2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

Evaluating the performance of continuous test-based cloud service certification

Philipp Stephanow and Christian Banse
Fraunhofer Institute for Applied and Integrated Security (AISEC), Munich, Germany
{firstname.lastname}@aisec.fraunhofer.de

***Abstract*—Continuous test-based cloud certification uses tests to automatically and repeatedly evaluate whether a cloud service satisfies customer requirements over time. However, inaccurate tests can decrease customers' trust in test results and can lead to providers disputing results of test-based certification techniques. In this paper, we propose an approach how to evaluate the performance of test-based cloud certification techniques. Our method allows to infer conclusions about the general performance of test-based techniques, compare alternative techniques, and compare alternative configurations of test-based techniques. We present experimental results on how we used our approach to evaluate and compare exemplary test-based techniques which support the certification of requirements related to security, reliability and availability.**

***Keywords*-cloud services; testing; certification**

I. Introduction

Continuous test-based cloud certification automatically and repeatedly tests if a customer's requirements are satisfied. Inaccurate tests undermine both provider's and customer's trust: While tests that incorrectly indicate satisfaction of requirements erode customer's trust, providers will dispute test results that incorrectly suggest customer's requirements are not fulfilled. Therefore, it is vital to evaluate how well continuously executed tests *perform*, that is, how close are produced test results to their true values?

Customers' requirements whose satisfaction should be checked can be derived from certificates, e.g. CSA STAR [1], or guidelines, e.g. NIST SP 800-53 [2] and ENISA IAF [3]. If the cloud service satisfies the requirements, then a report called *certificate* is produced, stating *compliance*.

Traditionally, producing a certificate is a discrete task whose results are valid for a period of time, usually ranging from one to three years. In regard to cloud services, the assumption of stability underlying traditional certification does not hold. A cloud service's attributes may change over time and such changes are hard to predict or detect by a customer [4]. Applying the concept of certification to cloud services therefore requires a different approach capable of continuously detecting ongoing changes and assessing their impact on customer requirements.

Requirements derived from, e.g., CSA's Cloud Control Matrix (CCM) [5] are generic, often times inherently ambiguous making automatic validation infeasible. Thus, supporting a continuous check whether cloud services comply with these high-level requirements requires an extraction of underlying properties which can be automatically evaluated, thereby bridging the *semantic gap*. Reasoning about these properties requires collecting and evaluating evidence [6], i.e. observable information of a cloud service, e.g. monitoring data, source code or documentation. *Test-based certification techniques* produce evidence by providing some input to the cloud service, usually during productive deployment, and evaluating the output, e.g. calling a cloud service's RESTful API and comparing responses with expected results.

Recent research proposes an approach to test-based cloud certification [7][8] whose implementation focuses on testing security properties of OpenStack. Other work focuses on designing specific test-based techniques [9][10]. Yet neither of them consider evaluating the performance of their test-based techniques when tests are executed continuously.

In this paper, we propose a method to evaluate the performance of test-based cloud service certification techniques when executed continuously. To that end, we introduce four universal test metrics which can be used with any test-based certification technique. Using these test metrics, we derive performance measures, allowing us to evaluate and compare alternative test-based techniques as well as alternative configurations of such techniques. Furthermore, based on the performance measures, our method permits us to draw conclusions about the general performance of a continuously triggered test-based technique. We present detailed experimental results on evaluating and comparing exemplary test-based techniques and their configurations.

The contributions of this paper are as follows:

- Four universal test metrics applicable to any continuously executed test-based certification technique,
- performance measures to evaluate and compare test-based certification techniques triggered continuously,
- a method to infer conclusions about the general performance of test-based certification techniques, and
- exemplary evaluations of test-based techniques supporting certification of requirements related to security configurations, resource availability, and reliability.

After presenting the main elements of our continuous test-based certification framework (Section II), we describe our method to evaluate the performance of continuously executed test-based certification techniques (Section III). Then we present experimental results of evaluating test-based techniques (Section IV). Finally, we discuss related work (Section V) and conclude this paper (Section VI).

978-1-5090-6611-7/17 $31.00 © 2017 IEEE
DOI 10.1109/CCGRID.2017.134

II. Continuous Test-based Cloud certification

This section outlines the main elements of our framework to support continuous test-based cloud certification: *Test suites* combining *test cases*, *workflows* modelling dependencies between test suites, *metrics* reasoning about test suites' results, and *preconditions* validating assumptions about the environment of the cloud service under test.

A. Test cases

Test cases form the primitive of any test, they implement any steps executed during the test, e.g. first establish an SSH connection to a virtual machine, then execute a command to download and install a package on the VM. A test case possesses initialization parameters, e.g. connecting to a VM via SSH may require username, hostname, and a path to a keyfile. Further, each test case possesses assert parameters specifying expected results, e.g. the returned values of the test case have to equal a particular string. Multiple test cases can be executed concurrently or successively.

Beyond simply passing or failing, the result of a test case run also includes start and finishing time of the run and can provide further information, e.g the maximum average response time of TCP segments measured to test latency of a remote host.

Note that our framework requires executions of test cases to be independent of each other, that is, whether a test case is executed or not does not depend on the results of other test cases. However, we note that concurrently executing multiple test cases on one service can naturally produce side-effects, i.e. test case results that affect each other.

B. Test suites

Test suites combine test cases, each suite containing at least one test case. A test suite passes if all contained test cases pass. Execution of a test suite can be triggered multiple times, possibly set to infinity, where the current iteration of a test suite has to be completed, i.e. all test cases bound to the test suite have to be completed, in order for the following iteration to start. The interval between consecutive iterations of a test suite can be fixed, e.g. 10 minutes after the previous test suite execution completed, or the interval can serve as a window from which the start of a test suite's execution is selected randomly.

Once a test suite run (tsr) completes, it returns failure or success, the run's start (tsr^s) and end time (tsr^e), as well as the results of all bound test cases. Hereafter, we use the term *test* and *test suite run* synonymously.

C. Workflow

A workflow models dependencies between iterations of a test suite and between iterations of different test suites. To that end, a workflow controls test suites' executions based on test suites' results. As a basic example, consider after having successfully completed a number of iteration, a test suite run fails. The workflow defines how to handle this failure, e.g. whether to continue running the test suite for the remaining iterations, to terminate the test or start another test suite.

