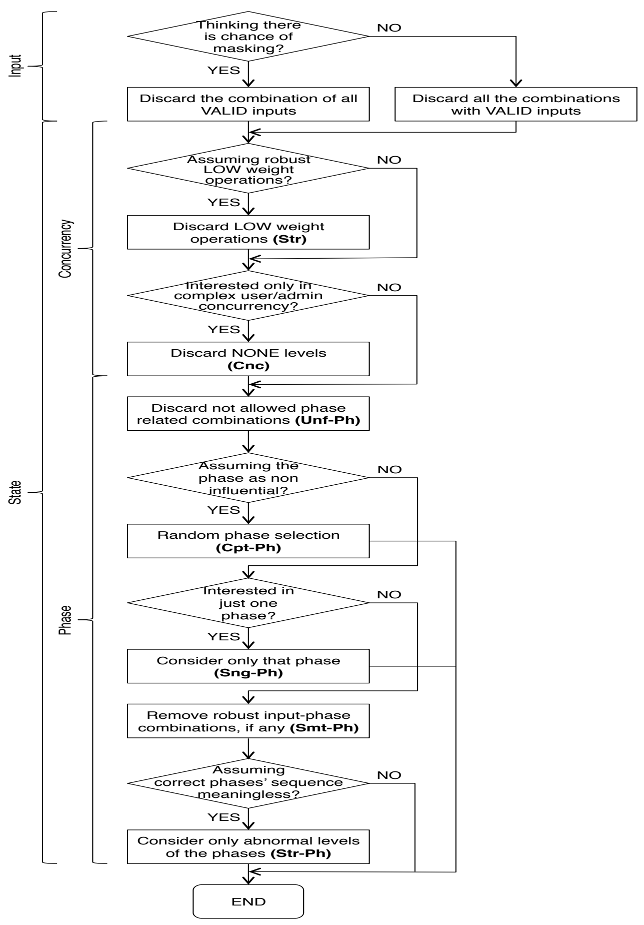
Input
Thinking there is chance of masking?
NO
YES
Discard the combination of all VALID inputs
Discard all the combinations with VALID inputs
Concurrency
Assuming robust LOW weight operations?
NO
YES
Discard LOW weight operations (Str)
Interested only in complex user/admin concurrency?
NO
YES
Discard NONE levels (Cnc)
State
Discard not allowed phase related combinations (Unf-Ph)
Assuming the phase as non influential?
NO
YES
Random phase selection (Cpt-Ph)
Phase
Interested in just one phase?
NO
YES
Consider only that phase (Sng-Ph)
Remove robust input-phase combinations, if any (Smt-Ph)
Assuming correct phases' sequence meaningless?
NO
YES
Consider only abnormal levels of the phases (Str-Ph)
END

Figure 1: Algorithm adopted for applying constraints.

- **Concurrency Reduction (Cnc)**: the tester could consider only complex cases of concurrency and discard those with *TypeUSER* and *TypeADM* set to *NONE*.

Other state constraints are related to *phases*.

- **Unfeasible Phase Reduction (Unf-Ph)**: when crossing a functionality with phases, there might be some cases not physically allowed by the platform; as an instance, the deletion of a VM may not be performed if it does not exist, since the system does not provide a means (e.g., a button) to perform the operation.
- **Semantic Phase Reduction (Smt-Ph)**: for some input-phase combinations the tester may be reasonably sure that the system is robust to them, and thus useless to test; for instance, a resource request before a successful authentication is a so simple case that previous testing stages have probably managed it.
- **Stress-based Phase Reduction (Str-Ph)**: cases for which the exceptional input is submitted under a normal sequence are discarded, while only abnormal levels of the phases are considered (e.g., provisioning of a VM for which a placement does not exist).
- **Single Phase Reduction (Sng-Ph)**: the tester may want to exercise a particular phase just once, and not on all the input combinations.
- **Complete Phase Reduction (Cpt-Ph)**: the extreme case is to submit the input case under a randomly chosen phase; this does not mean that the phase factor is not considered: with a large number of tests, it would be possible to cover a large portion of phases and expose the same failures of a testing where this constraint is not applied, even if with a reduced effectiveness.