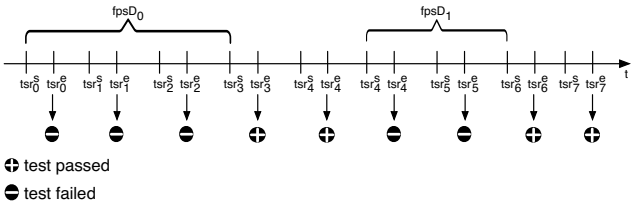

Figure 1: Continuously executed tests (tsr) with universal test metric $fpsD$

D. Test metrics

Automatically evaluating statements over cloud services properties, e.g. the availability of the cloud service is higher than 99.999% per day, requires one final construct: Metrics. A metric takes the results of test suite runs as input, performs a specified computation and returns the result. To that end, a metric can use any information available from the result of a test suite run, e.g. at what time the test suite run was triggered, when it finished, and further information contained in the results of test case runs bound to the test suite run.

Universal test metrics: The following four test metrics are universally applicable to any test-based certification technique, independent of particular designs of test cases, test suites or workflow. Their general applicability makes them an ideal basis to construct measures to evaluate the performance of test-based certification techniques.

1) Basic-Result-Counter (brC): This metric counts the number of times a test fails or passes. As Figure 1 shows, a test result is only returned after its execution has completed, i.e. at tsr_i^e. This metric can be used to evaluate properties only requiring to know if or how often a test failed or passed, e.g. if and how often a virtual machine is accessible through some blacklisted ports.

2) Fail-Pass-Sequence-Counter (fpsC): A *failed-passed-sequence* (fps) is a sequence of test suite run results which, after a test passed, starts with failed test and ends with the next occurrence of a passed test. As an example, consider trying to connect to a VM for ten times in a row. The first three times the login succeeds (p), then for four times, the login fails (f) and for the remaining three times the test passes again. The fps in this example is $fps_{SSH}^{10} = \langle f, f, f, f, p \rangle$.

Drawing on this definition, the *fpsC* counts the number of occurrences of fps of a particular test. Figure 1 shows the sequence $\langle f, f, f, p, p, f, f, p, p \rangle$ which contains two fps.

3) Fail-Pass-Sequence-Duration (fpsD): The metric $fpsD$ builds on the notion of fps, it measures the time between the first failed test until the next test passes. It can be used to evaluate properties over individual periods, e.g. time needed to fix a misconfigured webserver's TLS setup.

Evaluating properties over time requires to decide at which point in time a test is considered failed or passed, i.e. whether to consider start time tsr^s or end time tsr^e of the respective test. As displayed in Figure 1, we define the start of a $fpsD$ to be the start time of the first failed test tsr_i^s. For the end of a $fpsD$, we choose the start time of the next passed test tsr_{i+j}. This definition of $fpsD$ is robust against variations of a test's execution time, e.g. if a test failing takes longer than the test passing.

4) Cumulative-Fail-Pass-Seq-Duration (cfpsD): This metric builds on $fpsD$ by accumulating any measured durations until a particular point in time, thus providing a global value. This metric can, for instance, be used to describe the total downtime of a server per year.

E. Preconditions

Naively executing tests is prone to false negatives, e.g. testing a webserver's TLS configuration may fail not because of a vulnerable configuration but because the webserver cannot be reached. Computing metrics based on false negatives occurring in test suite runs will further propagate their error. Thus assumptions made about the environment of the cloud service under test, i.e. preconditions, also need to be tested.

One way to model preconditions within our framework is to design test suites to which test cases are bound serving to test preconditions, e.g. firstly establishing a virtual machine is reachable through ping and then test the TLS configuration. This allows to use preconditions to control the workflow, thus tests' execution adapt to environmental conditions discovered at runtime. Another option consists of testing preconditions as part of the main test suite run and considering the result during metric computation. For instance, when testing a webserver's TLS configuration, preconditions which check the reachability of the webserver, e.g. by issuing TCP connects, are executed concurrently. Only if the precondition pass, the result of the TLS test case will be considered when computing the test metric.

III. Evaluating test-based techniques

This section describes a method how to evaluate results of test-based certification techniques and infer conclusions about their general performance. We begin with a high-level overview of how our method works (Section III-A). Thereafter, we describe how to simulate violations of cloud services properties (Section III-B) and introduce performance measures to evaluate test-based techniques (Section III-C). Finally, we explain how to infer conclusions about test-based techniques' general performance (Section III-D).

A. Overview

We treat a test-based certification technique as a black box, i.e. we have no information how well this technique detects property violations. Put differently: Correct results and errors of a test-based technique follow some unknown distributions. We take samples from these unknown distributions by running experiments which simulate property violations. Using these experiment results, we infer conclusions about the general performance of the test-based technique.

Figure 2 provides an overview of our approach. As part of a simulation's configuration, we randomize duration of and time between each property violation event within some specified bounds (Step 1). Then the test-based technique is configured according to the framework described in the previous section: Selecting test cases, setting test suites parameter and choosing a workflow. Thereafter, the test-based technique as well as the property violation simulation are executed (Step 2). Provided our sample size is sufficiently large, i.e. the test-based technique has produced enough results (Step 3), we infer parameters of the unknown parent distribution, that is, we draw conclusions about the general performance of the test-based technique (Step 4). These inferences are considered valid with regard to the test and simulation configuration parameters.

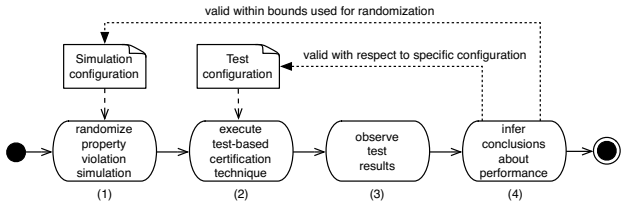

Figure 2: Evaluating performance of test-based techniques

B. Simulating property violations

A *simulation* manipulates a cloud service under test to mock violations of properties which a specific test-based certification technique aims to detect. Thus simulations are essential to establish the ground truth to which results produced by test-based certification techniques are compared.

1) Repeated property violation simulation: Since the test-based techniques whose performance we want to evaluate are repeatedly executed, simulating properties' violations need to be executed repeatedly as well. We can describe this repeated property violation simulation as a sequence $V = \langle pve_1, pve_2, \ldots, pve_i \rangle$ where each element pve is a property violation event. Each $pve \in V$ is a tuple consisting of the duration of the property violation $pveD$ and waiting time before start simulating an event $pveW$.

2) Simulation Design: The design of a simulation is driven by the property that should be tested. For example, a simulation may start and stop virtual machines to simulate violations of availability, publicly expose sensitive interfaces to mimic violations of secure service configurations, or limiting bandwidth to simulate violations of quality of service etc. The first step when designing a simulation thus consists of inspecting the property which a test-based techniques aims to check. Then identify potential violations to simulate, including the lower and upper bound of the

expected frequency of violations, i.e. size of $pveW$, and the duration of each violation $pveD$. Note that deciding on how many property violation events to simulate is driven by the selected performance measures which will be explained in detail in Section III-D.

3) Standardizing property violation events: Simulations establish the ground truth against which we evaluate specific test-based certification techniques. To infer conclusions about the general performance of a continuously executed test-based technique, ideally any possible sequence of any possible property violation event has to be simulated. Naturally, this is infeasible in practice and we have to select a sequence of property violation events V to simulate which meets our time and space constraints. But how do we select a sequence V still allowing us to draw conclusions about the general performance of a test-based certification technique?

The answer consists of two parts: At first, we need to standardize the property violation event: For each pve we use to construct V, the duration of the property violation $pveD$ and the waiting time before start $pveW$ are selected randomly from intervals $[pveD^L, pveD^R]$ and $[pveW^L, pveW^R]$, respectively. Choosing these intervals' bounds lets us configure a property violation simulation according to our space and time limitations. Secondly, we need to decide how many pve, i.e. $|V|$ are required to infer conclusions about the general performance of the test-based certification technique. This depends on the statistical inference method which, in turn, depends on the performance measure. We will address this question for each performance measure in Section III-D.

C. Performance measures

This section describes performance measures which are based on the test metrics brC, $fpsC$, $fpsD$, and $cfpsD$ introduced in Section II-D. To calculate the performance measures, we use the results produced by a test-based technique during a corresponding property violation simulation.

1) Basic-result-Counter: This section explains how to use the metric brC which counts failed and passed tests to evaluate a test-based technique. To that end, we check whether a technique's test results correctly indicated absence or presence of a simulated property violation.

True negative result: Any test produces a true negative result (br^{TN}) if the test fails at a time when a property violation is simulated. Counting all the true positive test results gives us brC^{TN}.

Pseudo true negative result: Consider the following example: A test has started measuring available bandwidth of a virtual machine and only after the test started, the bandwidth limitation is simulated. While at the first no property was violated, later during the test it was. If the test in total determines that available bandwidth was insufficient, then the test fails, producing a pseudo true negative result. This example describes a special case where a test has started before a property violation simulation starts and finishes after this violation has started. If the test fails, indicating a property violation, then we refer to this result as pseudo true negative (br^{PTN}). We describe the count of pseudo true negative test results by brC^{PTN}.

False negative result: A false negative result (br^{FN}) is observed if a test fails when there was no property violation. Counting all occurrences of false negative test results gives us brC^{FN}.

False positive result: False positive results (br^{FP}) are observed if a test passes when a property violation was simulated. brC^{FP} counts all false negative results.

Performance measures (ebrC): Based on these observations, we compute four standard performance measures for binary classification: True negative rate tnr, false positive rate fpr, false omission rate for, and negative predictive value npv:

$$tnr^{brC} = \frac{(brC^{TN}+brC^{PTN})}{(brC^{TN}+brC^{PTN}+brC^{FP})},$$

$$fpr^{brC} = \frac{brC^{FP}}{(brC^{TN}+brC^{PTN}+brC^{FP})},$$

$$for^{brC} = \frac{brC^{FN}}{(brC^{TN}+brC^{PTN}+brC^{FN})}, \text{and}$$

$$npv^{brC} = \frac{(brC^{TN}+brC^{PTN})}{(brC^{TN}+brC^{PTN}+brC^{FN})}.$$

2) Fail-Pass-Sequence-Counter: In this section, we explain how we derive performance measures using the metric $fpsC$ which counts the occurrence of fps produced by a test-based techniques. We check if and how any fps overlaps with property violation events pve.

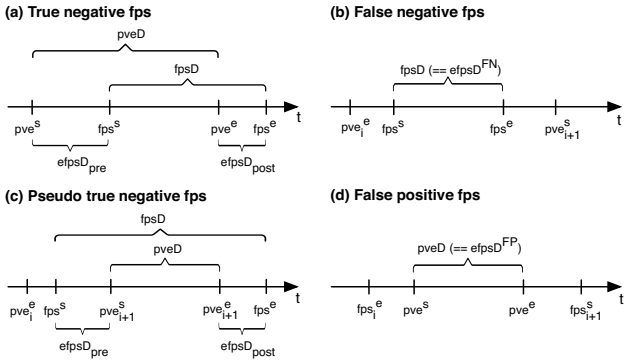

Figure 3: Performance measures based on fps and $fpsD$

True negative fps: If a fps starts (fps^s) after a pve starts (pve_i^s) and starts (fps^s) before this pve ends (pve_i^e), then fps is considered a true negative:

$$fps^{TN} = pve_i^s \leq fps^s \leq pve_i^e.$$

Note that Figure 3(a) only shows a single cve which is correctly detected by a fps. However, a single fps^{TN} can cover multiple pve if the interval between tests is a multiple of the property violation events' duration. Further, a fps^{TN} may contain a false positive test result (br^{FP}) or false negative test results (br^{FN}). The latter is the case if after the pve ended, basic results still incorrectly indicate a property

violation and are counted as failed tests of the fps^{TN}. We use $fpsC^{TN}$ to count the number of fps^{TN} observed during a property violation simulation.

False negative fps: If a fps starts after the last pve ends (pve_i^e) and ends (fps^e) before the next pve starts (pve_{i+1}^s), then this fps is considered a false negative result:

$$fps^{FN} = pve_i^e < fps^s \wedge fps^e < pve_{i+1}^s.$$

Figure 3(b) illustrates a fps^{FN}. $fpsC^{FN}$ counts all occurrence of fps^{FN} during a property violation simulation.

Pseudo true negative fps: A fps whose first failed tests are false negatives (br^{FN}) or whose first failed test is a pseudo true negative result (br^{PTN}, for details see Section III-C1). As Figure 3(c) shows, the fps starts after the last pve ends and starts before the next pve starts and ends only after the next pve starts:

$$fps^{PTN} = pve_i^e < fps^s < pve_{i+1}^s \leq fps^e.$$

The count of fps^{PTN} within a property violation simulation is described by $fpsC^{PTN}$. Note that, analogous to a true negative fps, a fps^{PTN} can cover multiple pve.

False positive fps: Figure 3(d) illustrates a pve that starts after the last fps ended and ends before the next fps starts (fps_{i+1}^s). A missed pve is considered a false positive:

$$fps^{FP} = fps_i^e < pve^s \wedge pve^e < fps_{i+1}^s.$$

We use $fpsC^{FP}$ to count occurrences of fps^{FP}.

Performance measures (efpsC): Based on $fpsC^{TN}$, $fpsC^{FN}$, $fpsC^{PTN}$, and $fpsC^{FP}$, we compute tnr, fpr, for, and npv. The calculation of these measures is analogous to those for brC presented in the previous section.

3) Fail-Pass-Sequence-Duration: This section details how to construct performance measures from the test metric $fpsD$. This metric captures the time between the start of the first failed test (fps^s), i.e. first element of a fps, and the start of the next subsequent passed test (fps^e), i.e. last element of a fps.