In Figure 1, we report an exemplary algorithm for applying constraints. Clearly, the selection of the constraints to be applied depends on (i) the intuition and the experience of the tester, who feels how much the system is robust to certain combinations, and (ii) the *budget* for the testing, that is, how many test cases can be performed.

4. APACHE VCL CASE STUDY

We apply the testing approach described in Section 3 to Apache *Virtual Computing Lab* (VCL) [7], an open source cloud platform. It can be easily applied to other platforms by following the four steps detailed in the previous Section and here exemplified for the VCL case study.

For testing the VCL platform, we created a testbed in our laboratories. It has two machines, acting as controller node and hosting node, interconnected by a LAN, equipped with: Apache VCL 2.3, VMware server 2 as hypervisor, CentOS 6 as OS on the two machines and both Linux and Windows images for guest machines. Other machines (up to 6) act as clients of the cloud service.

Table 1 reports the main functionalities of the platform, to which we apply the method. The corresponding service phase is reported in the last column.

As for the concurrency factor, we found the high level corresponding to 6 concurrent operations, and fixed the low level to 2 (integer approximating the 30% of 6).

On the listed functionalities, we perform:

- **Input-based robustness testing** (T_{ib}): this is a pure robustness testing as performed in [3]; it only evaluates the system response to combinations of valid, boundary and invalid inputs. To reduce the number of tests, input constraints as defined in Section 3.4 are adopted.
- **Concurrency-based robustness testing** (T_{cb}): this is the robustness testing considering only the concurrency factor for the *stressful environmental conditions* defined in [2]; test cases are generated by also considering *TypeADM*, *TypeUSER* and *Degree of Concurrency* introduced in Section 3.3; stress and concurrency related constraints are applied.
- **State-based robustness testing** (T_{sb}): this is the proposed testing approach described in Section 3.

In this way, we distinguish the impact due to considering the degree of concurrency in addition to an input-based approach, and the further impact due to the phases and the other state factors defined in Section 3.3. Note that the test space of the state-based robustness testing (TS_{sb}) includes the test space of concurrency-based testing (TS_{cb}). In turn, the latter includes the test space of the input-based robustness testing (TS_{ib}). Formally, $TS_{ib} \subseteq TS_{cb} \subseteq TS_{sb}$.

Failures that these testing approaches can expose are categorized in:

- **Input-dependent failures** (F_{id}), which require only invalid inputs to be exposed, regardless the state.
- **Concurrency-dependent failures** (F_{cd}), which require a specific concurrency level to be exposed, since their effect can be noted only if a certain number of users is performing operations of a specific type.

Table 1: Main functionalities of Apache VCL.

Name	Inputs	Description	Phase
ResRequest	*characteristics, start, end*	request for a new virtual resource	SB
ResModify	*reservation_id, start, end*	modify existing reservation	SB
Place	*characteristics, start, end*	placement of a requested resource in a time period	PT
PlcModify	*schedule_id, start, end*	modify an existing placement	PT
PlcDelete	*schedule_id*	delete an existing placement	PT
CreateVM	*image_id, computer_id, hypervisor, cpu, ram, disk, network*	load and instantiate the requested resource	PV
DeleteVM	*computer_id, vm_id*	remove an instantiated VM	DP
AddHost	*ip_address, state, owner, platform_id, schedule_id, ram, num_procs, proc_speed, network, hostname, type, notes, computer_group, provisioning_id*	add information on a new host computer	MT
ModifyHost	*computer_id, ip_address, state, owner, platform_id, schedule_id, ram, num_procs, proc_speed, network, hostname, type, notes, computer_group, provisioning_id*	modify information on an existing host computer	MT

- **State-dependent failures** (F_{sd}), which manifest only within a specific state, while remaining latent in others.
- **Pure-state-dependent failures** (F_{psd}), whose manifestation is not due to concurrency, but to other state factors (which in the case of cloud systems are defined in Section 3.3).