True negative fpsD: If we observe a true negative fps, then we can compute the difference between the duration of the fps, i.e. $fpsD = fps^e - fps^s$ and the duration $pveD$ of any property simulation events covered by the fps. Figure 3(a) shows that a fps^{TN} covers at least one pve. Yet it can, at most, cover all pve contained in the sequence V of the property violation simulation:

$$efpsD^{TN} = fpsD - \textstyle\sum_{i=1}^{|V|} pveD_i.$$

Note that we do *not* use the absolute value of $efpsD^{TN}$. This allows us to observe whether an $fpsD$ overestimates or underestimates a property violation event's duration. In order to assess the measurement error of an $efpsD^{TN}$ relative to the property violation duration, we use

$$efpsD_{rel}^{TN} = \frac{|efpsD^{TN}|}{\sum_{i=1}^{|V|} pveD_i}.$$

As Figure 3(a) illustrates, a test-based technique may be inaccurate when determining the start of a property violation event. Capturing this error, we compute the time difference between the start of an fps, i.e. when the test detected a property violation (fps^s), and the start of the simulated property violation event (pve^s), i.e. $efpsD_{pre}^{TN} = fps^s - pve^s$. Our test-based certification technique may also have an error when determining the end of a property violation event. To describe this error, we compute the time difference between the end of property violation event (pve^e) and the detected end of the property violation by the test (fps^e), i.e. $efpsD_{post}^{TN} = pve^e - fps^e$.

Pseudo true negative fpsD: If we observe a pseudo true negative fps, then computing the error is similar to true negative $fpsD$ (see Figure 3(c)): $efpsD^{PFN}$ is the difference between the duration of a pseudo true negative fps and the duration of any covered property simulation event. $efpsD_{pre}^{PFN}$ captures the difference between the observed and actual start of a cve while $efpsD_{post}^{PFN}$ describes a test-based technique's error when determining the end of a cve.

False negative fpsD: As Figure 3(b) shows, the entire duration of a false negative fps is erroneous since it incorrectly indicates a duration of a property violation event, i.e. $efpsD^{FN} = fps^e - fps^s$.

False positives fpsD: If we observe a false positive fps, i.e. the absence of a fps despite a pve occurred, the error of the missed duration is computed as follows (see Figure 3(d)): $efpsD^{FP} = pve^e - pve^s$

Performance measures (efpsD): During a property violation simulation, we may observe instances of each type of error on $fpsD$, e.g. $efpsD_{post}^{TN}$. We treat observations of each type of error on fps as separate distributions. For each of the observed distributions, we compute standard descriptive statistics, i.e. mean $\bar{x}$, median $\widetilde{x}$, standard deviation sd, min and max.

4) Fail-Pass-Sequence-Cumulative-Duration: This section shows how to derive performance measures based on the test metric $cpfsD$. This metric accumulates $fpsD$ over time, thus providing a global value.

Performance measures (ecfpsD): To describe the overall performance of a test-based technique, we calculate the absolute and relative error of the cumulative duration of true negative and pseudo true negative $fpsD$. Considering the case of $fpsD^{TN}$, we sum over the duration of any true negative $fps \in F$ which were observed during the property violation simulation V. The result is compared to the total duration of simulated property violations:

$$ecfpsD^{TN} = \textstyle\sum_{i=1}^{|F|} fpsD_i^{TN} - \sum_{j=1}^{|V|} cveD_j.$$

We then assess the measurement error of $ecfpsD^{TN}$ relative to the total property violation duration, that is,

$$ecfpsD_{rel}^{TN} = \frac{|ecfpsD^{TN}|}{\sum_{j=1}^{|V|} cveD_j}.$$

Regarding false negative and false positive $fpsD$, we simply calculate the sum of the durations of any $fpsD^{FN}$ and $fpsD^{FP}$ observed during the property violation simulation to obtain $ecfpsD^{FN}$ and $ecfpsD^{FP}$, respectively.

D. Inferring general performance

This section describes how we use the performance measures for test metrics bfC, $fpsC$, and $fpsD$ introduced in the previous section to infer conclusions about the general performance of test-based certification techniques.

1) Basic-Result-Counter: Among others, we use the negative predictive value npv on the test metric bfC to describe the performance of a test-based technique (see Section III-C1). Treating npv as a proportion allows us compute a confidence interval for npv. Considering, for example, a confidence interval of 95%, we can state that we are 95% confident that the npv of a specific test-based certification technique – with respect to the selected test and simulation configurations – is contained in the interval. We can compute this interval estimate with $npv \pm z_{95\%} \times se$ where $z_{95\%}$ is the value that separates the middle 95% of the area under the standard normal distribution, and se is the standard error which can be estimated with $se = \sqrt{p \times (1-p)/n}$. In our example for npv, the sample size n is the sum of brC^{TN}, brC^{PTN} and brC^{FN}. Using the standard normal distribution requires the sampling distribution of the proportion to be gaussian. To determine the required sample size $\tilde{n}$, the standard approach is to solve margin of error $E = z_{95\%} \times se$ for the sample size $\tilde{n}$:

$$\tilde{n} = \frac{z_{95\%} \times \widehat{npv} \times (1-\widehat{npv})}{E^2} \tag{1}$$

where $\widehat{npv}$ is an educated guess of npv proportion in the parent distribution and E is the desired margin of error. Note that choosing $\widehat{npv} = 0.5$ is the conservative option.

Recall that in Section III-B3, we posed the question how many violation events $|V|$ need to be simulated to allow for inferring conclusions about the general performance of continuously executed test-based certification techniques. Continuing our example for npv, finding the required size of V can be formulated as an optimization problem: We have to simulate at least as many property violation events pve as are required to observe $\tilde{n}$ test results. Following these steps, interval estimates for the remaining performance measures fpr^{brC}, for^{brC}, and tnr^{brC} can be computed analogously.

2) Fail-Pass-Sequence-Counter: Based on $fpsC$, we compute the same performance measures as for brC. Thus, we can use the approach described in the previous section to calculate interval estimates for the proportions of each performance measure. However, there is one important difference when determining the required number of property violation events $|V|$: We now have to simulate at least as many property violation events as are needed to observe $\tilde{n}$ fps during the property violation simulation.

3) Fail-Pass-Sequence-Duration: For each of type of error on $fpsD$, e.g. $efpsD^{TN}$, $efpsD^{PTN}_{pre}$, and $efpsD^{FN}$, we compute the mean $\bar{x}$ of the observed distribution as a performance measure. To make general statements about the errors on $fpsD$, we construct a confidence interval for each mean. As an example, consider $efpsD^{TN}$, i.e. the mean error that a test-based technique has in determining the duration of a property violation event: A confidence interval on this mean allows statements such as we are 99% confident that the average error of a specific test-based technique – constrained by the chosen test and simulation configurations – makes when estimating the duration of a property violation event is contained in the interval. We compute this estimate with $\bar{x}_{efpsD^{TN}} \pm t_{99\%} \times se$ where $t_{99\%}$ is the value that separates the middle 99% of the area under the t-Distribution and se is the standard error which can be estimated with $se = sd/\sqrt{n}$. In our example, the sample size n is the number of observed true negative fps and sd is the standard deviation. Similar to Section III-D1, to determine the required sample size $\tilde{n}$, the desired margin of error E is solved for the sample size $\tilde{n}$, that is,

$$\tilde{n} = \frac{\sigma^2 \times t^2}{E^2} \tag{2}$$

where σ^2 is an educated guess of the population variance based on initial samples of $efpsD^{TN}$ or historical values.

Inferring statements about the general performance of a test-based technique based on the mean of, e.g., $efpsD^{TN}$ requires simulating a minimum number of property violation events. In our example for $efpsD^{TN}$, we find the minimum size of V by simulating as least as many property violation events as are needed to observe $\tilde{n}$ fps^{TN}. Using these steps, interval estimates for the means of $efpsD^{TN}_{pre}$, $efpsD^{TN}_{post}$, $efpsD^{PTN}$, $efpsD^{PTN}_{pre}$, $efpsD^{PTN}_{post}$, $efpsD^{FN}$, and $efpsD^{FP}$ can be computed analogously.

IV. Application

This section presents experimental results of applying our method to evaluate and compare exemplary test-based certification techniques. We begin by outlining the components of our experiment setup (Section IV-A). Then we present two scenarios where a cloud service provider seeks certification of requirements related to availability, security and reliability (Section IV-B and IV-C).

A. Setup and environment

This section outlines tested cloud services as well as the components we used to conduct our experiments.

1) Cloud service under Test: We test IaaS instances of OpenStack Mitaka (*IaaS^{OS}*). Each of the tests presented in Sections IV-B and IV-C were executed on individual instances – in total five virtual machines – all attached to the same tenant network, each running Ubuntu 16.04 Server, with 2 vCPU, and 4 GB RAM. Machines used to test security configurations (Section IV-C) were additionally

running an *Apache* Webserver and a *MongoDB* database which we refer to as *SaaS*^{OS} and *PaaS*^{OS}, respectively.

2) Simulation framework: We implemented a lightweight framework in Java that supports simulation of property violation events as described in Section III-B. This framework runs on a designated instance, attached to the same tenant network as *IaaS*^{OS}, *PaaS*^{OS}, and *SaaS*^{OS}.

3) Test-based certification framework: Our prototype is implemented in Java and supports all elements of the framework introduced in Section II. It is deployed on an external host, in a different network than the tested cloud services.

4) Evaluation engine: This component is also built in Java and computes the performance measures described in Section III-C. We use the *Apache Commons Math* library to compute our test statistics and simulation parameter.

B. Testing resource availability

In this scenario, a cloud service provider seeks to certify requirements related to resource availability and provisioning. Such requirements can, e.g., be derived from: Control *SC-6 Resource Availability* of NIST SP 800-53 [2], *IVS-04* of the Cloud Control Matrix (CCM) upon which the CSA certificate is based [1], or *Section 6.3.7 Resource Provisioning* of ENISA IAF [3]. To that end, the provider wants to select a test-based technique which overestimates duration of detected violations of resource availability requirements as little as possible. This implies that false negative test results incorrectly indicating an availability requirement violation should be as low as possible.

1) Alternative test-based techniques: The cloud provider can select one of three candidate techniques: The first possibility is *PingTest* which pings *IaaS*^{OS} where each tests sends ten ECHO_REQUEST packets. A test passes if the returned round trip time (rrt) satisfies both of the following assertions: $assert\_rtt\_avg < 20ms$ and $assert\_rtt\_sd < 10ms$. The second possible technique uses TCP packets to determine whether *IaaS*^{OS} is available (*TCPTest*). We use *Nping*[1] to execute this test which passes if the maximum average response time and the maximum response time of probes are not greater than 75 and 100 ms, respectively. The third possibility is *SSHTest* which uses the *Trilead SSH2*[2] library to connect to *IaaS*^{OS} via SSH and then test the session using an SSH_MSG_CHANNEL_REQUEST. The test passes if no I/O exception is thrown when connecting or when session testing.

The interval of each of the three test-based techniques was configured to 60 seconds, i.e. the next test started 60 seconds after the previous one completed.

2) Simulation configuration: To evaluate our three candidate techniques, we simulated 1000 downtimes of *IaaS*^{OS}. Each event lasted at least 60 seconds plus selecting $[0, 30]$ seconds at random ($pveD$). The interval between consecutive downtimes was at least 120 seconds plus selecting $[0, 60]$ seconds at random ($pveW$). Table I shows the total downtime ($cpveD$), the mean duration of each downtime ($\bar{x}_{pveD}$) and standard deviation (sd_{pveD}) for each property violation simulation V used to evaluate *PingTest*, *TCPTest*, and *SSHTest*.

Table I: Simulation parameters used for *PingTest*, *TCPTest*, and *SSHTest*

Simulation parameter	$V^{PingTest}$	$V^{TCPTest}$	$V^{SSHTest}$
$cpveD$ (sec)	75102.11	75544.07	75706.71
$\bar{x}_{pveD}$ (sec)	75.11	75.54	75.71
sd_{pveD} (sec)	8.97	8.70	8.82

3) Test statistics: Table II summarizes the results of *PingTest*, *TCPTest*, and *SSHTest*. Besides the universal test metrics brC, $fpsC$, $fpsD$ and $cfpsD$ introduced in Section II-D, we also include the total number of executed tests ($tsrC$) as well as the mean ($\bar{x}_{tsr}$), standard deviation (sd_{tsr}), min (min_{tsr}) and max (max_{tsr}) duration of tests.

4) Performance: Table III shows the results of performance measures selected for this scenario: *TCPTest* and *SSHTest* perform perfect on npv and for since they do not return any false negative fps (Table II). Yet *SSHTest* has a relative average error of 102.47% ($\bar{x}^{rel}_{TN}$) when estimating the duration of a property violation event. This leads to a total overestimation of the simulated downtime of 8528208 ms (TN(ms)) or 11.26% (TN(%)). In context of our scenario, *SSHTest* is thus not a suitable choice.