Hence, $F_{cd} \subseteq F_{sd}$; $F_{sd} = F_{cd} \bigcup F_{psd}$, and $F_{cd} \bigcap F_{psd} = \emptyset$. Also, we notice that (i) input-dependent failures (F_{id}) can be found with whatever testing approach (T_{ib}, T_{cb}, T_{sb}); (ii) concurrency-dependent failures (F_{cd}) can be discovered only by means of concurrency-based or state-based testing (T_{cb}, T_{sb}); (iii) state-dependent failures (F_{sd}) can be found only if using state-based testing (T_{sb}).
As a metric for the failure-exposing ability of the testing, we introduce the *failures-tests ratio* (FTR). It is the total number of exposed failures over the number of run test cases.

4.1 Test Case Generation

To generate the test cases, we start from considering all the combinations of input and state levels, to which constraints are then applied. For illustrative purpose, we describe the application of the defined constraints to the *ResRequest* functionality. The same procedure is applied to all the tested functionalities. We point out that input constraints can be used for the three testing approaches; concurrency related constraints can be applied to concurrency- and state-based testing; constraints about phases can be used only for the state-based testing.
ResRequest has 3 input parameters: considering all their potential combinations of valid, invalid, and boundary values, there are $3^3 = 27$ *input* combinations. The state space includes the discussed (3 *TypeADM* $\times$ 3 *TypeUSER* $\times$ 2 *Concurrency* $\times$ 11 *Phase*) = 198 *state* combinations. The total number of test frames is $5,346$. For *input* constraints, we observe that the first input parameter, *characteristics*, can be selected only from a drop-down list with fixed values. Thus, neither *invalid* nor *boundary* inputs can be submitted. We discarde the combination of all valid inputs. Hence, combinations of inputs are 3^2-1=8. For *concurrency*, being interested in the case of end-user and administrator tasks concurrency, we discard *NONE* levels (*Concurrency Reduction* constraint). As for the *states*, we remove combinations considering the relation between submission and the other phases (*Semantic Phase Reduction*). In fact, while it makes sense considering the usage of a non-existing resource, we do not identify any relation between the submission phase and the following ones. Additionally, we do not consider the relation with the authentication phase, since the system does not physically allow a user to access other functionalities if not authenticated (*Unfeasible Input-Phase Reduction*). Thus, the final number of test frames in the state-based case is $(3^2 - 1) \times (2 \times 2 \times 2 \times 1) = 64$.

Table 2: Failures-Tests ratio for input-based, concurrency-based, and state-based testing.

Functionality	$FTR(T_{ib})[\%]$	$FTR(T_{cb})[\%]$	$FTR(T_{sb})[\%]$
ResRequest	37.50	37.50	37.50
ResModify	33.33	75.00	62.50
Place	37.50	37.50	68.75
PlcModify	37.50	50.00	75.00
PlcDelete	–	50.00	56.25
CreateVM	37.50	78.13	73.44
DeleteVM	–	50.00	56.25
AddHost	62.50	72.22	81.25
ModifyHost	37.50	58.33	55.00
Total	**40.30**	**56.36**	**62.22**

4.2 Analysis of the Results

Results from the testing of Apache VCL are summarized in Tables 2 and 3. Table 2 reports, for each functionality, the failures-tests ratio in the cases of input-based testing, of concurrency-based testing, and of state-based testing. Results highlight the importance of considering the state when testing a cloud computing service. It forces the tester to consider not only input, but also state configurations, in which the service is likely to fail. Overall performance presents an average failures-tests ratio of **62%** for the proposed testing approach. When considering only inputs, the average FTR is **40%**. Robustness testing accounting for also concurrency obtained an average FTR of **56%**. While the difference between 40% and 56% is due to considering the *concurrency* state factor, the difference between 62% and 56% is due to the additional failures exposed when the other state factors are taken into account, especially the *phase*.
Table 3 details the results of the *state-based* testing (T_{sb}). The number of state-dependent failures can be computed as $F_{sd} = F_{cd} + F_{psd}$. By way of example, consider the *CreateVM* functionality. 24 over 47 failures are exposed in presence of invalid inputs. These likely manifest also through a robustness testing as described in [3] (namely, a T_{ib}-like approach). Additional 23 failures are due to state. Of such

Table 3: For each functionality tested with the state-based approach: test space dimension, number of performed state-based tests, applied constraints, input-dependent failures, concurrency-dependent failures, and pure-state-dependent failures.