PingTest has the lowest relative average error ($\bar{x}^{rel}_{TN}$) when measuring a simulated downtime. Yet *PingTest* on average overestimates a simulated downtime by 5725 ms ($\bar{x}_{TN}$), resulting in a total overestimation of the simulated downtime of 3503495 ms or 4.66%. The cloud provider will select the *TCPTest* because it underestimates a simulated downtime on average by 425 ms, resulting in underestimating the total simulated downtime by 963437 ms or 1.28%.

5) Inferring general performance: Table III shows margins of error E_{tnr}, E_{for}, E_{npv} or E_{TN} which we use to construct interval estimates providing conclusions about the general performance of a test-based technique. As described in Section III-D3, such inferences require a minimum sample size, i.e. a minimum number of observed brC or fps. The next two paragraphs exemplify how to calculate the minimum sample size for *PingTest* and *TCPTest*.

In the case of *PingTest*, we observed 966 true negative fps which we use to calculate a 95% confidence level for $\bar{x}_{TN}$, i.e. the average error each true negative fps makes on estimating a downtime event: 5725±893ms. As pointed out in Section III-D3, we can determine the minimum number of fps^{TN} required for this inference using formula (2). Lets assume that the observed value for the margin of error coincides with our desired value for $E^{95\%}_{TN}$: If we use the observed value for $E^{95\%}_{TN}$, the standard deviation from *PingTest* as an estimate for the population variance (σ^2) and

[1]https://nmap.org/nping/

[2]https://github.com/jenkinsci/trilead-ssh2

Table II: Summary of test statistics of *PingTest*, *TCPTest*, and *SSHTest*

Test statistic		PingTest	TCPTest	SSHTest
	$tsrC$	3153	3546	2491
tsr	$\bar{x}_{tsr}$ (sec)	12.04	4.38	30.91
	sd_{tsr} (sec)	4.51	0.47	44.90
	min_{tsr} (sec)	9.01	4.02	0.54
	max_{tsr} (sec)	20.03	5.22	127.34
brC	brC^{TN}	1002	1142	508
	brC^{PTN}	0	0	0
	brC^{FP}	4	8	479
	brC^{FN}	2	6	0
fpsC	$fpsC^{TN}$	966	983	476
	$fpsC^{PTN}$	1	4	0
	$fpsC^{FP}$	33	13	492
	$fpsC^{FN}$	1	0	0
fpsD	$\bar{x}_{TN}$ (sec)	81.23	75.28	176.96
	sd_{fpsDTN} (sec)	13.57	23.82	61.40
	$\bar{x}_{PTN}$ (sec)	139.1	145.19	0.0
	sd_{PTN} (sec)	0.0	32.06	0.0
	$\bar{x}_{FP}$ (sec)	63.74	62.98	72.99
	sd_{FP} (sec)	2.63	4.13	8.36
	$\bar{x}_{FN}$ (sec)	69.12	0.0	0.0
	sd_{FN} (sec)	0.0	0.0	0.0
cfpsD	TN (sec)	78466.5	73999.86	84234.91
	PTN (sec)	139.10	580.77	0.0
	FP (sec)	2103.27	818.69	35909.87
	FN (sec)	69.12	0.0	0.0

Table III: Evaluation of techniques to test resource availability

Performance measure		PingTest	TCPTest	SSHTest
efpsC	tnr	0.967	0.987	0.4917
	$E_{tnr}^{95\%}$	0.0111	0.007	0.0315
	for	0.001	0.0	0.0
	$E_{for}^{95\%}$	0.002	0.0	0.0
	npv	0.999	1.0	1.0
	$E_{npv}^{95\%}$	0.002	0.0	0.0
efpsD (ms)	$\bar{x}_{TN}$	5725	-425	79709
	$\widetilde{x}_{TN}$	3641	-6385	98840
	sd_{TN}	14147	22326	35413
	$E_{TN}^{95\%}$	893	1397	3189
efpsD$_{rel}$ (%)	$\bar{x}_{TN}^{rel}$	13.19	19.95	102.47
	sd_{TN}^{rel}	14.54	18.98	46.42
	$E_{rel,TN}^{95\%}$	0.92	1.19	4.18
ecfpsD	TN(ms)	3503495	-963437	8528208
	TN(%)	4.66	1.28	11.26
	FN(ms)	69124	0	0

500 degrees of freedom for the t distribution, then we obtain

$$\tilde{n} = \frac{\sigma^2 \times t^2}{E^2} = \frac{14147^2 \times 1.96^2}{893^2} \approx 965$$

required true negative fps. Thus we have to simulate sufficient property violation events to observe at least 965 fps^{TN}. Alternatively, if our desired margin of error E_{TN} was 1000 ms, then $\approx$ 769 fps^{TN} would suffice. Since we observed 966 fps^{TN}, we can state that we are 95% confident that the average error that PingTest makes on each fps^{TN}– with respect to the chosen test and simulation configurations – is between 4832 and 6618 ms.

Considering *TCPTest*, the sum of observed $fpsC^{TN}$, $fpsC^{PTN}$ and $fpsC^{FP}$ equals 1000 fps (Table II). On this basis, we compute the 95% confidence level for the true negative rate tnr, i.e. 0.987 ± 0.007. If we assume the observed value of E_{tnr} to be our desired one, then we can obtain the required sample size for confidence intervals for proportions by applying formula (1) to tnr, that is,

$$\tilde{n} = \frac{z_{95\%} \times \widehat{tnr} \times (1 - \widehat{tnr})}{E^2} = \frac{1.96 \times 0.987 \times (1 - 0.987)}{0.007^2} \approx 514.$$

This means that sufficient property violation events have to simulated to observe 514 fps. Since we observed 1000 fps, we can state that we are 95% confident that *TCPTest* – with respect to the selected test and simulation configurations – has a true negative rate between 0.98 and 0.994.

C. Testing security configurations

In this scenario, the cloud provider wants to certify requirements related to secure communication and configuration. Such requirements may stem from, e.g., *Section 6.4.5 Encryption* of ENISA IAF, *SC-8 Transmission Confidentiality and Integrity* of NIST SP 800-53 or *IVS-04* of the CCM. As an example, we assume that the provider has decided on two techniques: One tests if data transferred to the cloud services is vulnerable during transit. The other tests if the cloud service exposes vulnerable interfaces. The remaining question for the provider is now how the two test-based techniques have to be configured to correctly detect how many times configurations of interfaces and communication were vulnerable and how long it took to fix these vulnerabilities.

1) Design of test-based technique: Regarding secure communications, we use *sslyze*[3] to analyze the TLS configuration of *SaaS^{OS}* and parse its output to identify weak cipher suites (*TLSTest*). To detect vulnerable interfaces, we use *Nmap* to discover reachable ports of *PaaS^{OS}* in the range 1-65535 (*PortTest*).

2) Alternative test configurations: Here the provider can choose from three test configurations for *PortTest*, i.e. executing *PortTest* either every 10, 30 or 60 seconds. In case of the *TLSTest*, the provider does not want to fix but to randomize the waiting time until the next test starts. To that end, he can choose between configuring three intervals: $[0, 10]$, $[0, 30]$ or $[0, 60]$ seconds.

3) Simulation configuration: To evaluate our three candidate test configurations for *TLSTest*, we simulated 1000 vulnerable TLS configurations of *SaaS^{OS}* by making the weak cipher suite TLS_RSA_WITH_DES_CBC_SHA available. Each event lasted at least 60 seconds plus selecting $[0, 30]$ seconds at random. The interval between consecutive downtimes was at least 120 seconds plus selecting $[0, 60]$ seconds at random. Further, we simulated 1000 events of an exposed interface by opening port 27018 on *PaaS^{OS}*,

[3]https://github.com/nabla-c0d3/sslyze

the default port of sharded MongoDB instances which should not be publicly reachable. Duration of each property violation event and interval between events are identical to those of the vulnerable TLS configuration simulation. Table IV gives an overview of the simulation parameters.

Table IV: Simulation parameters used for *TLSTest* and *PortTest*

Simulation parameter	$\mathbf{V^{TLSTest}}$			$\mathbf{V^{PortTest}}$		
	[0,10]	**[0,30]**	**[0,60]**	**10**	**30**	**60**
$pveD$(sec)	75050.77	74817.15	75477.49	76020.15	75418.79	75070.35
$\bar{x}_{pveD}$(sec)	75.05	74.82	75.48	76.02	75.42	75.07
sd_{pveD}(sec)	8.9	8.97	9.26	8.78	9.3	8.82

4) Test statistics: Table V provides an overview of the results of *TLSTest* and *PortTest*. Note that the reason for the relatively high number of executed tests ($tsrC$) of *TLSTest* lies in randomization of intervals between successive tests.

Table V: Summary of test statistics of *TLSTest* and *PortTest*

Test statistic		**TLSTest**			**PortTest**		
		[0,10]	**[0,30]**	**[0,60]**	**10**	**30**	**60**
	$tsrC$	34801	13771	7332	22039	7501	3767
tsr (sec)	$\bar{x}_{tsr}$	1.5	1.43	1.38	0.2	0.19	0.16
	sd_{tsr}	0.59	0.62	0.46	0.17	0.13	0.17
	min_{tsr}	0.1	0.1	0.1	0.05	0.1	0.1
	max_{tsr}	19.73	19.39	19.18	2.48	2.16	2.86
brC	brC^{TN}	11701	4540	2429	7297	2477	1232
	brC^{PTN}	0	0	0	0	0	0
	brC^{FP}	104	34	21	86	20	12
	brC^{FN}	260	83	39	11	5	3
fpsC	$fpsC^{TN}$	997	996	999	1003	999	998
	$fpsC^{PTN}$	2	3	0	0	0	0
	$fpsC^{FP}$	1	1	1	1	1	2
	$fpsC^{FN}$	56	4	0	0	0	0
fpsD (sec)	$\bar{x}_{TN}$	74.54	75.39	75.41	75.31	75.17	74.42
	sd_{TN}	10.08	14.18	23.19	10.79	15.38	25.57
	$\bar{x}_{PTN}$	74.39	85.37	0.0	0.0	0.0	0.0
	sd_{PTN}	0.5	19.37	0.0	0.0	0.0	0.0
	$\bar{x}_{FP}$	87.02	73.02	84.02	61.16	76.26	66.25
	sd_{FP}	0.0	0.0	0.0	0.0	0.0	5.66
	$\bar{x}_{FN}$	5.28	18.47	0.0	0.0	0.0	0.0
	sd_{FN}	2.78	4.49	0.0	0.0	0.0	0.0
cfpsD (sec)	TN	74319.02	75091.12	75335.13	75531.65	75091.74	74268.62
	PTN	148.77	256.12	0.0	0.0	0.0	0.0
	FP	87.02	73.02	84.02	61.16	76.26	132.5
	FN	295.88	73.9	0.0	0.0	0.0	0.0

5) Performance TLSTest: Selected performance measures in Table VI show that running *TLSTest* every $[0, 30]$ seconds provides the highest true negative rate tnr and the lowest false positive rate fpr for the test metric brC, i.e. when evaluating performance of *TLSTest* using every test result observed during simulation. In turn, when considering how many simulated vulnerable TLS configurations *TLSTest* correctly detects (fps^{TN}), then running it every $[0, 10]$ seconds performs best as it has the highest tnr and lowest fpr.

Lets assume that the cloud provider prefers conservative estimates of security requirements' satisfaction, that is, the provider prefers overestimating over underestimating the duration of vulnerable TLS configurations. Thus the provider will run *TLSTest* every $[0, 30]$ seconds because on average this configurations overestimates the duration a vulnerability by 538 ms ($\bar{x}_{TN}$) whereas running *TLSTest* every $[0, 10]$ or $[0, 60]$ seconds both underestimate a property violation event on average by 509 ms or 58 ms, respectively.

6) Performance PortTest: As shown in Table VI, when evaluating performance of *PortTest* based on test metric brC, then running *PortTest* every 30 seconds is superior to the other two configurations because it has the highest true negative rate tnr and the lowest false positive rate fpr. Further, performance measures derived from $fpsC$ indicate that the *PortTest* running every 10 and 30 seconds have similar tnr and fpr while running every 60 seconds has inferior performance. Lastly, the relative error that *PortTest* makes on average ($\bar{x}^{rel}_{TN}$) when estimating duration of vulnerable interfaces increases with longer intervals between tests. Hence running *PortTest* every 10 seconds provides most accurate results.

If the cloud provider was only interested in correctly detecting the number of times an interface was vulnerable, then measuring performance based on brC is most suitable and running *PortTest* every 30 seconds would be his choice. Lets assume that in this case the provider prefers to most accurately measure how long it takes to fix a vulnerable configuration based on $fpsD$. In this case, running *PortTest* every 10 seconds is the provider's best choice. In our scenario, the provider thus cannot achieve both goals simultaneously, he has to compromise on the accuracy of either brC or $fpsD$. These results highlight how our method can reveal performance trade-offs between alternative configurations of continuously executed test-based certification techniques.

V. Related Work

A. Test-based certification of cloud services

Anisetti et al. [7][8] present a test-based certification scheme for cloud services and describe how to test five properties derived from the security guide of OpenStack. Ullah et al. [11] propose using vulnerability assessment tools to test cloud service compliance. Anisetti et al. [12][13] present a model-based approach to re-use test-based evidence for re-certification within evolving, i.e. changing services. Services are modeled as Symbolic Transition Systems from which test-models are generated. Stephanow et al. [14] propose an approach to support test-based cloud certification of opportunistic cloud providers. These are fraudulent providers who will reduce costs by only pretending to satisfy customers' requirements but only if their deception remains undetected. Our work is complementary to [7][8][11][12][13][14] as neither of them considers concrete test metrics and how to evaluate the performance of continuously executed test-based certification techniques.