Functionality	**# tests (initial)**	**# tests (reduced)**	**Constraints**	F_{id}	F_{cd}	F_{psd}
ResRequest	$5,346$	64	Cnc, Unf-Ph, Smt-Ph	24	0	0
ResModify	$1,782$	32	Cnc, Unf-Ph, Smt-Ph	16	4	0
Place	$5,346$	16	Cnc, Unf-Ph, Str-Ph	0	7	4
PlcModify	$1,782$	16	Cnc, Unf-Ph, Str-Ph	0	8	4
PlcDelete	594	16	Cnc, Unf-Ph, Str-Ph	0	8	1
CreateVM	$433,026$	64	Cnc, Unf-Ph, Str-Ph, Sng-Ph	24	21	2
DeleteVM	$1,782$	48	Cnc, Unf-Ph, Smt-Ph	0	24	3
AddHost	$947,027,862$	64	Str, Unf-Ph	40	12	0
ModifyHost	$2,841,083,586$	40	Str, Unf-Ph	10	7	5
Total	$\mathbf{3,788,561,106}$	**360**	-	**114**	**91**	**19**

failures, 21 are due to the high level of concurrency and/or to the type of the concurrent request. However, 2 out of the 47 failures are exposed only when submitting the request in an abnormal sequence of the service life cycle; specifically, this happens when the placement phase does not execute successfully.

Overall, 224 failures are exposed through the 360 performed tests. Most of such failures ($\sim$ **51%**) depend only on the inputs; hence, they would occur also without forcing concurrency, stress, or specific phase sequences. 110 failures (**49.11%**) are state-dependent and represent very hard-to-reach cases. Exposing these failures with a pure input-based approach is extremely difficult, since the forced states represent rare conditions that cannot be reached by testing the system always in a clean state. Of this percentage, **40.63** points represent the failures that manifest only if adopting a testing approach considering stress and concurrency, apart from inputs (i.e., a T_{cb}-like approach). Such an approach should be used if respecting the standard definition of robustness testing [2]. The remaining **8.48** percentage points represent those failures that can be exposed only if considering the state as defined for the cloud service. These pure-state-dependent failures are the most subtle ones to expose, as they are not found if not explicitly forcing an invalid state, and are the actual advantage of our method. A conventional approach (like input- or concurrency-based) can achieve a certain level of robustness that is hardly improvable even employing a lot of further testing resources, as it is not conceived to expose pure state-dependent failures. The state-based method went beyond that level, as it is naturally oriented toward that kind of failures.

The conducted experiment also provides hints on the selection of input and state combinations. First, it confirms the importance of not discarding valid inputs, besides the single combination of *all* valid inputs. This choice avoids a potential failure masking. For instance, if both the *hostname* and the *owner* parameters of the *AddHost* function are invalid, the system checks the *owner* parameter first, which, if wrong, causes a failure, masking the behavior in presence of an invalid *hostname*. In fact, if only *hostname* is invalid a similar behavior is experienced. Similarly to inputs, also reducing the number of states could mask failures. By way of example, consider the creation of a virtual machine, which fails both in presence of heavy administrator operations and of heavy end user operations (*TypeADM=TypeUSER=HIGH*). Without considering light operations of both actors, it was impossible to figure out that concurrent LOW operations can cause a failure, hence hindering the correct identification of the cause. Finally, input and state combinations could mutually mask some failures too. For example, consider the *ModifyHost* function. If the *hostname* is invalid and the service correctly reached the provisioning phase ($PHASE_4$-*VALID*), a failure is experienced because of the incorrect input. However, with the system in the provisioning phase and a given combination of other state classes, a failure is experienced also if the input is valid.

5. CONCLUSION

The presented robustness testing approach started from the basic intuition that *given a software platform providing a cloud service, its response to the same input may vary due to (i) the number and type of operations performed concurrently, and (ii) the different use of interacting software components during various phases of the service life cycle.* This hypothesis was confirmed by the empirical results from our experience with Apache VCL. By comparing several testing approaches, we showed that:

i) When considering *only the inputs*, few failures from a specific, restricted set may manifest.

ii) When considering stressful environmental conditions as the *concurrency* in users' operations, additional failures are exposed; this should be the basic robustness testing approach to be considered for cloud platforms, accounting, at least, for the common large number of users.

iii) By forcing the *state*, as we specifically defined for the cloud, further failures may manifest. Such failures are the peculiarity of the proposed state-based approach, and can not be exposed otherwise, since underlying faults are activated only in specific situations.

An important next step is to automatize the execution of the selected test cases. Open specifications and APIs for cloud offerings can be adopted to this aim. An example is the Open Cloud Computing Interface (OCCI) [20], providing a protocol and APIs to perform both management tasks (e.g., monitoring, adding hosts) and common end user requests (e.g., create a new VM, access an existing VM). Many cloud platforms (e.g., CloudStack, OpenNebula, OpenStack) support this standard, as well as Amazon EC2 interfaces. Our future avenues of research will thus be focused on the automatic benchmarking of platforms for IaaS cloud in order to make them able to deliver reliable services and making the cloud even more trusted and usable.

6. REFERENCES

[1] A. Avizienis et al., “Basic concepts and taxonomy of dependable and secure computing,” *Dependable and Secure Computing, IEEE Trans. on*, vol. 1, no. 1, pp. 11–33, 2004.

[2] IEEE - Standards Association, “IEEE STANDARD 610.12-1990 - IEEE Standard Glossary of Software Engineering Terminology,” July 2014.

[3] P. Koopman and J. DeVale, “The Exception Handling Effectiveness of POSIX Operating Systems,” *IEEE Trans. on Software Engineering*, vol. 26, no. 9, pp. 837–848, Sep. 2000.

[4] A. Kalakech et al., “Benchmarking the dependability of Windows NT4, 2000 and XP,” in *Dependable Systems and Networks, 2004 International Conference on*, June 2004.

[5] A. Johansson et al., “On the impact of injection triggers for os robustness evaluation,” in *18th IEEE International Symposium on Software Reliability (ISSRE)*, Nov 2007.

[6] D. Cotroneo et al., “A case study on state-based robustness testing of an operating system for the avionic domain,” in *Computer Safety, Reliability, and Security*, ser. Lecture Notes in Computer Science, 2011.

[7] Apache, “Apache VCL,” March 2013, https://cwiki.apache.org/ VCL/.

[8] L. Riungu-Kalliosaari et al., “Testing in the cloud: Exploring the practice,” *IEEE Software*, vol. 29, no. 2, pp. 46–51, 2012.

[9] OakLeaf Systems, “Azure Storage Services Test Harness,” Feb 2015, http://oakleafblog.blogspot.it/2008/11/azure-storage-services-test-harness.html.

[10] A. Vallone, “Testing Google’s New API Infrastructure,” Feb 2015, http://googletesting.blogspot.it/2012/08/testing-googles-new-api-infrastructure.html.

[11] G.N. Iyer et al., “Pctf: An integrated, extensible cloud test framework for testing cloud platforms and applications,” in *International Conference on Quality Software*, July 2013.

[12] W. Jenkins et al., “Framework for testing cloud platforms and infrastructures,” in *International Conference on Cloud and Service Computing (CSC)*, dec. 2011, pp. 134–140.

[13] P. Joshi et al., “Prefail: A programmable tool for multiple-failure injection,” in *Proc. of the ACM Int.l Conference on Object Oriented Programming Systems Languages and Applications*, 2011.

[14] H.S. Gunawi et al., “Fate and destini: A framework for cloud recovery testing,” in *Proceedings of USENIX NSDI*, 2011.

[15] D. Cotroneo et al., “A measurement-based ageing analysis of the JVM,” *Software Testing Verification and Reliability*, 2011.

[16] A. Bovenzi et al., “Workload characterization for software aging analysis,” in *IEEE 22nd International Symposium on Software Reliability Engineering (ISSRE)*, December 2011.

[17] A.M. Ambrosio et al., “Designing fault injection experiments using state-based model to test a space software,” in *Latin-American Conference on Dependable Computing*, Berlin, Heidelberg, 2007.

[18] B. Lei et al., “State based robustness testing for components,” *Electronic Notes in Theoretical Computer Science*, vol. 260, pp. 173 – 188, 2010.

[19] F. Frattini et al., “Analysis of bugs in Apache Virtual Computing Lab,” in *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, 2013.

[20] Open Grid Forum, “Open Cloud Computing Interface,” July 2014.