Table VI: Evaluation of techniques to test security configurations

Performance measure		TLSTest			PortTest		
		[0,10]	[0,30]	[0,60]	10	30	60
ebrC	tnr	0.9912	0.9926	0.9914	0.9884	0.992	0.99
	$E_{tnr}^{95\%}$	0.0017	0.0025	0.0037	0.0025	0.0035	0.0054
	fpr	0.0088	0.0074	0.0086	0.0116	0.008	0.0096
	$E_{fpr}^{95\%}$	0.0017	0.0025	0.0037	0.0024	0.0035	0.0054
efpsC	tnr	0.999	0.999	0.999	0.999	0.999	0.998
	$E_{tnr}^{95\%}$	0.002	0.002	0.002	0.002	0.002	0.0028
	fpr	0.001	0.001	0.001	0.001	0.001	0.002
	$E_{fpr}^{95\%}$	0.002	0.002	0.002	0.002	0.002	0.0028
efpsD (ms)	$\bar{x}_{TN}$	-509	538	-58	-734	-251	-670
	$\widetilde{x}_{TN}$	-87	480	-653	-324	330	-9036
	sd_{TN}	4786	10728	21185	6027	11718	24192
	$E_{TN}^{95\%}$	297	667	1315	373	727	1502
efpsD$_{\text{rel}}$	$\bar{x}_{TN}^{rel}(\%)$	4.86	11.51	23.17	5.10	12.32	25.06
	$sd_{TN}^{rel}(\%)$	4.38	8.95	16.56	6.09	9.70	18.21
	$E_{rel,TN}^{95\%}(\%)$	0.27	0.56	1.03	0.38	0.6	1.13
ecfpsD	TN(sec)	-582.9	530.1	-142.4	-488.5	-327.1	-801.7
	$TN(\%)$	0.78	0.71	0.19	0.64	0.43	1.06
	FP(sec)	87.02	73.1	84.02	61.2	76.3	132.5

B. Test-based certification techniques

Focusing on specific test-based techniques, Huang et al. [10] propose an approach to detect a provider who is cheating on providing CPU resources to customers. Albelooshi et al. [9] present a test to verify memory and disk sanitization of virtual machines which can be used to support certification of customer requirements regarding unwanted data remanence. Juels and Oprea propose an approach to verify data stored in a cloud using dynamic Proofs of Retrievability (PoR) [15]. Our work is orthogonal to [9][10][15] since they can use our approach to evaluate the performance of their test-based techniques when continuously executed.

VI. Conclusion and future work

In this paper, we introduced a method to evaluate the performance of continuous test-based cloud service certification. Our methods allows to evaluate and compare continuously executed test-based certification techniques and it supports inference of conclusions about a test-based technique's performance in general.

One drawback of our approach lies in the arbitrary selection of duration and frequency of property violation events. As part of future work, we will develop a method to design property violation simulations based on real world requirements, e.g. rare events, and extend our approach accordingly. Also, we will explore how to measure the overhead imposed on cloud services under test, especially when facing multiple continuous tests. This will allow us to select test-based techniques and configurations which incur minimal overhead while retaining required accuracy of test results.

Acknowledgment

This work was partly funded by the Federal Ministry of Education and Research of Germany, within the project *NGCert* (http://www.ngcert.eu), Grant No. 16KIS0075K, and by the EU H2020 project *EU-SEC*, Grant No. 731845.

References

[1] Cloud Security Alliance (CSA), "Security, Trust and Assurance Registry (STAR)." https://cloudsecurityalliance.org/star/certification/.

[2] National Institute of Standards and Technology (NIST), "Security and privacy controls for federal information systems and organizations," *Special Publication*, vol. 800, p. 53, 2013.

[3] D. Catteddu, G. Hogben, *et al.*, "Cloud computing information assurance framework," *European Network and Information Security Agency (ENISA)*, 2009.

[4] B. S. Kaliski Jr and W. Pauley, "Toward risk assessment as a service in cloud environments," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pp. 13–13, USENIX Association, 2010.

[5] Cloud Security Alliance (CSA), "Cloud Control Matrix: Seurity Controls Framework for Cloud Providers & Consumers." https://cloudsecurityalliance.org/research/ccm/, 2013.

[6] S. Cimato, E. Damiani, F. Zavatarelli, and R. Menicocci, "Towards the certification of cloud services," in *9th World Congress on Services (SERVICES)*, pp. 92–97, IEEE, 2013.

[7] M. Anisetti, C. A. Ardagna, E. Damiani, F. Gaudenzi, and R. Veca, "Toward Security and Performance Certification of OpenStack," in *8th International Conference on Cloud Computing (CLOUD)*, pp. 564–571, IEEE, 2015.

[8] M. Anisetti, C. Ardagna, F. Gaudenzi, and E. Damiani, "A certification framework for cloud-based services," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC)*, pp. 440–447, ACM, 2016.

[9] B. Albelooshi, K. Salah, T. Martin, and E. Damiani, "Experimental Proof: Data Remanence in Cloud VMs," in *8th International Conference on Cloud Computing (CLOUD)*, pp. 1017–1020, IEEE, 2015.

[10] Q. Huang, L. Ye, X. Liu, and X. Du, "Auditing CPU Performance in Public Cloud," in *9th World Congress on Services (SERVICES)*, pp. 286–289, IEEE, 2013.

[11] K. W. Ullah, A. S. Ahmed, and J. Ylitalo, "Towards Building an Automated Security Compliance Tool for the Cloud," in *12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pp. 1587–1593, IEEE, 2013.

[12] M. Anisetti, C. A. Ardagna, and E. Damiani, "Security certification of composite services: a test-based approach," in *International Conference on Web Services (ICWS)*, pp. 475–482, IEEE, 2013.

[13] M. Anisetti, C. A. Ardagna, and E. Damiani, "A Test-Based Incremental Security Certification Scheme for Cloud-Based Systems," in *12th International Conference on Services Computing (SCC)*, pp. 736–741, IEEE, 2015.

[14] P. Stephanow, G. Srivastava, and J. Schütte, "Test-based cloud service certification of opportunistic providers," in *8th IEEE International Conference on Cloud Computing (CLOUD)*, July 2016.

[15] A. Juels and A. Oprea, "New approaches to security and availability for cloud data," *Communications of the ACM*, vol. 56, no. 2, pp. 64–73, 2013.